

1 Friends

- A friend function of a class is defined out of the class' scope, but the function has direct access to all of the private fields of the class.
- We use the “friend” keyword to do so.
- Consider the following code:

```
class Vec {
    int x, y;
public:
    friend ostream &operator<<(ostream &out, const Vec &v);
};

ostream &operator<<(ostream &out, const Vec &v) {
    return out << v.x << " " <<< v.y << endl;
}
```

- We can use this on a class as well, to allow a class to be able to access all fields of another class.
- For example, putting “friend class ClassTwo” in ClassOne will allow ClassTwo to be able to directly access all fields in ClassOne.
- Note that friends are not mutual, and friendship is not inherited by children. That is, if ClassThree is a child of ClassOne, ClassTwo cannot access ClassThree despite being able to access ClassOne.

2 Unified Modelling Language (UML)

- A way to visualize the design of a system.
- We can use this for C++, Java, etc.
- An example would be for a Vector class. A class in UML consists of the name, fields, then methods.
- Use “+” to symbolize a public field, and “-” to symbolize a private field.
- What we can use UML for is to show the RELATIONSHIP between classes and the like.
- For example, we can create a “Basis” class that takes in two vectors.
- Side note, we need to create ctors using MIL(member initialization base) for Basis as our Vec class doesn't cover a zero argument default ctor.
- Remember, when we initialize an object, the following run in order:

- Alloc memory
- Initialize/construct fields
- Run ctor body

- Back on topic:

- We can see the problem:

```
class Vec {
    int x, y;
public:
    Vec (int x, int y): x{x}, y{y} {}
};


class Basis {
    Vec v1, v2;
    // Right now, we don't have default ctors for the Vec class!
    // Instead, we'll make a new Basis ctor.
    // We CANNOT initialize the fields in the ctor body, so we MUST use MIL.
    Basis():
        v1{0, 0},
        v2{0, 0}
    {}
};

Basis b; // This is now valid.
```


- Basically, it's kinda like the charts you get if you look at class hierarchy on Netbeans.

3 Composition

- Our “Basis” class depends and uses a Vector class. It is composed with it — it relies on another class to exist.
- Note the opposite is not true, as we can have Vectors without ever declaring a Basis.
- Note that the Vector has no identity outside of A. That is, we cannot access the v1 or v2 fields without going through Basis first. Destroying a Basis object destroys the Vectors that are associated with it.
- This extends to copying — if a Basis is copied, then the Vectors are also copied.
- For composition, we show this with a class, a black diamond, and linking to the class it is composed of.

Basis  [Vec] represents Basis owning two Vectors.

4 Aggregation

- We represent this with A  B (a white triangle).
- It means that B exists within A, but apart from it. That is, if A is destroyed, B still exists.
- If A is copied, B is not copied.
- Essentially, B does not know about the existence of A. Furthermore, B could belong to multiple A instances.
- We do so by making B out of scope of A, and using pointers to B from A. We also have to make sure that in our destructor for A, we do not destroy B (a shallow destructor).
- A simple example of aggregation:

```
class Teacher {
    string name;
public:
    Teacher (string name): name{name}{}
    string getName() {return Name;}
};

class Department {
    Teacher *loT[100];
    int tNum;


public:
    Department(): tNum{0}{}
    void addTeacher(Teacher *teacher) {
        loT[tNum] = teacher;
        ++tNum;
    }
};

int main() {
    Teacher *teacher1 = new Teacher("Bob");
    Teacher *teacher2 = new Teacher("Jerry");
    {
        Department d;
        d.addTeacher(teacher1);
        d.addTeacher(teacher2);
        //d does not exist after this line/we exit scope!
    }
    cout << teacher1.getName() << "and " << teacher2.getName() << " are alive" << endl;
    // If we end here, we would get a memory leak! A does not destroy B, Dep
```

```

        // If you're confused, note that Department d does not use heap, so we can
        delete teacher1;
        delete teacher2;
    }

```

- If we destroy Department, then the Teacher still exists. Note the default destructor is a shallow destructor — it WILL NOT DESTROY TEACHER INSTANCES.
- Same with the default copy ctor — it is a shallow copy!
- In UML, we represent this with [Department]  [Teacher]

5 Inheritance

- This is basically a class which is the parent class with changes. This is also known as specialization/generalization.
- Specialization is inheriting down. That is, a Dog is a more specialized Animal. It is more specific.
- Generalization is going up. An Animal is a very general way to describe a Dog.
- Basically, all Dogs are Animals, but not all Animals are Dogs.
- We use a simple arrow pointing from the specialized children/subtype to the parent/supertype.
- Why do we need inheritance?
- Well, if some classes are very similar in fields to another class, where there are only some changes and are nearly identical, we can use a parent-child relationship to resolve this.
- We can also use a union.
- Union is similar to struct. The size of union is the size of its largest field, and it holds at most one of its field. Unions do not have a destructor.

