# COMP 322: Introduction to C++

Chad Zammar, PhD
Mar. 17, 2023

# Lecture 9

## (Operator overloading)

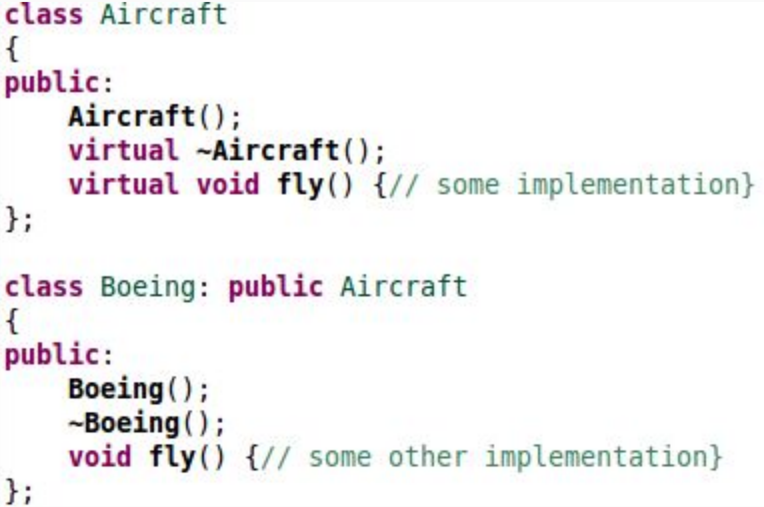
- Function overloading
- Class polymorphism
- Operators overloading

# Recap:  function overloading

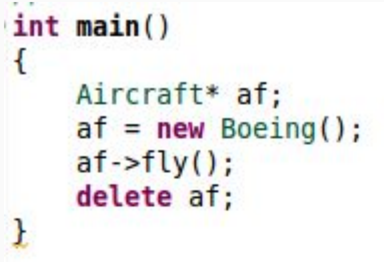
- Function name is the same but the function signature is different
- Multiple functions may have the same name but different number of arguments
  - int max(int i, int j);
  - int max(int i, int j, int k);
- Multiple functions may have the same name and same number of arguments but different types
  - int max(int i, int j);
  - float max(float i, float j);
- Compile time polymorphism

# Recap:  class polymorphism

- Different objects accessed by the same interface
- Run time polymorphism



```
class Aircraft
{
public:
    Aircraft();
    virtual ~Aircraft();
    virtual void fly() {// some implementation}
};

class Boeing: public Aircraft
{
public:
    Boeing();
    ~Boeing();
    void fly() {// some other implementation}
};
```

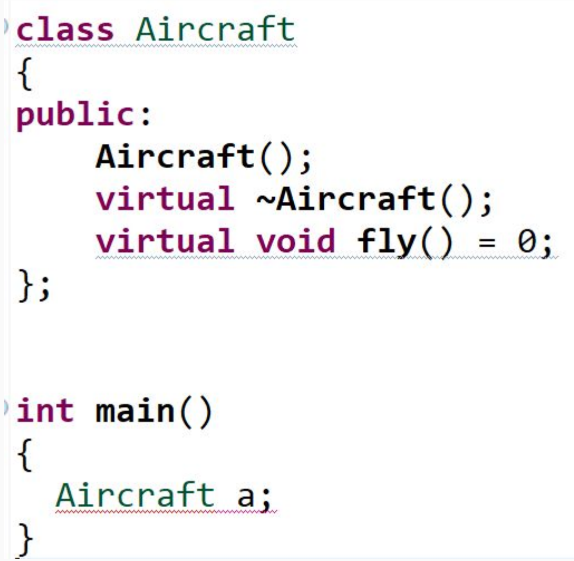


```
int main()
{
    Aircraft* af;
    af = new Boeing();
    af->fly();
    delete af;
}
```

# Virtual methods VS pure virtual methods

- Virtual method has an implementation in the base class and can be overridden by a derived class to obtain polymorphic behavior
- Pure virtual method does not have an implementation in the base class and should necessarily be implemented in the derived classes
  - virtual void fly() = 0;
- Class that does have at least one pure virtual method is called an abstract base class (similar to Java's interface classes)
- Abstract base classes cannot be instantiated. Only derived classes can

