ECE 220: Computer Systems & Programming

Lecture 21: Intro to C++: Inheritance and Polymorphism

Thomas Moon

April 9, 2024

Reference

- Alias for a variable/object.
- A variable can be declared as reference by '&' in the declaration.
- A reference must be initialized when declared.

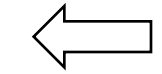
```
int val = 10;

int *ptr = &val;// & to get address
int &ref = val;// & to declare reference

cout<<val<<endl;
cout<<*ptr<<endl;
cout<<ref<<endl;

ref = 20;
cout<<val<<endl;

val = 30;
cout<<ref<<endl;
```

Pass by Pointer(address) vs by Reference

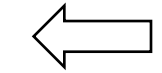
```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void swap(int &a, int &b){
    int temp = a;
    a = b;
    b = temp;
}
```

```
int val1, val2;
val1 = 10, val2 = 20;
swap(&val1, &val2);
```



```
swap(val1, val2);
```



Which function is called?

Which function can possibly cause “segment fault”?

Arrays & Pointers & Objects

- Array of objects

```
Person p[2] = {Person("Alice", 20), Person("Bob", 22) };
p[0].ShowData();
p[1].ShowData();
```

- Array of pointers to objects

```
Person *ptr[2];
ptr[0] = new Person("Alice",20);
ptr[1] = new Person("Bob",22);
ptr[0]->ShowData();
ptr[1]->ShowData();
```

- Reference to objects

```
Person &ref = p[0];
ref.ShowData();
```

```
Person &ref = *ptr[0];
ref.ShowData();
```

C++

- **Object Oriented Programming** (OOP)
  Programming style associated with **class** and **objects** and other concepts like
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

Inheritance – Why?

```
class Airplane{
private:
    int passenger;
    double baggage;
    int crew_man;
public:
    Airplane(int person, double weight, int crew){
        passenger = person;
        baggage = weight;
        crew_man = crew;
    }
    void Ride(int person){
        passenger += person;
    }
    void Load(double weight){
        baggage += weight;
    }
    void TakeCrew(int crew){
        crew_man += crew;
    }
};
```

```
class Train{
private:
    int passenger;
    double baggage;
    int length;
public:
    Train(int person, double weight, int len){
        passenger = person;
        baggage = weight;
        length = len;
    }
    void Ride(int person){
        passenger += person;
    }
    void Load(double weight){
        baggage += weight;
    }
    void AddLength(int len){
        length += len;
    }
};
```

'Airplane' and 'Train' share many data and functions!

base class

```
class Vehicle{
private:
    int passenger;
    double baggage;
public:
    void Ride(int person){passenger += person;}
    void Load(double weight){baggage += weight;}
    int getPassenger(){ return passenger;}
    double getBaggage(){ return baggage;}
};
```

derived class

```
class Airplane : public Vehicle{
private:
    int crew_man;
public:
    Airplane(int crew) {crew_man = crew;}
    void TakeCrew(int crew){crew_man += crew;}
    int getCrew(){  return crew_man;}
    void ShowData(){
        cout<<"<<Airplane>> "<<endl;
        cout<<"passenger:"<<getPassenger()<<endl;
        cout<<"baggage:"<<getBaggage()<<endl;
        cout<<"crew man:"<<getCrew()<<endl;
    }
};
```

1. *Airplane class is inherited from Vehicle class.*

2. `public` *inheritance:*
   it makes `public` *members in base* `public` *members in derived*

3. `private` *inheritance:*
   it makes `public` *members in base* `private` *members in derived*

Warning: `private` *mambers in base are NOT accessiable in derived class.*

Inheritance Access Control

```
class Base{
    private:
        int x;
    public:
        int y;
};

class DPublic : public Base{
    public:
        int getX(){return x;}
};

class DPrivate: private Base{
    public:
        int getX(){return x;}
        int getY(){return y;}
};
```

getX

← Compile Error!

← Compile Error!

