# MULTIPLE INHERITANCE

# Multiple Inheritance

We'll finish our current topic by demonstrating multi-inheritance. We need to emphasize that using this technique is commonly recognized as error-prone and obfuscating class hierarchy.

Any solution that avoids multi-inheritance is generally better and in fact many contemporary object programming languages don't offer multi-inheritance at all. We think it's a good argument to consider when you're making design assumptions.

# Multiple Inheritance (contd…)

```
1. class A
2. {
3.     public:
4.         A() { cout << "A's constructor called" << endl; }
5. };
6.
   class B
7. {
8. public:
9.     B() { cout << "B's constructor called" << endl; }
10.};
11.
   class C: public B, public A // Note the order
12.{
13.    public:
14.        C() { cout << "C's constructor called" << endl; }
15.};
16.
   int main()
17.{
18.    C c;
19.    return 0;
20.}
```

**Output**

```
B's constructor called
A's constructor called
C's constructor called
```

# Multiple Inheritance - Ambiguity

This example should be clear (we hope).

The program will produce the following output:

storage = 3

safe = 5

```
#include <iostream>
using namespace std;
class SuperA {
protected:
      int storage;
public:
      void put(int val) { storage = val; }
      int get(void) { return storage; }
};
class SuperB {
protected:
      int safe;
public:
      void insert(int val) { safe = val; }
      int takeout(void) { return safe; }
};
class Sub : public SuperA, public SuperB {
public:
      void print(void) {
            cout << "storage = " << storage << endl;
            cout << "safe    = " << safe << endl;
      }
};
int main(void) {
      Sub object;

      object.put(1);       object.insert(2);
      object.put(object.get() + object.takeout());
      object.insert(object.get() + object.takeout()
);
      object.print();
      return 0;
}
```

# Multiple Inheritance - Ambiguity

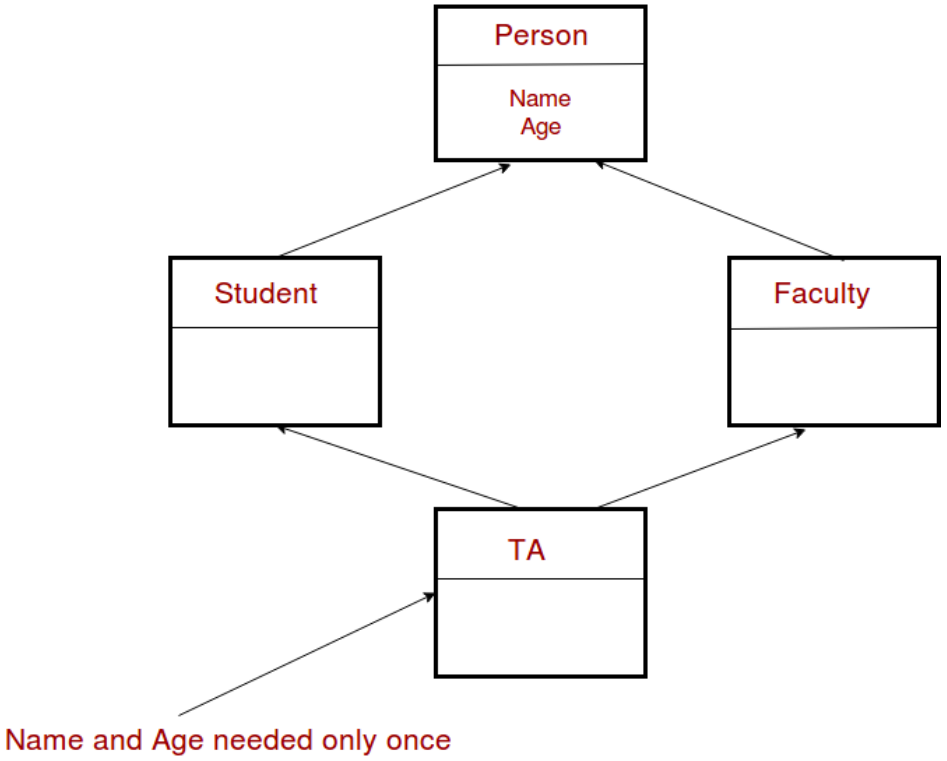
```
class SuperA
{
    protected:
        int storage;
    public:
        void put(int val){ storage = val; }
        int get(){ return storage; }
};

class SuperB
{
    protected:
        int safe;
    public:
        void put(int val){ safe = val; }
        int get(){ return safe; }
};
```

```
class Sub: public SuperB, public SuperA
{
    public:
        void print()
        {
            cout<<"Storage: "<<storage<<endl;
            cout<<"Safe: "<<safe<<endl;
        }
};

int main()
{
    Sub s;

    s.SuperA::put(5);
    cout<<s.SuperA::get()<<endl;
    return 0;
}
```

**Output:** 5

# Multiple Inheritance – Diamond Problem

- The diamond problem occurs when two superclasses of a class have a common base class.
- For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities



# Multiple Inheritance – Diamond Problem

```
class Person {
// Data members of person
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
};

class Faculty : public Person {
// data members of Faculty
public:
    Faculty(int x):Person(x) {
    cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : public Person {
// data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};
```

```
class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}
```

## Output

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

# Diamond Problem – The Solution

```
class Person {
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
    Person()      { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x) {
    cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};
```

```
class TA : public Faculty, public Student {
public:
   TA(int x):Student(x), Faculty(x), Person(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}
```

## Output

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```