CS 246 - Object Oriented Programming **Fall 2016**

Lecture 7: October 20, 2016

Lecturer: Brad Lushman *Notes By: Harsh Mistry*

7.1 Abstraction of make files

7.1.1 Variables

We can abstract make files by introducing variables. These variables can be used to define common values such as compiler name and execution flags.

- CXX = g++-5
- CXXFLAGS = -std=c++14 -Wall

Variables must be their own line and to use them you must reference them within `${ ... }`

7.1.2 Shortcut

In addition, to variables, we can abstract the make file further by excluding the build line for any rule that follows the pattern : `x.o : x.cc a.h b.h .... .` So, `${CXX} ${CXXFLAG} -c -o x.0` does not have to be included.

7.1.3 Tracking dependencies

One of the biggest problems with writing make files is tracking dependencies and maintain them if they change. To achieve this, we can utilize the compilers ability to determine the dependencies by including the `-MMD` flag. `g++14 --MMD -c inter.cc` will then generate a compiled object file and a .d file which contains the dependencies needed by the make file. In principle, we now only have to include the .d files within the makeFile

Example 7.1 *makeFile with dependency tracking*

```
1 CXX = g++-5
2 CXXFLAGS -std=c++14 -Wall -MMD
3 OBJECTS = main.o list.o inter.o node.o
4 DEPENDS = ${OBJECTS: .o = .d}
5 Exec = myprogam
6 ${EXEC} : ${OBJECTS}
7     ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}
8 - include ${DEPENDS}
```

With dependency tracking, we only have to add the .o (object files) to the makeFIle as the project expands.

7.2 Software Engineering : System Modelling

Developing Object Oriented Systems involves identifying abstractions and analysing the related steps among them. So, it is helpful outline the implementation and design using a diagram. A popular standard for this is referred to as UML - (Unified Modelling Language)

Modelling a Class :

Name	-Vec
Fields	-x : Integer
	- y : Integer
Methods (Optional)	+ getX : Integer
	+getY : Integer

"+" Indicates Public and "-" indicates private

7.2.1 Relationships

Composition of Classes :

```
1  Class Vec {
2          int x, y;
3      public :
4          Vec (int x, int y);
5  };
6  Class Basis {
7          Vec v1, v2;
8      public :
9          Basis() : v1 {1, 0}, v2{0, 1} {}
10 };
```

Embedding one object (e.g Vec) inside another (Basis) is called **composition**. The Relationship between Basis and Vec is called "owns-a", A basis object "owns" a Vec object.

If A "owns a" B, then typically :

- B has no identity outside A (no independent existence)
- If A is destroyed, B is destroyed
- If A is copied, B is copied (Deep copies)

Example 7.2 *Physical Example of Ownership*
A car owns 4 wheels and a wheel is part of a car

- *If you destroy the car, you destroy the wheels*
- *If you copy the car, you copy the wheels*

To illustrate this, we draw a arrow from one class to another with a filled diamond at the end.

$$[Basis] \blacklozenge \longrightarrow_2 [Vec]$$

Aggregation :

If you compare car parts in a var ("owns a") to car parts in a catalogue.

The catalogue contains the parts, but the parts are an independent existence. This is a "has -a" relationship

If A "has a" B, then typically

- B has an existence, apart from its association with A
- If A is destroyed, B lives on
- If A is copied, B is not (Shallow Copy)
  - Copies of A share the same B

To illustrate this, we draw a arrow from one class to another with a empty diamond at the end.

$$[Pond] \diamond \longrightarrow [Ducks]$$

Specialization/Generalization (Inheritance) :

Suppose you want to track your collection of books.

```
Class Book{
        string title, author;
        int numPages;
    public :
        Book ( ---- ) ;
        ...
};

//For textbooks, also want to know the topic
Class Text {
        string title, author;
        int numPages;
        string topic;
    public :
        Text ( --- );
        ...
};

//For Comic Books with name of hero
Class Comic {
        string title, author;
        int numPages;
        String hero;
    public :
        comic ( --- ) ;
        ...
};
```

Despite being separate classes, there is a relationship. They all represent books. Now, if we wanted to represent all of the types in one array we can :

1. One solution is to use `union`, but this is not recommended as this type from C does not provide a predicate for checking what each value points to

```
union BookType {Book *b, Text *t, Comic *c};
Booktype myBooks[20]; //Declares array of 20 items
```

2. An alternate solution is to use void pointers to point to different classes, but this also requires us to know the type each pointer is pointing to.