← This is ok

DPublic object
Base's private member: x
DPublic's public member: y

DPrivate object
Base's private member: x
DPrivate's private member: y

```
int main(){
    DPublic a;
    cout<<a.y<<endl;
    cout<<a.x<<endl;

    DPrivate b;
    cout<<b.y<<endl;
}
```

← This is ok

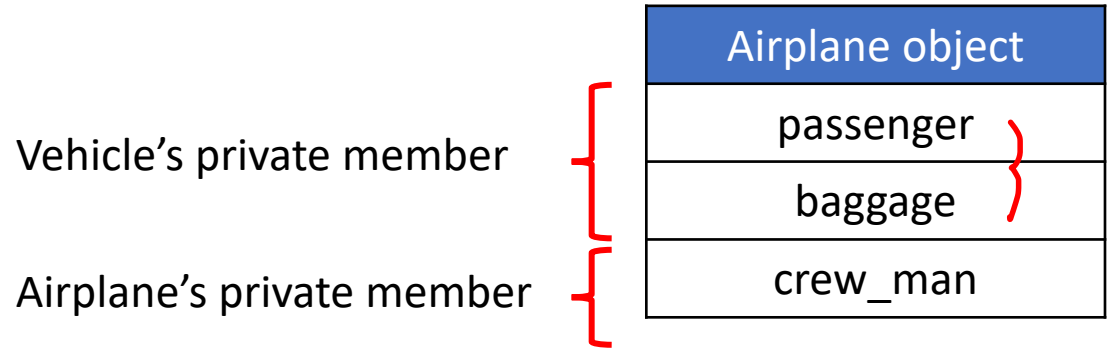
← Compile Error!

← This is now Compile Error!

Inheritance

```
class Airplane : public Vehicle{
private:
    int crew_man;
public:
    Airplane(int crew) {crew_man = crew;}
```

```
int main(){
    Airplane a(10);
    a.ShowData();
}
```




```
<<Airplane>>
passenger: 4196928
baggage: 2.07321e-317
crew man: 10
```

How do we initialize the members from the base class?

compile error!
because the members are private.

```
Airplane(int person, double weight, int crew) {
    passenger = person;
    baggage = weight;
    crew_man = crew;
}
```

Constructors: Base vs Derived

```
class Base{
    public:
        Base(){
            cout<<"Base() called."<<endl;
        }
        Base(int a){
            cout<<"Base(int a) called."<<endl;
        }
};
class Derived: public Base{
    public:
        Derived(){
            cout<<"Derived() called."<<endl;
        }
        Derived(int a){
            cout<<"Derived(int a) called."<<endl;
        }
};
```

```
int main(){
    cout<<"<<d1 declared.>>"<<endl;
    Derived d1;

    cout<<"<<d2 declared.>>"<<endl;
    Derived d2(1);
}
```

```
<<d1 declared.>>
Base() called.
Derived() called.

<<d2 declared.>>
Base() called.
Derived(int a) called.
```

1. The base constructor is called, then the derived one.
2. The default constructor is called for the base class.

How can we call `Base(int a)`?

Constructors: Base vs Derived

```
class Base{
    public:
        Base(){
            cout<<"Base() called."<<endl;
        }
        Base(int a){
            cout<<"Base(int a) called."<<endl;
        }
};
class Derived: public Base{
    public:
        Derived(){
            cout<<"Derived() called."<<endl;
        }
        Derived(int a): Base(a){
            cout<<"Derived(int a) called."<<endl;
        }
};
```

Initializer list

```
int main(){
    cout<<"<<d1 declared.>>"<<endl;
    Derived d1;

    cout<<"<<d2 declared.>>"<<endl;
    Derived d2(1);
}
```

```
<<d1 declared.>>
Base() called.
Derived() called.

<<d2 declared.>>
Base(int a) called.
Derived(int a) called.
```

Call the base class constructor that has one integer argument.

1. Initialize Base members: Initializer List

```
class Vehicle{
    int passenger;
    double baggage;
public:
    Vehicle(int person, double weight){
        passenger = person;
        baggage = weight;
    }
        . . .
};

class Airplane : public Vehicle{
    int crew_man;
public:
    Airplane(int person, double weight, int crew): Vehicle(person, weight) {
        crew_man = crew;
    }
        . . .
};
```

When this constructor is called,
we will first call
`Vehicle(int person, double weight)`.

```
int main(){
    Airplane a(120, 1300.0, 10);
    a.ShowData();
}
```

```
<<Airplane>>
passenger: 120
baggage: 1300
crew man: 10
```