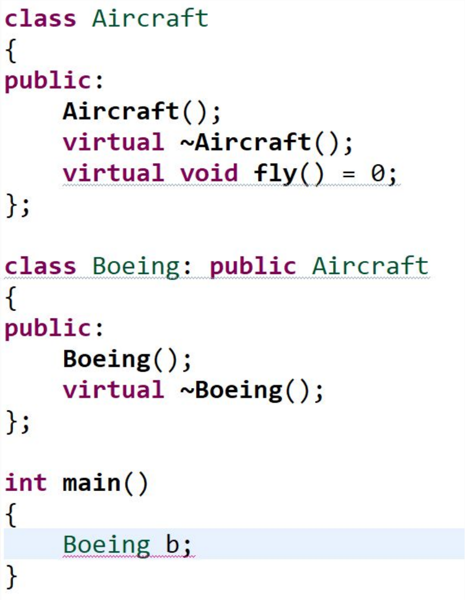
# Virtual methods VS pure virtual methods



```
class Aircraft
{
public:
    Aircraft();
    virtual ~Aircraft();
    virtual void fly() = 0;
};


int main()
{
  Aircraft a;
}
```

- Abstract base classes (ABC) cannot be instantiated
- If you try to instantiate an ABC you’ll get compilation error:
  - error: cannot declare variable 'a' to be of abstract type 'Aircraft'

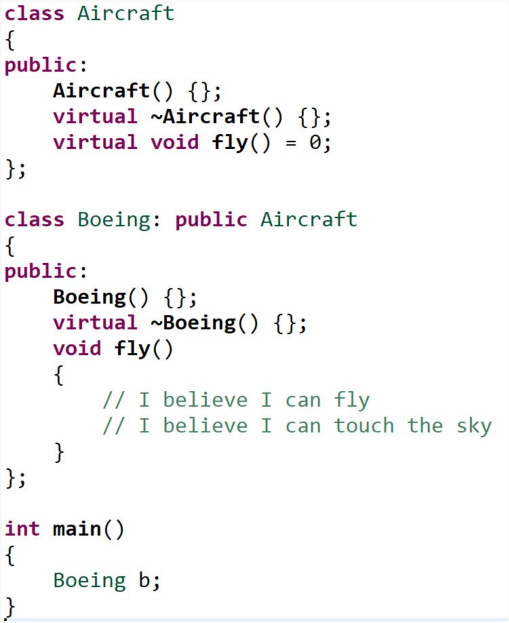
# Virtual methods VS pure virtual methods



```
class Aircraft
{
public:
    Aircraft();
    virtual ~Aircraft();
    virtual void fly() = 0;
};

class Boeing: public Aircraft
{
public:
    Boeing();
    virtual ~Boeing();
};

int main()
{
    Boeing b;
}
```

- Abstract base classes (ABC) cannot be instantiated
- If you try to instantiate a derived class you'll get compilation error:
  - error: cannot declare variable 'b' to be of abstract type 'Boeing'
  - We still don't have an implementation for the method fly()

# Virtual methods VS pure virtual methods



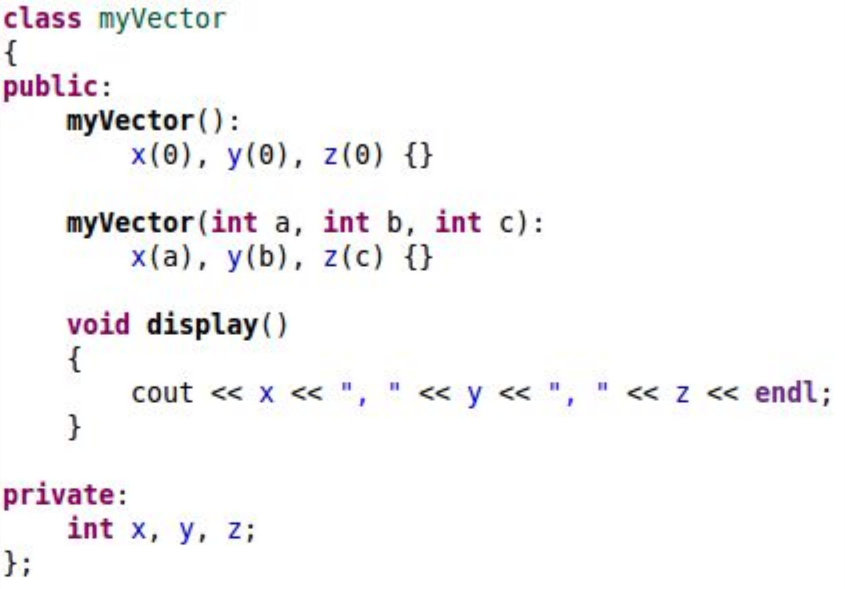
```
class Aircraft
{
public:
    Aircraft() {};
    virtual ~Aircraft() {};
    virtual void fly() = 0;
};

class Boeing: public Aircraft
{
public:
    Boeing() {};
    virtual ~Boeing() {};
    void fly()
    {
        // I believe I can fly
        // I believe I can touch the sky
    }
};

int main()
{
    Boeing b;
}
```

