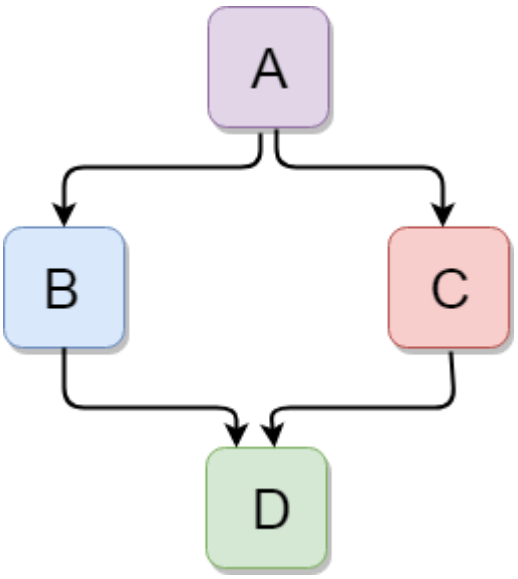
In C++, what's the diamond problem, and how can it be avoided?

Taking a look at the graphic below helps in explaining the diamond problem.



Suppose we have 2 classes B and C that derive from the *same* class – in our example above it would be class A. We also have class D that derives from *both* B and C by using multiple inheritance

You can see in the figure above that the classes essentially form the shape of a diamond – which is why this problem is called the diamond problem. Now, let's take the graphic above and make it more concrete by translating it into actual code:

main.cpp

```cpp
#include <iostream>
using namespace std;
class Animal{
public:
    Animal(){
        cout << "animal constructor" << endl;
    }
    int age;
    void walk(){
        cout << "animal walks" << endl;
    }
};
```

```
13
14  class Tiger : public Animal{
15  public:
16      Tiger(){
17          cout << "constructor of tiger" << endl;
18      }
19  };
20
21  class Lion : public Animal{
22  public:
23      Lion(){
24          cout << "constructor of Lion" << endl;
25      }
26  };
27
28  class Liger : public Tiger , public Lion{
29  public:
30      Liger(){
31          cout << "constructor of Liger" << endl;
32      }
33  };
```

In the code above, we've given a more concrete example of the diamond problem. The Animal class corresponds to the topmost class in the hierarchy (A in our graphic above), Tiger and Lion respectively correspond to B and C in the graphic, and the Liger class corresponds to D.

Now, the question is what is the problem with having an inheritance hierarchy like this. Take a look at the code below so that we can best answer that question:

```
35  int main()
36  {
37      Liger li;
38      li.walk();
39      return 0;
40  }
```

In our inheritance hierarchy, we can see that both the Tiger and Lion classes derive from the Animal base class. And here is the problem: **because Liger derives from both the Tiger and Lion classes – which each have their own copy of the data members and methods of the Animal class- the Liger object "li" will contain *two* subobjects of the *Animal* base class.**

So, you ask, what's the problem with a Liger object having 2 **sub**-objects of the Animal class? Take another look at the code above – the call "lg.walk()" will result in a compiler error. This is because the compiler **does not know whether the call to getWeight refers to the copy of** walk **that the Liger object lg inherited through the Lion class or the copy that lg inherited through the Tiger class**. So, the call to walk in the code above is ambiguous and will not get past the compiler.

input

```
Compilation failed due to following error(s).
main.cpp: In function 'int main()':
main.cpp:38:10: error: request for member 'walk' is ambiguous
     anil.walk();
          ^~~~
main.cpp:9:10: note: candidates are: void Animal::walk()
     void walk(){
          ^~~~
main.cpp:9:10: note:                 void Animal::walk()
```

How to Remove Diamond Problem in C++?

We've explained the diamond problem, but now we want to give you a solution to the diamond problem. If the inheritance from the Animal class to both the Lion class and the Tiger class is marked as **virtual**, then C++ will ensure that only one subobject of the Animal class will be created for every Liger object. This is what the code for that would look like:

```
14  class Tiger : virtual public Animal{
15  public:
16      Tiger(){
17          cout << "constructor of tiger" << endl;
18      }
19  };
20
21  class Lion : virtual public Animal{
22  public:
23      Lion(){
24          cout << "constructor of Lion" << endl;
25      }
26  };
27
```

```
animal constructor
constructor of tiger
constructor of Lion
constructor of Liger
animal walks
```