1. Initialize Base members: Initializer List

```
class Airplane : public Vehicle{
    int crew_man;
public:
    Airplane(int p, double w, int c): Vehicle(p, w) {
    crew_man = c;
    }
        . . .
};
```

int x;
x = 10;

```
class Airplane : public Vehicle{
    int crew_man;
public:
    Airplane(int p, double w, int c): Vehicle(p, w), crew_man(c) {
    }
        . . .
};
```

int x = 10;

You can also use Initializer List for the data member

2. Initialize Base members: Protected Member

```
class Vehicle{
protected:
    int passenger;
    double baggage;
public:
    Vehicle(){}
       . . .
};
class Airplane : public Vehicle{
    int crew_man;
public:
    Airplane(int person, double weight, int crew){
        passenger = person;
        baggage = weight;
        crew_man = crew;
    }
       . . .
};
```

Access	public members	protected members	private members
Same Class	Y	Y	Y
Derived Class	Y	Y	N
Outside Class	Y	N	N

Protected members of a class A are not accessible outside of A's code, but is accessible from the code of any class derived from A.

Polymorphism

- A call to a member function will cause a different function to be executed depending on the type of the object that invokes the function.
- Function overriding allows to have the same function in derived class which is already defined in its base class.

```
class Vehicle{
    public:
    void ShowData(){cout<<"<<Vehicle>> "<<endl;}
};
class Airplane : public Vehicle{
    public:
    void ShowData(){cout<<"<<Airplane>> "<<endl;}
};
class Train : public Vehicle{
    public:
    void ShowData(){cout<<"<<Train>> "<<endl;}
};
```

```
int main(){
    Airplane a(100,300,20);
    a.ShowData();
}
```

```
<<Airplane>>
```

Airplane::ShowData() overrides Vehicle::ShowData().

Declared Type vs. Actual Type

```
int main(){
    Airplane a(100,300,20);
    Train t(50,100,30);

    a.ShowData();          <<Airplane>>
    t.ShowData();          <<Train>>

    Vehicle *ptr;
    ptr = &a;
    ptr->ShowData();       <<Vehicle>>

    ptr = &t;
    ptr->ShowData();       <<Vehicle>>

    //ptr->AddLength(10); Compile Error!
}
```

- Base class pointer (or reference) can point its derived class.
- However, the base class does not have access to its derived class members.

Virtual Function – Why?

```
class City{
    private:
        Vehicle *vlist[100];
        int index;
    public:
        City(){ index = 0;}
        void AddVehicle(Vehicle *v){
            vlist[index++] = v;
        }
        void ShowList(){
            for(int i=0;i<index;i++)
                vlist[i]->ShowData();
        }
};
```

```
int main(){
    City Champaign;

    Champaign.AddVehicle(new Airplane(30,100,5));
    Champaign.AddVehicle(new Train(100,300,10));
    Champaign.AddVehicle(new Train(130,300,15));

    Champaign.ShowList();
}
```

Virtual Function – Why?

```
class City{
    private:
        Vehicle *vlist[100];
        int index;
    public:
        City(){ index = 0;}
        void AddVehicle(Vehicle *v){
            vlist[index++] = v;
        }
        void ShowList(){
            for(int i=0;i<index;i++)
                vlist[i]->ShowData();
        }
};
```

```
int main(){
    City Champaign;

    Champaign.AddVehicle(new Airplane(30,100,5));
    Champaign.AddVehicle(new Train(100,300,10));
    Champaign.AddVehicle(new Train(130,300,15));

    Champaign.ShowList();
}
```

We want to print out the full information about Airplane or Train.

But, it will only print out Vehicle.

We want to manage *base class*, not *derived classes*.

→ Wish to resolve functions at run-time, a.k.a. dynamic binding.

Virtual Function

- Virtual functions are the member function in the base class that is expected to **be redefine in the derived class**.

```
class Vehicle{
    public:
    virtual void ShowData(){
        cout<<"<<Vehicle>> "<<endl;
    }
};
class Airplane : public Vehicle{
    public:
    void ShowData(){
        cout<<"<<Airplane>> "<<endl;
    }
};
class Train : public Vehicle{
    public:
    void ShowData(){
        cout<<"<<Train>> "<<endl;
    }
};
```

```
int main(){
    Airplane a(100,300,20);
    Train t(50,100,30);

    a.ShowData();          <<Airplane>>
    t.ShowData();          <<Train>>

    Vehicle *ptr;          static binding
    ptr = &a;
    ptr->ShowData();       <<Airplane>>

    ptr = &t;              dynamic binding
    ptr->ShowData();       <<Train>>
}
```

