Classes and Objects

Week 2_Part 2

Topics

1. The **this** Pointer and Constant
2. Member Functions
3. Static Members
4. Friends of Classes
5. Memberwise Assignment
6. Shallow Copy
7. Deep Copy
8. Copy Constructors

The `this` Pointer and Constant Member Functions

- **this** pointer:
  - Implicit parameter passed to every member function
  - it points to the object calling the function
- **const** member function: does not modify its calling object
- Can be used to access members that may be hidden by parameters with the same name

```
class SomeClass
{
private:
  int num;

public:
  void setNum ( int num )
  {
    this->num = num;
  }
};
```

```
class Student {
private:

   int sid;
   string name;
   float marks[5];
   float gpa;

public:
  Student () {
       sid = 0;
       name = "Default";
       for (int i=0;i<5;i++)
           marks[i] = 0;
       gpa = 0;
  }
```

```
Student(int sid, string nm, float marks[], float gpa)
  {
    this->sid = sid;
    name = nm;
    for (int i=0;i<5;i++)
      this->marks[i] = marks[i];
    this->gpa = gpa;
  }

void show ( ) {
       cout << "ID :" << this->sid  << endl
            << "Name :" << name <<  endl;
       cout << "Marks: ";
       for (int i=0;i<5;i++) cout << marks[i] << ":";
       cout << endl;
       cout << "GPA :" << gpa << endl;
    }
};
```

Can be used but unnecessary

```
int main() {
    float m[ ] {90.0f,80.0f,70.0f,90.0f,86.0f};
    string name = "Goh";
    Student st(10,name,m,3.6f);
    st.show();
    return 0;
}
```

```
ID :10
Name :Goh
Marks: 90:80:70:90:86:
GPA :3.6
```

Constant Member Functions

- Declared with keyword const
- When const appears in the parameter list,

const myPair &p

- the function is prevented from modifying the parameter.
- The parameter is read-only.
- When const follows the parameter list,

myPair getPair() const

- the function is prevented from modifying the object.

```
int main ( )
{
    myPair p(3,4);
    print(p);
    return 0;
}
```

```
3,4
```

```
void print (const myPair &p)
{
    cout << p.getPair().getX()
    << ","
    << p.getPair().getY()
    << endl;
}
```

```
class myPair{
private:
    int x;
    int y;
public:
    myPair(int x=0, int y = 0) {
        this->x = x;
        this->y = y;
    }
    void setPair(int x, int y){
        this->x = x;
        this->y = y;
    }
    myPair getPair() const {
        return myPair(x,y);
    }
    int getX(){ return x;}
    int getY(){ return y;}
};
```

Static Member Variables

- Static member variable:
  - One instance of the variable for the entire class
  - It is shared by all objects of the class

- Static member function:
  - Can be used to access static member variables
  - Can be called before any class objects are created

1) Must be declared in class with keyword **static**
2) Must be defined outside of the class
3) Can be accessed or modified by any object of the class: Modifications by one object are visible to all objects of the class

Counting the number of objects created

```
class myPair {
private:
    int x;
    int y;
public:
    static int NoOfObjects;
    myPair(int x=0, int y = 0) {
        this->x = x;
        this->y = y;
        NoOfObjects++;
    }
    ~myPair(){
        NoOfObjects--;
    }
    void setPair(int x, int y){
        this->x = x;
        this->y = y;
    }
    myPair getPair() const {
        return myPair(x,y);
    }
    int getX(){ return x;}
    int getY(){ return y;}
};
```

1

2

```
void print (const myPair &p){
    cout << p.getPair().getX()
    << ","
    << p.getPair().getY()
    << endl;
}
int myPair::NoOfObjects = 0;
int main(){
    myPair p1(3,4);
    cout << p1.NoOfObjects << endl;
    myPair p2;
    cout << myPair::NoOfObjects << endl;
    myPair pa[3];
    cout << pa[2].NoOfObjects << endl;
    return 0;
}
```

3

```
1
2
5
```

Static Member Functions

```
class myPair {
private:
   int x;
   int y;
   static int NoOfObjects;
public:
   myPair(int x=0, int y = 0) {
      this->x = x;
      this->y = y;
      NoOfObjects++;
   }
   ~myPair() { NoOfObjects--; }
   static int Counter() {
      return NoOfObjects;
   }
   void setPair(int x, int y) {
      this->x = x;
      this->y = y;
   }
   myPair getPair() const {
      return myPair(x,y);
   }
   int getX(){ return x;}
   int getY(){ return y;}
};
```