- For example:

```

union BookTypes{ Book *b, Text *t, Comic *c };
BookTypes myBook[20]; // Every item is a pointer to book, text, or comic, but

```

- We can make a superclass by using the “union” keyword in C++:

```

union book {
    Text t;
    Comic c;
}

```

- But there's a better way to do inheritance in C++, without union or void pointers.
- As a refresher from Java:
 - Composition is “owns a”. For example, a Room class depends on a House class to exist. Remove the House, and there isn't any Rooms.
 - Aggregation is “has a”. For example, a Children class can exist without a Class class.
 - Inheritance is “is a”. For example, a Dog has the features of a Animal class.

```
class Parent {
    public:
        int id_p;
};

class Child : public Parent {
    public:
        int id_c;
        //id_p is inherited.
};
```

- Another example from class:

```
class Book {
    // string title , author;
    // int numPages;
    // If we do not make them protected over protected , then we will not be a

    protected string title , author;
    protected int numPages;

    public:
        Book(title , author , numPages) {}
};

class Text : public Book {
    string topic;

    public:
        Text(title , author , numPages , topic):
            Book{title ,author ,numPages}, //USE MIL
            topic{topic}
        {}
};

class Comic: public Book {
```

```

        string hero;

    public:
        Comic() {}
};

```

- Note that when the object is constructed, now:
 - Space is allocated.
 - The superclass fields are constructed.
 - Fields are constructed.
 - The ctor body runs.
- Thus, when we initialize a subclass, it constructs the superclass fields AND THEN constructs its additional fields.
- Every subclass inherits everything from the superclass, including fields and methods, if they are public or protected! If they are private, it will not be accessible.
- Thus, we have two problems if we didn't do the above: we wouldn't be able to access private fields/members, and we need the superclass to be constructed before the subclass.
- If the superclass has no default ctor, the subclass must have a non-default superclass ctor call in the MIL.
- Public means that anyone can access field.
- Private means only the class itself can access the field.
- Recall protected means we can access it directly within the class or through inheritance. For example, a protected field in Book can be accessed by Comic and Text, but a protected field in Text cannot be accessed by Comic or Book.
- What happens to private or protected fields that we inherit from a parent?
- If we try:

```

class Parent {
    public:
        int a;
    private:
        int b;
    protected:
        int c;
};

class Child : public Parent {
    //Can access a publicly , c protected , b is private and inaccessible

```

```

};
class Child : protected Parent {
    //Can access a and c protected , b is private and inaccessible .
};
class Child : private Parent {
    //Cannot access anything; all private .
};

```

- We can use accessors and setters, and make them protected, to only allow subclasses to access them, but still protect private fields.
- “Book b = Comic ...” will create a Book object.
- When we initialize a superclass with a subclass, we “slice” — we cut off the extra fields that are not used by the superclass.

- Consider the following:

```

Comic c {” ”, ” ”, 40, ” ”};
Book *pb = &c;
Comic *pc = &c;
pc -> isItHeavy ();
pb -> isItHeavy ();

```

- When we access an object through pointers, there is no slicing (no need and will not occur). But pc-> will return the Comic value, and pb-> will return the Book value.
- The overriding depends on the type of the pointer, not what the pointer is pointing at!
- Our problem is that the same object behaves differently, depending on the type of the pointer that points at it, even though slicing does not happen.

```

class Book {
    //...
    virtual bool isItHeavy() const {...} //add virtual
};

class Text: public Book {
    //...
    bool isItHeavy() const override {...} //add override
}

```

- This will force the earlier example work logically — that is, a Book pointing to a Comic will still run a Comic’s override.
- Remember to use the virtual and override keywords when overriding functions!

- This thus fixes our earlier problem.