Abstraction – Pure Virtual Function & Abstract Class

- 'Vehicle' class will never be instantiated as it is. Instead, it will be either 'Airplane' or 'Train' object.
- Abstract class cannot be instantiated (pointer is fine) and implemented with one or more "pure" virtual function

```
class Vehicle{    <= abstract class (has a pure virtual function)
    public:
    virtual void ShowData() = 0;  <= pure virtual function (has no body)
};
class Airplane : public Vehicle{
    public:
    void ShowData(){
        cout<<"<<Airplane>> "<<endl;
    }
};
class Train : public Vehicle{
    public:
    void ShowData(){
        cout<<"<<Train>> "<<endl;
    }
};
```

```
int main(){
    Vehicle *vptr;        // this is ok
    Vehicle v(100,10);    // compile error
}
```

*Derived class must define a body for the pure virtual function, otherwise it will also be considered an abstract base class.

Constructor & Destructor

```
class Person{
    char name[20];
    int age;
public:
    Person(char const *_name, int _age){
        strcpy(name, _name);
        age = _age;
        cout<<"constructing name: "<<name<<endl;
    };
    ~Person(){
        cout<<"destroying name: "<<name<<endl;
    };
};

int main(){
    Person p1 = Person("Alice", 20);
    Person p2 = Person("Bob", 20);
}
```

```
constructing name: Alice
constructing name: Bob
destroying name: Bob
destroying name: Alice
```

Copy Constructor

- Initialize an object using another object (member-by-member).
- If a copy constructor is not provided by the user, it will be automatically inserted (default copy constructor)

```
class Point{
private:
    int x,y;
public:
    Point(int _x, int _y){x = _x; y = _y;}
    Point(const Point &p){
        x = p.x;
        y = p.y;
        //p.x = 0; // Don't want to allow this
    }
    void ShowData(){ cout<<"("<<x<<", "<<y<<")"<<endl;}
};
int main(){
    Point p1(10,20);
    Point p2(p1);

    p1.ShowData();
    p2.ShowData();
}
```

Use "const" to prevent modification on p

Shallow Copy

```
class Person{
private:
    char *name;
    int age;
public:
    Person(){};
    Person(const char *_name, int _age);
    void ShowData();
    ~Person();
};
Person::Person(const char *_name, int _age){
    name = new char[strlen(_name)+1];
    strcpy(name, _name);
    age = _age;
}
Person::~Person(){
    delete []name;
}
```

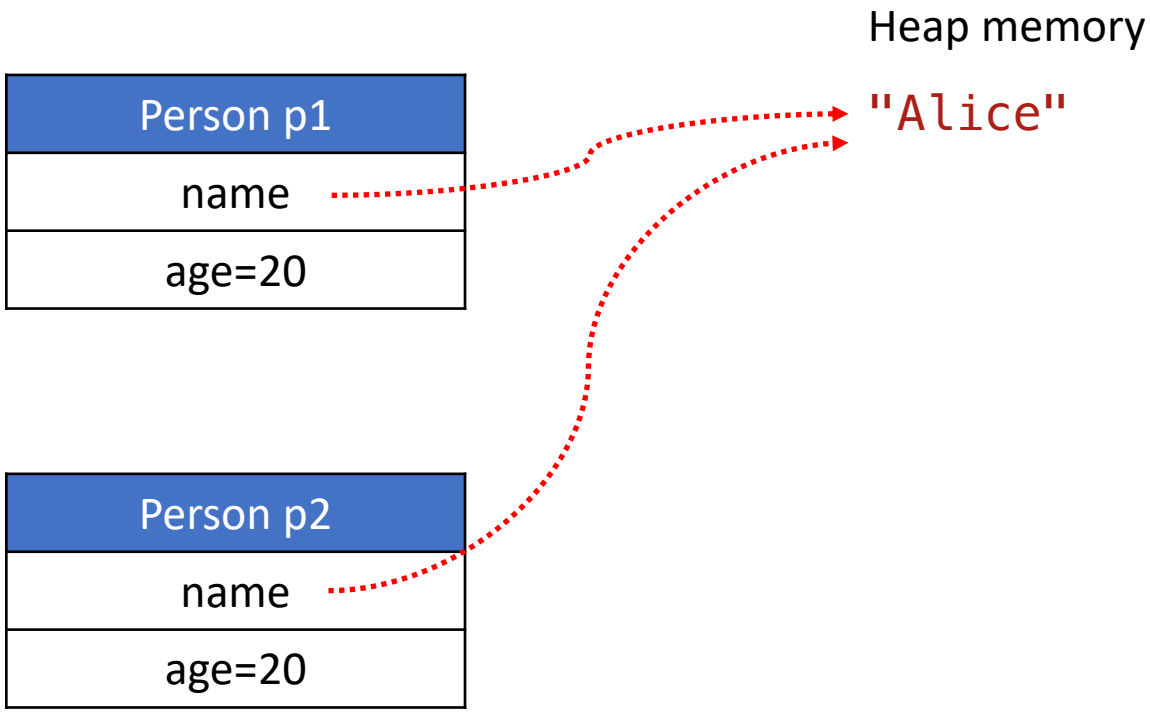
```
int main(){
    Person p1 = Person("Alice", 20);
    Person p2(p1);
    p1.ShowData();
    p2.ShowData();
}
```