1) Declared with **static** before return type
2) Can be called independently of class objects, through the class name:
3) Because of item 2 above, the **this** pointer cannot be used
4) Can be called before any objects of the class have been created
5) Used primarily to manipulate static member variables of the class

```
int myPair::NoOfObjects = 0;

int main(){
   myPair p1(3,4);
   cout << p1.Counter() << endl;
   myPair p2;
   cout << myPair::Counter() << endl;
   myPair pa[3];
   cout << pa[2].Counter() << endl;
   return 0;
}
```

```
1
2
5
```

Stand-alone Function as a Friend of a Class

- Friend function: a function that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with the friend keyword in the function prototype

```
class myPair {
private:
    int x;
    int y;
public:
    friend void print (const myPair &p);
    myPair(int x=0, int y = 0){
        this->x = x;
        this->y = y;
    }
    void setPair(int x, int y){
        this->x = x;
        this->y = y;
    }
    int getX(){ return x;}
    int getY(){ return y;}
};
```

Must declare the print function as a friend

```
void print (const myPair &p)
{
    cout << p.x
    << ","
    << p.y
    << endl;
}
int main()
{
    myPair p[3] = {{1,3},{2,4},{3,5}};
    print(p[1]);
    return 0;
}
```

Print now can access the private members of p

2,4

A Function member of a class as a Friend of another Class

```
class myPair;       //forward declaration
```

```
class Printer {
public:
   void show(myPair &p);
};
```

1

```
class myPair {
private:
  int x;
  int y;
public:
  myPair(int x=0, int y = 0){
     this->x = x;
     this->y = y;
  }
  void setPair(int x, int y){
     this->x = x;
     this->y = y;
  }
  friend void Printer::show(myPair &p);
};
```

2

```
void Printer::show(myPair &p)
{
  cout << p.x << ",";
  cout << p.y << endl;
}
```

3

```
int main(){
   myPair p{1,3};
   Printer pr;
   pr.show(p);
   return 0;
}
```

1,3

- It can be done in the following order
  - myPair is forward declared first
  - Printer class specification is defined second
  - myPair class is declared third
  - The implementation of show that belongs to class Printer is defined after myPair class.

A class is a friend of another Class

- A friend class is a class whose members have access to the private or protected members of another class
- Printer class is a friend of myPair class, therefore, all member functions of Printer have unrestricted access to all members of myPair class, including the private members.
- In general, you should restrict the property of Friendship to only those functions that must have access to the private members of a class.

```
class Printer;
class myPair {
private:
    int x;
    int y;
public:
    friend class Printer;
    myPair(int x=0, int y = 0){
        this->x = x;
        this->y = y;
    }
};

class Printer {
public:
    void show(myPair &p){
        cout << p.x << ",";
        cout << p.y << endl;
    }
};
```

```
int main ( ) {
    myPair p{1,3};
    Printer pr;
    pr.show(p);
    return 0;
}
```

```
1,3
```

Two classes Share a Friend Function

- Class A and class B share a common friend function add

```
class B;
class A;
void add(A,B);

class A {
private:
    int a;
public:
    A(){
        a = 100;
    }
    friend void add(A,B);
};
```

```
class B{
private:
    int b;
public:
    B(){
        b = 100;
    }
    friend void add(A,B);
};

void add (A Aobj, B Bobj) {
    cout << (Aobj.a + Bobj.b)
         << endl;
}
```

```
int main ( ) {
    A A1;
    B B1;
    add(A1,B1);
    return 0;
}
```

```
200
```

Members Assignment

- Can use = to assign one object to another, or
- To initialize an object with an object's data

```
class myPair {
private:
    int x;
    int y;
public:
    myPair(int x=0, int y = 0) {
      this->x = x;
      this->y = y;
    }
    int getX(){ return x;}
    int getY(){ return y;}

    void show(){
        cout << x << "," << y << endl;
    }
};
```

```
int main()
{
    myPair a(10,20);
    a.show();

    myPair b = a;

    b.show();

    myPair c;
    c = a;

    c.show();

    return 0;
}
```