- The above fix will resolve the issues of:

```
Comic c { " ", " ", 40, " " };
Book *pb { &c };
Book &rb { c };
Comic *pc { &c };
pc -> isItHeavy (); // true
b.isItHeavy (); // true
pb->isItHeavy (); // true
```

- Note if you have a method in a subclass, you can't invoke it from a superclass.
- Now, what we can do is make an array of Book pointers, and by using virtual, we can access the correct methods. And remember that pointers do not slice. Thus, an array of Book pointers can easily handle all Books and its subclasses. Or IOW, an array of the parent class will work as an array of it and its subclasses.
- This trait — being able to deal with multiple forms of a superclass and its children — is known as polymorphism. That is, we can abstract to accomodate everything.
- For example, istream is a superclass for ifstream, istream, etc. Based on the parameter (which is a reference), it will call the appropriate override.
- See C++/inheritance/*
- If we want to use polymorphism, do NOT use array of objects. Always use an array of pointers to said object(s).
- It is a good idea to always make the destructor of a superclass virtual, and override the dtor of subclasses, even if the dtor doesn't do anything.
- Even if we want to use the default dtor, we should instead create an empty dtor body and add the virtual keyword.
- Add the “final” keyword to a class declaration to specify that this subclass will not have children:

```
class y final: public X {
};
```
- What if we want to forbid the user from creating a superclass? For example, maybe the superclass is always either one subclass or another — and we don't want the user to be able to make said superclass.
- For example:


```

class Student {
    protected:
        int numCourses;
    public:
        //We state that this function has no implementation in this class.
        //This is called a pure virtual method. It allows the compiler
        //to not complain about a lack of a definition.
        //A class with a pure virtual method cannot be instantiated.
        //This is called an abstract class (think Java).
        virtual int fees() const = 0;
};

class Regular: public Student {
    public:
        int fees() const override;
};

class Coop: public Student {
    public:
        int fees() const override;
};

```

- As mentioned, a class with a pure virtual method is an abstract class.
- Its purpose is to organize the subclasses. For example, we group Regular and Coop students together as “Students”, but we don’t want the user to call the Student superclass.
- Subclasses are also abstract unless they implement all the pure virtual methods.
- We call classes that can be instantiated (that is, they have no pure virtual methods) as “concrete” classes.
- In UML, all abstract classes are in italic, and all virtual/pure virtual methods are in italics.
- See c++/purevirt

6 Inheritance — Copy/Move Operators

- See c++/inheritance
- What happens if we make the subclass with no copy/move ctors/assign, but the superclass has them?
- For example:

```

class Book {
public:
    //Assume copy/move operations are defined.
};

class Text : public Book {
    string topic;
public:

    //copy ctor
    Text::Text(const Text &other) :
        Book{other}, topic{other.topic}
        //We slice other to fit Book.
    {}

    //copy assignment
    Text& Text::operator=(const Text &other) {
        Book::operator=(other);
        topic = other.topic;
        return *this;
    }

    //move ctor
    Text::Text(Text &&other) :
        Book{std::move(other)}, topic{std::move(other.topic)}
    {}

    //move assignment
    Text & Text::operator=(Text &&other) {
        Book::operator=(other);
        topic = other.topic;
        return *this;
    }
};

```

- Let's say we called:

```

Text t1 {...}, t2 {...};
Book *pb1 = &t1, *pb2 = &t2;
*pb1 = *pb2;
//This will call the Book CAssOp. We don't have a virtual!
//We call this a partial assignment – the compiler sees this as a Book due
//to pointers, so it only partially assigns the value.

```

- But we realize that this doesn't work for the operator overrides — we return *this, but *this is not the same in the Book and Text operators!

- Luckily, C++ is smart, and deals with this for us, so we can continue to use virtual/override as per usual.

- Consider:

```
Text t { ... };
Book b { ... };
Text *pt = &t;
Book *pb = &b;
*pt = *pb; // This invokes the subclass now, not the superclass.
```

- Note that you can't override a virtual function with different parameters. We don't want to mix sub/superclasses.
- This is why we have problems of partial and mixed assignment.
- We solve by making an abstract superclass, instead of using, say, a Book -> subclass hierarchy, use a AbstractBook -> Book and subclass hierarchy.
- We don't write any of the Big 5 in the abstract superclass — we avoid the override problem by straight up removing the Big 5!
- Basically, always make your superclass abstract. Easiest way is to make the dtor a pure virtual method.
- Don't forget to implement a destructor if we do this, even if we don't need to implement a dtor that does deep copy destroy, as it has to be implemented to not be abstract.
- To avoid assigning one subclass to another, add the protected keyword to the abstract assignment override.