Default copy constructor will be inserted.

```
Person::Person(const Person &p){
    name = p.name;
    age = p.age;
}
```

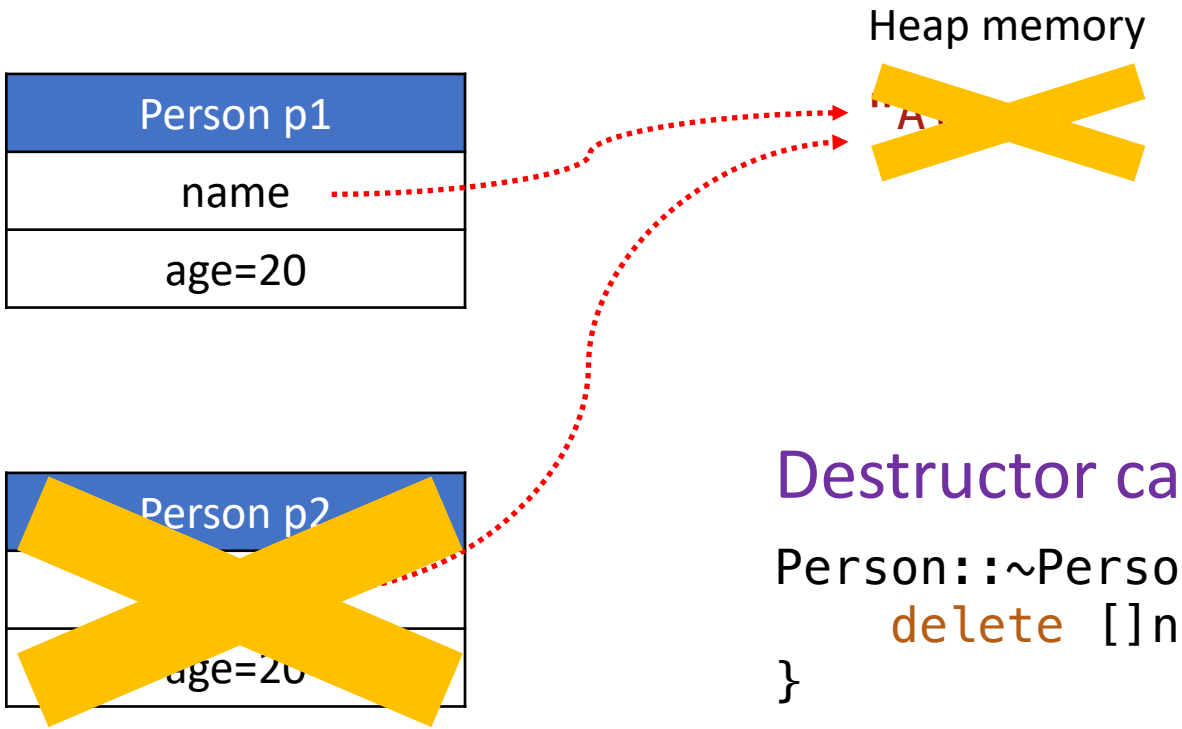
Run-time error!