Initialization of b using a default Copy constructor

Assignment of object a to object c (member to member assignment)

Member to Member Assignment

- Can use = to assign one object to another, or to initialize an object with an object's data
- Copies member to member.
- Can be used at initialization

```
int main( ) {
   Circle c1 (10,20, 5);
   Circle c2;
   c2 = c1;
   cout << c2.Area() << endl
        << c1.Area() << endl;
   c1.setR(10);
   cout << c2.Area() << endl
        << c1.Area() << endl;
   Circle c3 = c1;
   cout << c3.Area() << endl
          << c1.Area() << endl;
   return 0;
}
```

```
class Circle {
private:
    int x, y;
    double radius;
public:
    Circle ();
    Circle
(int,int,double);
    Circle (int, int);
    void setX(int x);
    void setY(int y);
    void setR(double r);
    double getR() const;
    int getX() const;
    int getY() const;
    double Area ();
};
```

```
Circle::Circle():x(0),y(0),radius(0)
{}
Circle::Circle(int x, int y){
    this->x = x;    this->y = y;
    radius = 1.0;
}
Circle::Circle(int x, int y, double r){
    this->x = x;    this->y = y;
    radius = r;
}
void Circle::setX(int x){this->x = x; }
void Circle::setY(int y){this->y = y; }
void Circle::setR(double r){
   radius = r;
}
int Circle::getX()const{return x;}
int Circle::getY()const{return y;}
double Circle::getR()const{
    return radius;
}
double Circle::Area(){
    return 3.14159 * radius * radius;
}
```

```
78.5397
78.5397
78.5397
314.159
314.159
314.159
```

Copy constructors

```
class Car {
private:
    string *name;
    int    *engsize;
    int    passengers;
public:
    Car();
    Car(string s,int, int);
    void setName(string n);
    void setEngSize(int size);
    void setPass(int);
    void print();
};
```

```
int main(){
    Car c1;
    c1.print();
    Car c2(c1);
    c2.print();
    c2.setName("Alza");
    c2.setEngSize(1500);
    c2.setPass(7);
    c2.print();
    c1.print();
    return 0;
}
```

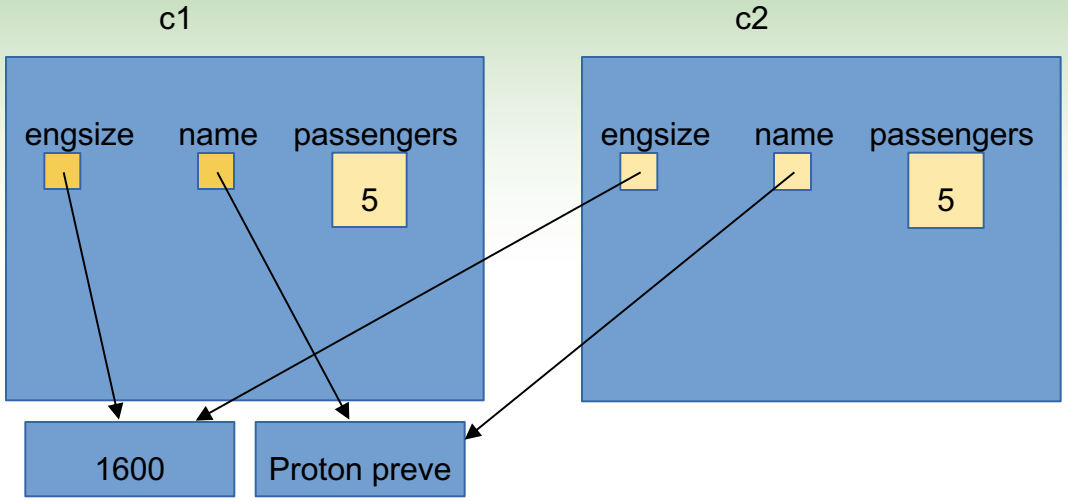
```
Car::Car() {
    name = new string("Proton Preve");
    engsize = new int(1600);
    passengers = 5;
}

Car::Car(string n,int sz, int pss) {
    name = new string(n);
    engsize = new int(sz);
    passengers = pss;
}

void Car::setName(string s) {
    *name = s;
}

void Car::setEngSize(int size){
    *engsize = size;
}

void Car::print (){
    cout << *name << endl
         <<  *engsize
         << endl << passengers
         << endl;
}
```

