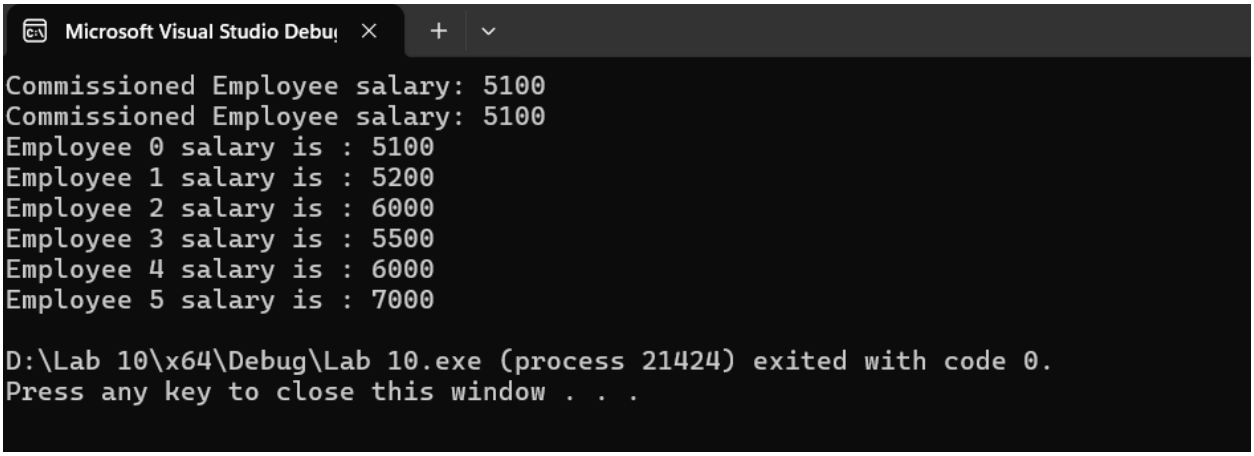
Experiment 10

Task 5

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

class Employee {
protected:
    int empId;
    float basicSalary;

public:
    Employee(int id, float salary) : empId(id), basicSalary(salary) {}
    virtual ~Employee() {}

    virtual float calculateSalary() {
        return basicSalary;
    }
};

class CommissionedEmployee : public Employee {
    float salesAmount;
    float rate;

public:
    CommissionedEmployee(int id, float salary, float sales, float commissionRate)
        : Employee(id, salary), salesAmount(sales), rate(commissionRate) {}

    float calculateSalary() {
        return (salesAmount * rate / 100) + basicSalary;
    }
};

class HourlyEmployee : public Employee {
    float payPerHour;
    float extraHours;

public:
    HourlyEmployee(int id, float salary, float payHour, float extraHrs)
        : Employee(id, salary), payPerHour(payHour), extraHours(extraHrs) {}

    float calculateSalary() {
        return basicSalary + (payPerHour * extraHours);
    }
};
```

```
class RegularEmployee : public Employee {
    float bonus;

public:
    RegularEmployee(int id, float salary, float bonusAmount)
        : Employee(id, salary), bonus(bonusAmount) {}

    float calculateSalary() {
        return basicSalary + bonus;
    }
};

int main() {
    // CASE 1 - derived Class Pointer pointing to Derived class object
    CommissionedEmployee E1(25, 5000, 1000, 10);
    CommissionedEmployee* ptr;
    ptr = &E1;
    cout << "Commissioned Employee salary: " << ptr->calculateSalary() << endl;

    // CASE 2 - Base Class Pointer pointing to Derived class object
    Employee* eptr;
    eptr = &E1;
    cout << "Commissioned Employee salary: " << eptr->calculateSalary() << endl;

    // Create other employees
    CommissionedEmployee E2(25, 5000, 1000, 10);
    CommissionedEmployee E3(26, 5000, 2000, 10);
    HourlyEmployee H1(27, 5000, 10, 100);
    HourlyEmployee H2(28, 5000, 5, 100);
    RegularEmployee R1(29, 5000, 1000);
    RegularEmployee R2(29, 5000, 2000);

    Employee* list[6];
    list[0] = &E2;
    list[1] = &E3;
    list[2] = &H1;
    list[3] = &H2;
    list[4] = &R1;
    list[5] = &R2;

    for (int i = 0; i < 6; i++) {
        cout << "Employee " << i << " salary is : " << list[i]->calculateSalary()
<< endl;
    }
```

```
    return 0;
}
```



Task 3

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

class Animal {
public:
   virtual void speak() = 0;
};

class Dog : public Animal {
public:
   void speak() {
       cout << "Woof!" << endl;
   }
};

class Cat : public Animal {
public:
   void speak() {
       cout << "Meow!" << endl;
   }
};
class Cow :public Animal {
public:
   void speak() { cout << "Moo moo\n"; }
```

```
};
int main() {
   // Animal anim; // This line will error out since Animal is an abstract class
   Animal* pAnim;
   Cat mano;
   Dog qatmeer;
   Cow C;

   pAnim = &mano;
   pAnim->speak();

   pAnim = &qatmeer;
   pAnim->speak();

   pAnim = &C;
   pAnim->speak();

   return 0;
}
```

```
Meow!
Woof!
Moo moo

D:\Lab 10\x64\Debug\Lab 10.exe (
Press any key to close this wind
```

Task 4

```
#include <iostream>

// Task3 classes
class Animal {
public:
   virtual void speak() = 0;
   virtual ~Animal() {}
};

class Dog : public Animal {
```

```
public:
    void speak() override {
        std::cout << "Woof!" << std::endl;
    }
};

class Cat : public Animal {
public:
    void speak() override {
        std::cout << "Meow!" << std::endl;
    }
};

class Cow : public Animal {
public:
    void speak() override {
        std::cout << "Moo!" << std::endl;
    }
};

class AnimalFarm {
public:
    AnimalFarm();
    AnimalFarm(int pCap);
    ~AnimalFarm();

    void AddAnimal(Animal* ptr);
    void AnimalsTalk();

private:
    Animal** aPtr;
    int capacity;
    int currentCount;
};

AnimalFarm::AnimalFarm() {
    currentCount = 0;
    capacity = 5;
    aPtr = new Animal * [capacity];
}

AnimalFarm::AnimalFarm(int pCap) {
    capacity = pCap;
    aPtr = new Animal * [capacity];
    currentCount = 0;
```

```
}

AnimalFarm::~AnimalFarm() {
   for (int i = 0; i < currentCount; ++i) {
       delete aPtr[i]; // Deallocate memory for each animal
   }
   delete[] aPtr; // Deallocate memory for the array of animal pointers
}

void AnimalFarm::AddAnimal(Animal* ptr) {
   if (currentCount < capacity) {
       aPtr[currentCount] = ptr;
       currentCount++;
   }
   else {
       std::cout << "Full capacity, can't add another animal" << std::endl;
   }
}

void AnimalFarm::AnimalsTalk() {
   for (int i = 0; i < currentCount; ++i) {
       aPtr[i]->speak();
   }
}

int main() {
   AnimalFarm* Frm = new AnimalFarm();
   Cat* c1 = new Cat();
   Cat* c2 = new Cat();
   Dog* d = new Dog();
   Cow* cw1 = new Cow();
   Cow* cw2 = new Cow();

   Frm->AddAnimal(c1);
   Frm->AddAnimal(d);
   Frm->AddAnimal(cw1);
   Frm->AddAnimal(c2);
   Frm->AddAnimal(cw2);

   Frm->AnimalsTalk();

   delete Frm;

   return 0;
}
```

Meow!

Woof!

Moo!

Meow!

Moo!