Shallow Copy




```
Person::Person(const Person &p){
    name = p.name;
    age = p.age;
}
```

Shallow Copy



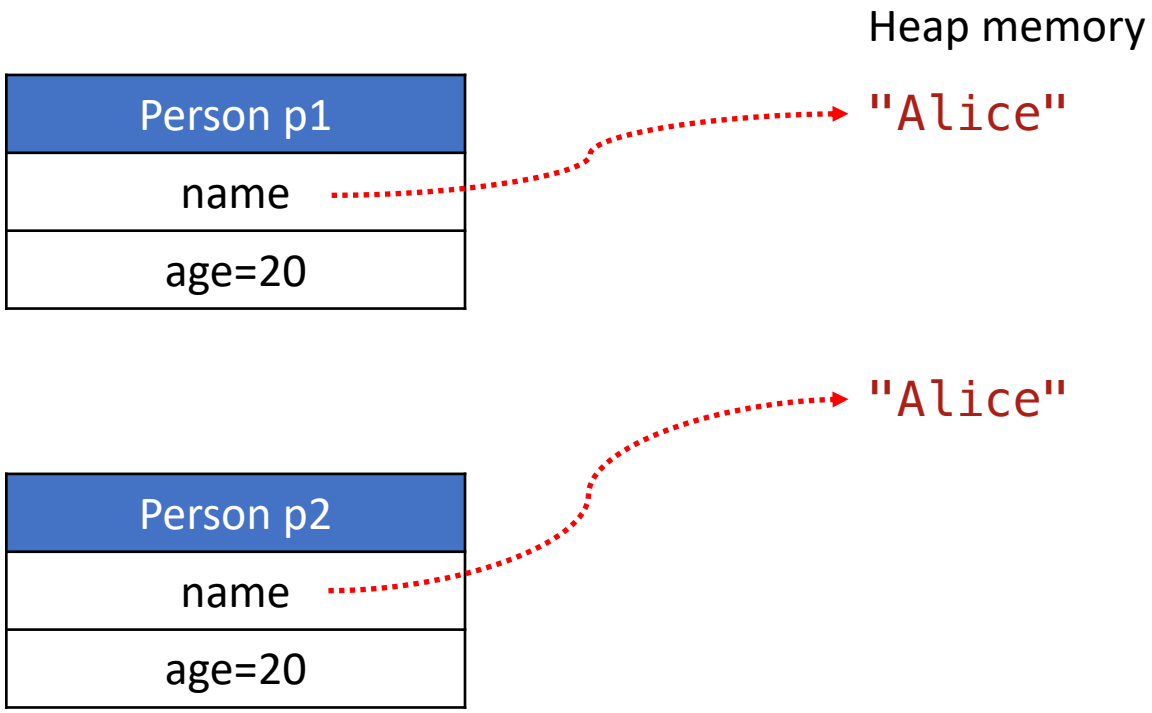
Destructor called!

```
Person::~Person(){
    delete []name;
}
```

When p1 calls its destructor,
the heap memory pointed by “name” is already deallocated.

🚫 double free!

Deep Copy




```
Person::Person(const Person &p){
    name = p.name;
    age = p.age;
}
```

```
Person::Person(const Person &p){
    name = new char[strlen(p.name)+1];
    strcpy(name, p.name);
    age = p.age;
}
```

`this` Pointer

- The this pointer holds the address of the current object.

```
class AAA{
    public:
        AAA *getAddress(){
            return this;
        }
};

int main(){
    AAA *a1 = new AAA();
    cout<<"pointer a1: "<<a1<<endl;
    cout<<"this of a1: "<<a1->getAddress()<<endl;
```

```
pointer a1: 0xddb010
this of a1: 0xddb010
```

Operator Overloading: Copy Assignment (=)

```
Point p1(1,2);

Point p2(p1);
Point p3 = p1;

p3 = p2;
```

→Call copy constructor
→Call copy constructor

→p3 is already initialized. Cannot call copy constructor.

p3.operator=(p2)

```
Point& operator=(const Point &p){
    x = p.x;
    y = p.y;
    return *this;
}
```

- Copy assignment is implicitly defined, if user did not provide it.

- The return value is a reference to `*this`.
- It allows "chained assignment".

```
p3 = (p2 = p1);   → p3 = p2;
```

p2.operator=(p1)

Default things added by compiler, if user doesn’t provide

- constructor
- destructor
- copy constructor
- copy assignment