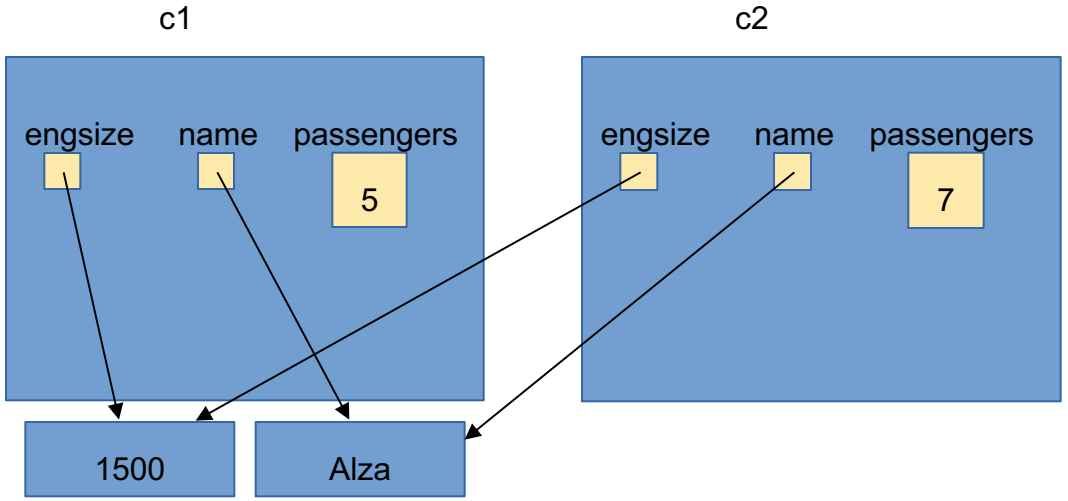
- Special constructor used when a newly created object is initialized to the data of another object of same class
- Default copy constructor copies field-to-field
- Default copy constructor works fine in many cases
- Problem: what if object contains a pointer?

```
Proton Preve
1600
5
Proton Preve
1600
5
Alza
1500
7
Alza
1500
5
```

Shallow Copy

```
int main(){
   Car c1;
   c1.print();
   Car c2(c1);
   c2.print();


   c2.setName("Alza");
   c2.setEngSize(1500);

   c2.setPass(7);
   c2.print();
   c1.print();

   return 0;
}
```






```
Proton Preve
1600
5
Proton Preve
1600
5
```

```
Alza
1500
7
Alza
1500
5
```

Deep Copy

```
class Car {
private:
    string *name;
    int    *engsize;
    int    passengers;
public:
    Car();
    Car (const Car &);
    Car(string s,int, int);
    void setName(string n);
    void setEngSize(int
size);
    void setPass(int);
    void print();
};
```

```
Car::Car(const Car &c) {
    name = new string;
    *name = *(c.name);

    engsize = new int;
    *engsize = *(c.engsize);

    passengers = c.passengers;
}
```

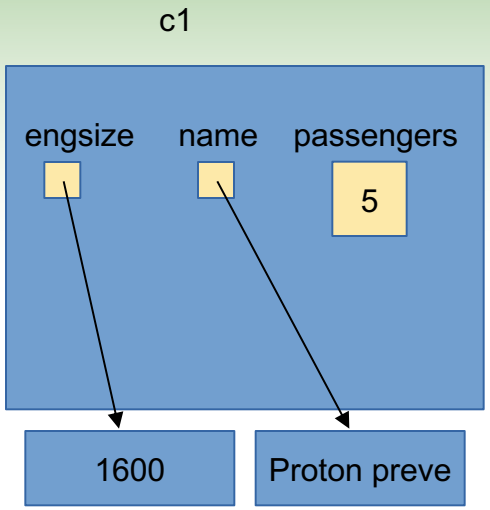
- Since copy constructor has a reference to the object it is copying from, it can modify that object.
- To prevent this from happening, make the object parameter const

```
int main() {
    Car c1;
    c1.print();

    Car c2(c1);
    c2.print();

    c2.setName("Alza");
    c2.setEngSize(1500);
    c2.setPass(7);

    c2.print();
    c1.print();

    return 0;
}
```

