Overloading vs Overriding

Overloading

Overloading is a feature that allows a class to have multiple methods with the same name, the only difference lies in their list of arguments. i.e. The argument list for each of the methods differ, and it helps the compiler or the run-time environment to identify which method to call depending upon the parameters passed. Constructors can be overloaded too.

```
#include <bits/stdc++.h>
using namespace std;

class Math
{
    public:
        //Overloaded add() methods
        static int add(int x, int y) { return x + y; }
        static void add(int a[], int b[], int sum[], int n) {
            for (int i=0;i<n;i++)
                //use the 1st form to get addition of
                //two numbers
                sum[i] = add(a[i], b[i]);
        }

        //Overloaded mul() methods
        static int mul(int x, int y) { return x * y; }
        static void mul(int a[], int b[], int prod[], int n)
            for (int i=0;i<n;i++)
                //use the 2nd form to get product
                //of two numbers
                prod[i] = mul(a[i], b[i]);
        }
};

int main()
{
    int a[] = {1, 2, 3, 4, 5};
    int b[] = {9, 8, 7, 6, 5};
    int sum[5], prod[5];

    Math::add(a, b, sum, 5);
    Math::mul(a, b, prod, 5);

    for (int i=0; i<5; i++)
        cout << sum[i] << " ";
    cout << endl;

    for (int i=0; i<5; i++)
        cout << prod[i] << " ";
    cout << endl;

    return 0;
}
```

Output:

```
10 10 10 10 10
9 16 21 24 25
```

As shown in the above program, we have 2 sets of overloaded functions, namely integer addition-array addition and integer product-array-product (sort of dot product). Depending on the type of arguments passed, the compiler decides which one of the set of overloaded functions it needs to call.

```
#include <bits/stdc++.h>
using namespace std;

class Math
{
    public:
        //Overloaded add() methods
        static int add(int x, int y) {
            cout << "1st form called: ";
            return x + y;
        }
        static double add(int x, double y) {
            cout << "2nd form called: ";
            return x + y;
        }

        static double add(double x, double y) {
            cout << "3rd form called: ";
            return x + y;
        }
};

int main()
{
    cout << Math::add(2, 3) << endl;
    cout << Math::add(2, 3.0) << endl;
    cout << Math::add(2.0, 3.0) << endl;
    return 0;
}
```

Output:

```
1st form called: 5
2nd form called: 5
3rd form called: 5
```

NOTE: Return-type is not a factor of uniqueness: Having the list of arguments identical with different return-types doesn't remove ambiguity. Hence, the following declarations are invalid:

```
int add(int x, int y) { ... }
double add(int x, int y) { ... }
```

Overriding

Inheritance allows Derived Classes to inherit Base class data-members as well as member functions. Thus, if we call base class function with derived class instance, it would run perfectly:

```
#include <bits/stdc++.h>
using namespace std;

class Base {
    public:
        void whoami() {
            cout << "I'm Base Class!!\n";
        }
};

class Derived : public Base {

};

int main()
{
    Base b;
    Derived d;

    b.whoami();
    d.whoami();

    return 0;
}
```

Output:

```
I'm Base Class!!
I'm Base Class!!
```

In the above program, we see no definition of *whoami( )* method inside the Derived Class. So when we call *d.whoami()* , the compiler first looks into the Derived Class for its definition. Then, it looks inside its Parent class, where it finds the definition and that version is called.

However, suppose we want to change the behavior of the inherited method inside our Derived Class. This feature provided by OOP is called Overriding. To do that, we provide the full method definition as usual:

```
#include <bits/stdc++.h>
using namespace std;

class Base {
    public:
        void whoami() {
            cout << "I'm Base Class!!\n";
        }
};

class Derived : public Base {
    public:
        void whoami() {
            cout << "I'm Derived Class!!\n";
        }
};

int main()
{
    Base b;
    Derived d;

    b.whoami();
    d.whoami();

    return 0;
}
```

Output:

```
I'm Base Class!!
I'm Derived Class!!
```

Sometimes, we don't want to replace the complete functionality of the inherited methods, but add some extra functionality to it. We can do so by calling the Base Class's method from the Derived class and then continue with our overriding added statements.

```cpp
#include <bits/stdc++.h>
using namespace std;

class Base {
    public:
        void whoami() {
            cout << "I'm Base Class!!\n";
        }
};

class Derived : public Base {
    public:
        void whoami() {
            // call Base class's version
            Base::whoami();
            cout << "I'm Derived Class!!\n";
        }
};

int main()
{
    Derived d;

    d.whoami();

    return 0;
}
```

Output:

```
I'm Base Class!!
I'm Derived Class!!
```

We call the Base class's *whoami( )* using the scope-resolution operator with the classname preceding it.