[1] pure virtual constructor and destructor

```
#include <iostream>
using namespace std;

class Animal {
public:
    virtual void speak() const = 0;
    virtual ~Animal() {}
};
class Lion:public Animal {
public:
    void speak() const override {
        cout<<"roars"<<endl;
    }
};
class Elephant : public Animal {
public:
    void speak() const override {
        cout<<"trumpets"<<endl;
    }
};
class Monkey : public Animal {
public:
    void speak() const override {
        cout<<"chatters";
    }
};
int main() {
    int zooSize=3;
    Animal* zoo[zooSize];
    zoo[0]=new Lion();
    zoo[1]=new Elephant();
    zoo[2]=new Monkey();
    for (int i=0; i<zooSize;i++) {
        zoo[i]->speak();
    }



    return 0;
}
```

[2]

```
#include <iostream>
#include <cstring>
using namespace std;
class String {
private:
    char* s;
    int size;
public:
    String(char*); // constructor
    ~String();     // destructor

    void display() const;
};
String::String(char* c) {
    size=strlen(c);
    s=new char[size + 1];
    strcpy(s, c);
}
String::~String() {
    delete[] s;
}
void String::display() const {
    cout << s << endl;
}
int main() {
    char text[] = "sunny";
    String str(text);
    cout << "5trring is: ";
    str.display();
    return 0;
}
```

[3] constructor and destructor

```
#include <iostream>
using namespace std;
class base{
    public:
```

```
    base(){
    cout<<"constructor base"<<endl;
    }
    ~base(){
        cout<<"destructor base"<<endl;
    }

};
class derived: public base{
    public:
    derived()
    {
      cout<<"constructor derived"<<endl;
    }
    ~derived(){
        cout<<"destructor derived"<<endl;
    }


};
int main() {
   derived *d=new derived();
   base *b=d;
   delete b;
   getchar();
    return 0;
}
```

[4]
Where you use all types of constructor
And destructor

```
#include <iostream>
#include <string>
using namespace std;
class Person {
private:
    string name;
    int age;
public:
    Person() {                      // Default constructor
        name="rohit sharma";
        age=37;
        cout<<"defaultconstructor"<<endl;
    }
    Person(string n, int a) {                // Parameterized constructor
        name n;
        age = a;
        cout << "Parameterized constructor"<<endl;
    }
    ~Person() {                              // Destructor
        cout << "Destructor  " << name <<endl;
    }
    void display() {
        cout << "Name is: " << name << "Age is: " << age <<endl;
    }
};

int main() {
    Person person1;
    person1.display();
    Person person2("sunny", 23);
    person2.display();
    return 0;
}
```

[5]

```
// CPP program without virtual destructor
// causing undefined behavior

#include<iostream>
using namespace std;

class base {
public:
    base() {
        cout <<"baseConstructing"<<endl;
    }
    ~base() {
        cout <<"Destructing base"<<endl;
    }
};

class derived : public base {
public:
    derived() {
        cout << "Constructing derived "<<endl;
    }
    ~derived() {
        cout << "Destructing derived "<<endl;
    }
};

int main() {
    base *b = new derived();
    delete b;
    return 0;
}
```

FRIEND FUNCTION

```
#include <iostream>
class A {
private:
    int a;

public:
    A() { a = 0; }
    friend class B; // Friend Class
};

class B {
private:
    int b;

public:
    void showA(A& x) {
        // Since B is a friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};

int main() {
    A a;
    B b;
    b.showA(a);
    return 0;
}
```

[7]

```
#include <iostream>

class B;

class A {
public:
   void showB(B&);
};

class B {
private:
   int b;
public:
   B() { b = 0; }
   friend void A::showB(B& x);
};

void A::showB(B& x) {
   // Since showB() is a friend of B, it can
   // access private members of B
   std::cout << "B::b = " << x.b;
}

int main() {
   A a;
   B x;
   a.showB(x);
   return 0;
}
```

[8]

```
/*You have a TemperatureSensor class that measures temperature in Celsius. You want a
separate DisplayTemperature function to print the temperature in Fahrenheit. However, the
conversion formula requires accessing the private celsius member.

Create a TemperatureSensor class with a private celsius member and a public constructor.
Implement a friend function DisplayTemperature that takes a TemperatureSensor object and
prints the temperature in Fahrenheit (conversion formula provided).
Write a main function to demonstrate how to use the classes.*/
```

```
#include<iostream>
using namespace std;
class temp{
private:
    float c;
public:
    temp(float temp):c(temp) {}
    friend void printtemp(const temp&);         // Friend function declaration
};
void printtemp(const temp& sensor) {
    float f=(sensor.c * 9.0 / 5.0)+32;
    cout<<"fahrenheit temperature: " << f << endl;
}

int main(){
    temp sensor(67);
    printtemp(sensor);
    return 0;
}


/*Friend Class for Stream Insertion:

Scenario: You have a Point class with private members for x and y coordinates. You want to
define a way to easily print Point objects to output streams like cout.

Create a Point class with private x and y members and a public constructor.
Design a friend class PointOutputStream that has an overloaded << operator to format and
insert Point objects into output streams.
In main, demonstrate creating Point objects and printing them using cout.*/

#include <iostream>
using namespace std;
class Point {
private:
    int x, y;
public:
    Point(int xVal, int yVal) : x(xVal), y(yVal) {}
    friend class PointOutputStream;
};

class PointOutputStream {
public:
    friend ostream& operator<<(ostream& os, const Point& point) {
        os << "Point(" << point.x << ", " << point.y << ")";
        return os;
    }
};
int main() {
    Point p1(3, 4);
    Point p2(7, 8);
    cout << p1 << endl;
    cout << p2 << endl;
    return 0;
}
```