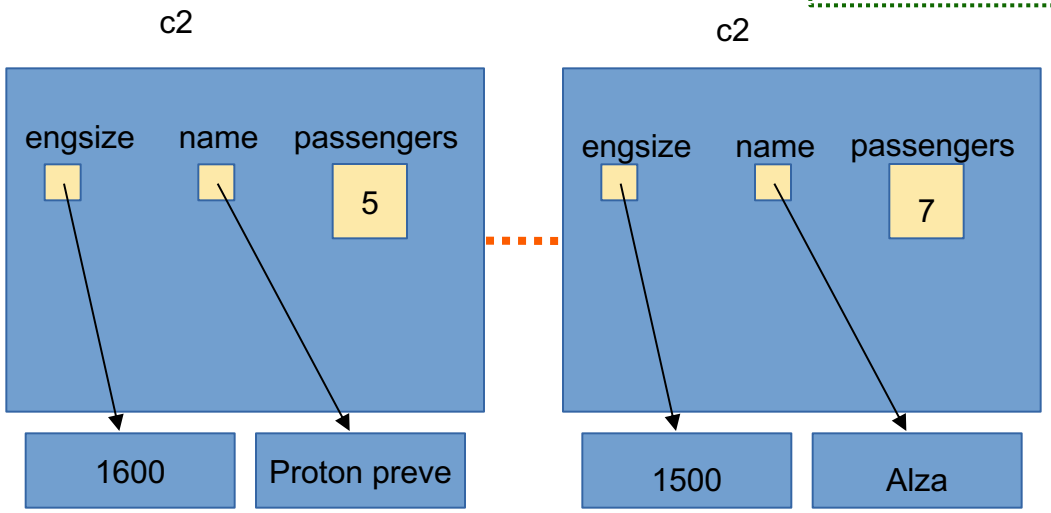
- Copy constructor is a constructor with a **const** reference to an object of the same type passed as a parameter.
- This copy constructor performs a deep copy to all the data members in the object, it creates all the dynamic data structures then copy the values from the referenced object.

Deep Copy

```
int main(){
    Car c1;
    c1.print();

    Car c2(c1);
    c2.print();

    c2.setName("Alza");
    c2.setEngSize(1500);
    c2.setPass(7);

    c2.print();
    c1.print();

    return 0;
}
```




```
Proton Preve
1600
5
Proton Preve
1600
5
Alza
1500
7
Proton Preve
1600
5
```



Copy Constructor

- Modification of memory by one object affects other objects sharing that memory
- A copy constructor is one that takes a reference parameter to another object of the same class
- The copy constructor uses the data in the object passed as parameter to initialize the object being created
- The reference parameter should be **`const`** to avoid potential for data corruption
- The copy constructor avoids problems caused by memory sharing
- It can allocate separate memory to hold new object's dynamic member data
- It can make the new object's pointer point to this memory
- It copies the data, not the pointer, from the original object to the new object
- A copy constructor is called when
  - An object is initialized from an object of the same class
  - An object is passed by value to a function
  - An object is returned using a **`return`** statement from a function

Copy Constructor

```
class myPair {
private:
   int *px;
   int *py;

public:
  myPair() {
     px = new int;
     py = new int;
  }

  void setPair(int x, int y){
     *px=x;     *py=y;
   }

   ~myPair() {
       delete px;    delete py;
    }

    void show() {
       cout << *px << ":"
            << *py << endl;
    }
};
```

```
int main()
{
    myPair p;
    myPair q(p);
    p.setPair(10,20);
    q.setPair(20,40);
    p.show();
    q.show();
    return 0;
}
```

```
20:40
20:40
week2(8607,0x1000c05c
0) malloc: *** error
for object
0x1005098b0: pointer
being freed was not
allocated
week2(8607,0x1000c05c
0) malloc: *** set a
breakpoint in
malloc_error_break to
debug
```

```
10:20
20:40
```

```
class myPair {
private:
   int *px;
   int *py;
public:
  myPair() {
     px = new int;
     py = new int;
  }
  myPair(const myPair& ot)
  {
       px = new int;
       py = new int;
       *px = *(ot.px);
       *py = *(ot.py);
   }
   void setPair(int x, int y){
     *px=x;     *py=y;
   }
   ~myPair() {
       delete px;    delete py;
    }
    void show() {
       cout << *px << ":"
            << *py << endl;
    }
};
```