- Now we can define an object of type Boeing.

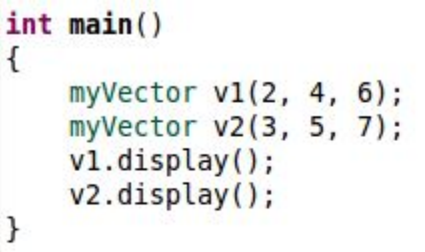
# Operator overloading

- The ability to reimplement (overload) most of the built in operators
  - The only non-overloadable operators are:
    - ::, ., ?: and .*
- Why overloading built in operators?
  - To be able to use them with user defined types
  - a + b is more natural than add(a, b)
- Java doesn’t provide operator overloading mechanism

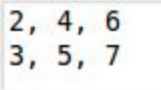
# Operator overloading: example



```
class myVector
{
public:
    myVector():
        x(0), y(0), z(0) {}

    myVector(int a, int b, int c):
        x(a), y(b), z(c) {}

    void display()
    {
        cout << x << ", " << y << ", " << z << endl;
    }

private:
    int x, y, z;
};
```

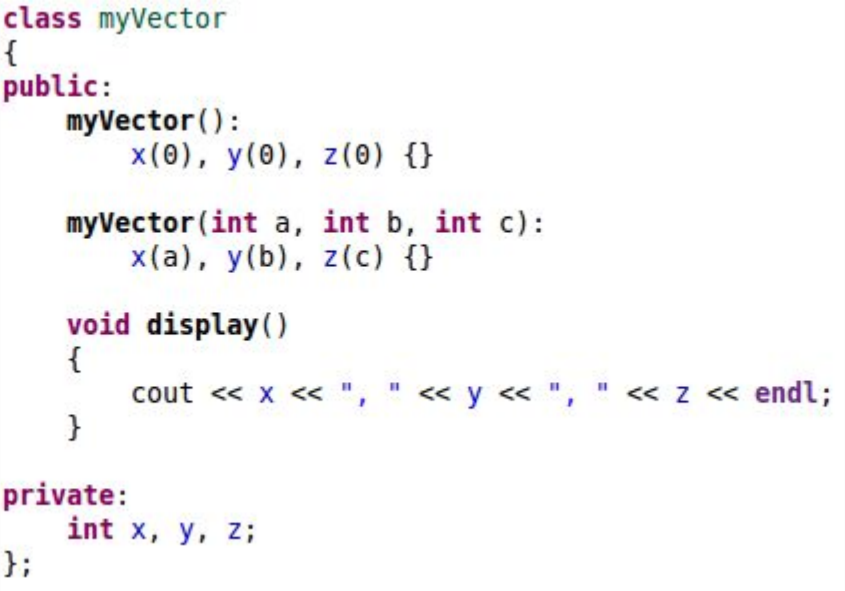


```
int main()
{
    myVector v1(2, 4, 6);
    myVector v2(3, 5, 7);
    v1.display();
    v2.display();
}
```

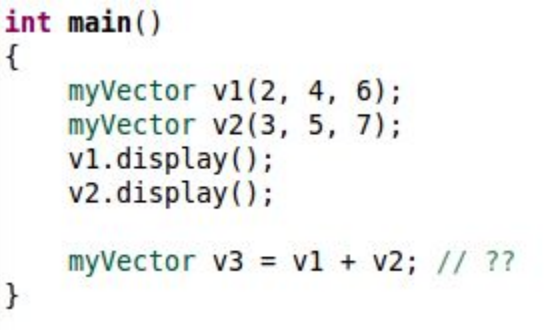


```
2, 4, 6
3, 5, 7
```

# Operator overloading: example
# How to add two vectors?

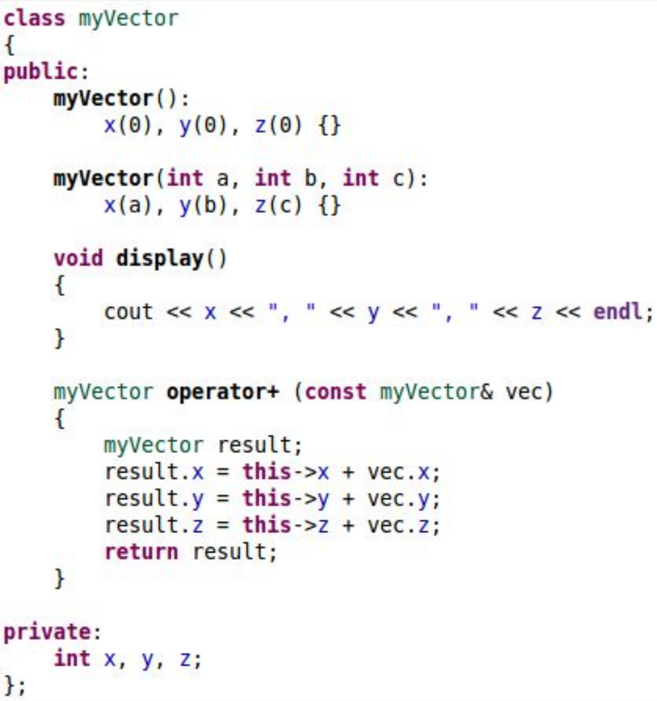


```cpp
class myVector
{
public:
    myVector():
        x(0), y(0), z(0) {}

    myVector(int a, int b, int c):
        x(a), y(b), z(c) {}

    void display()
    {
        cout << x << ", " << y << ", " << z << endl;
    }

private:
    int x, y, z;
};
```

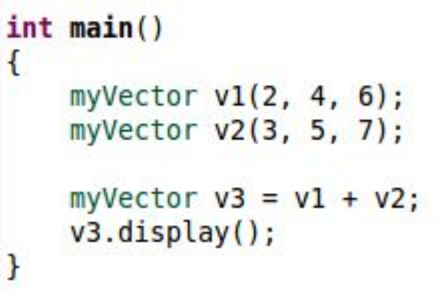


```cpp
int main()
{
    myVector v1(2, 4, 6);
    myVector v2(3, 5, 7);
    v1.display();
    v2.display();

    myVector v3 = v1 + v2; // ??
}
```

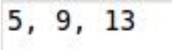
# Operator overloading: example
# How to add two vectors?



```
class myVector
{
public:
    myVector():
        x(0), y(0), z(0) {}

    myVector(int a, int b, int c):
        x(a), y(b), z(c) {}

    void display()
    {
        cout << x << ", " << y << ", " << z << endl;
    }

    myVector operator+ (const myVector& vec)
    {
        myVector result;
        result.x = this->x + vec.x;
        result.y = this->y + vec.y;
        result.z = this->z + vec.z;
        return result;
    }

private:
    int x, y, z;
};
```

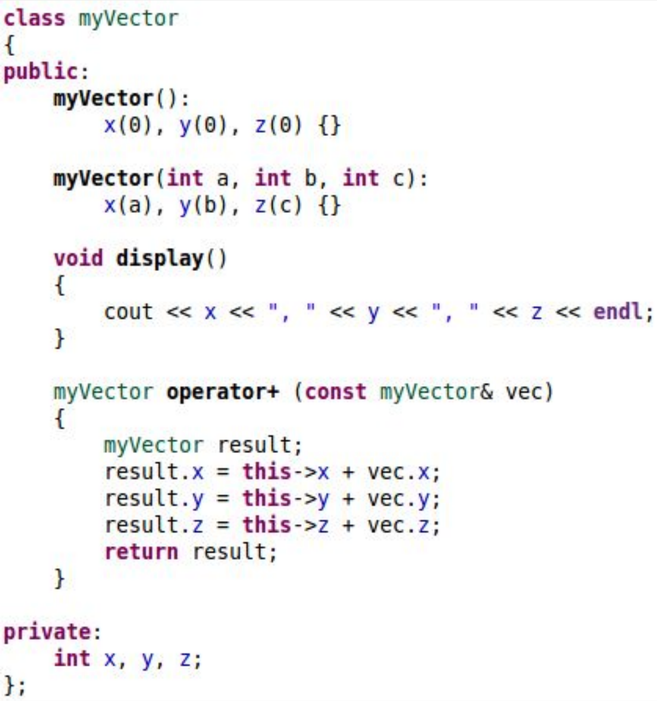


```
int main()
{
    myVector v1(2, 4, 6);
    myVector v2(3, 5, 7);

    myVector v3 = v1 + v2;
    v3.display();
}
```



```
5, 9, 13
```

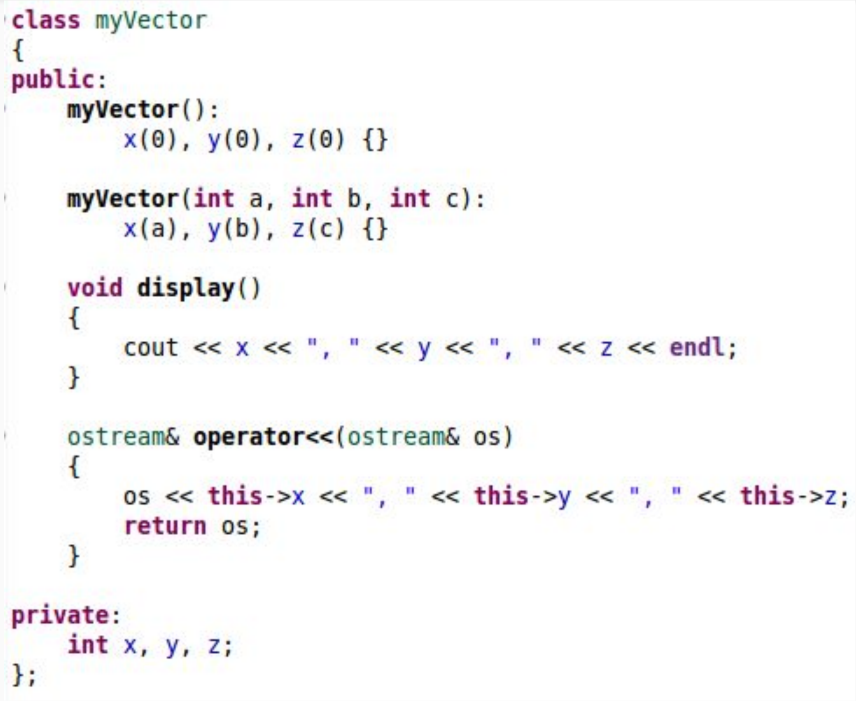
# Operator overloading: example
# Overloading << operator



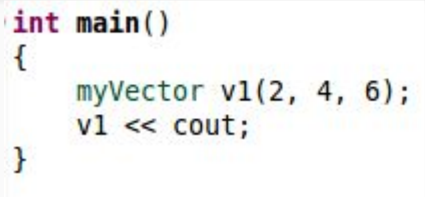
```
class myVector
{
public:
    myVector():
        x(0), y(0), z(0) {}

    myVector(int a, int b, int c):
        x(a), y(b), z(c) {}

    void display()
    {
        cout << x << ", " << y << ", " << z << endl;
    }

    myVector operator+ (const myVector& vec)
    {
        myVector result;
        result.x = this->x + vec.x;
        result.y = this->y + vec.y;
        result.z = this->z + vec.z;
        return result;
    }

private:
    int x, y, z;
};
```

- Let's replace display() method by overloading the ostream operator <<
- To be able to use cout to display a vector like any other built-in type
- Two options:
  - Member method
  - Friend method

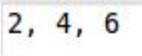
# Operator overloading: example
# Overloading << operator (member method)



```cpp
class myVector
{
public:
    myVector():
        x(0), y(0), z(0) {}

    myVector(int a, int b, int c):
        x(a), y(b), z(c) {}

    void display()
    {
        cout << x << ", " << y << ", " << z << endl;
    }

    ostream& operator<<(ostream& os)
    {
        os << this->x << ", " << this->y << ", " << this->z;
        return os;
    }

private:
    int x, y, z;
};
```



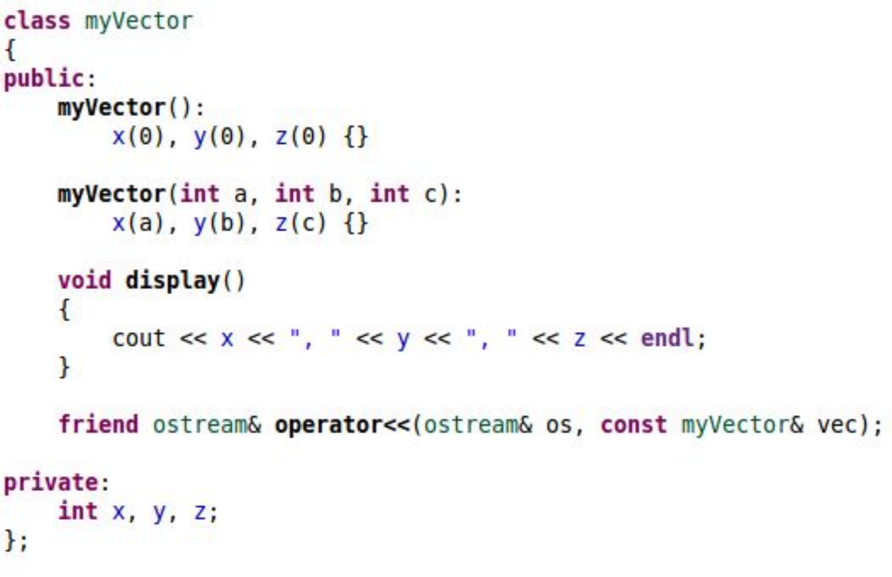
```cpp
int main()
{
    myVector v1(2, 4, 6);
    v1 << cout;
}
```



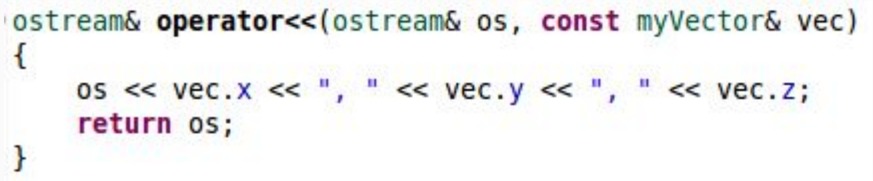
```
2, 4, 6
```

- Notice that the syntax is a bit confusing
  - v1 << cout instead of cout << v1
- Stream operator << is being called on v1 object and not on cout object
- It is recommended to use a friend method to avoid this confusion

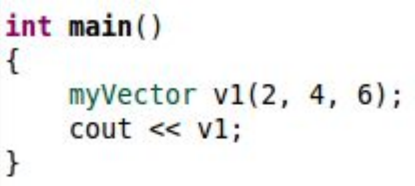
# Operator overloading: example
# Overloading << operator (friend method)



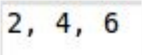
```cpp
class myVector
{
public:
    myVector():
        x(0), y(0), z(0) {}

    myVector(int a, int b, int c):
        x(a), y(b), z(c) {}

    void display()
    {
        cout << x << ", " << y << ", " << z << endl;
    }

    friend ostream& operator<<(ostream& os, const myVector& vec);

private:
    int x, y, z;
};
```



```cpp
ostream& operator<<(ostream& os, const myVector& vec)
{
    os << vec.x << ", " << vec.y << ", " << vec.z;
    return os;
}
```



```cpp
int main()
{
    myVector v1(2, 4, 6);
    cout << v1;
}
```



```
2, 4, 6
```