The highlighted solutions from C are very inconvenient and not ideal. So to improve the implementation, we can use C++ inheritance, which will allow us to explicitly tell the compiler that a text/comic is a type of book

```
//Base Class (Superclass)
Class Book {
        string title, author;
        int numPages
    public
        Book ( --- );
        ...
};

//Derived Classes (Subclasses)
Class Text: public Book {
        string topic;
    public :
        Text ( --- );
        ...
};

Class Comic : public Book {
        string hero;
    public :
        Comic ( --- ) ;
          ..
};
```

The derived classes inherit fields from the base class, so in the provided example, title, author, and numpages are inherited. Given this, any function that can be called on book, can be called on Text and Comic.

Despite being apart of Book as a derived classes, subclasses can not access fields that it inherits, so requests must be made using the public functions provided by book.

What Happens when a object is constructed with derived classes

1. Space is allocated
2. Superclass part is constructed (New)
3. Fields are constructed
4. Constructor Body Runs

Now that we have derived classes, initialization is now a concern. Since derived classes are sub classes of a superclass, standard initializations do not work because

1. The derived classes do not have access to the inherited variables
2. Book does not have a default constructor

To resolve this issue, we can invoke Book's constructor in Text's MIL.

```
Class Text:public Book {
        ..
   public :
        Text (string title, string author, int numpages, string topic) :
            Book{title, author, numPages}, topic{topic} {}
};
```

In general, if superclass has no default constructor, the subclass must call a superclass constructor in the derived class MIL.

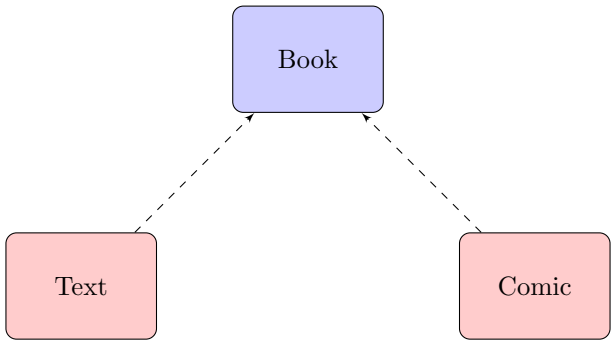
This all stems out of the fact that subclasses do not have access to superclass fields. There are some good reason for this, but if you wish, you may grant access to certain fields by using `protected` visibility

```
Class Book {
     protected :
         string title, author;
         int numpages;
     public :
         ...
};
```

The use of protected is ill advised, as it gives unlimited access to the fields. A better solution would be to make fields private, but provide protected accessor/mutator methods

```
Class Book {
         string title, author;
         int numpages;
     protected :
         string getTitle() const;
         void setAuthor (string newAuthor);
     public :
         Book ( --- ) ;
         bool isItHeavy () const;
         ...
};
```

The sub classes have a "is-a" relationship with the superclass. In UML this is noted using a flowchart style graph which has subclasses pointing to a superclass.



7.3 Virtual Methods and Polymorphism

Going back to inheritance, if we want to initialize a super class, we can use the subclass

```
Book b  = Comic {"A big Comic", "_____" , 40, "_____"};
```

Unfortunately, because of this, any functions called will call the Book version of function. This happens as Book only has enough space for its fields, slicing occurs and the fields from comic are used to initialize the fields of book. Thus, a book is created. This issue does apply to pointers of a superclass. So depending on the pointer type, the object will behave differently, as the compiler uses the type of the pointer to decide which function to run.

```
Comic c {"A big Comic", "_____" , 40, "_____"};
Book *pb = &c; //Book functions are executed not Comic
Book *pc = &c; //Comic functions are executed.
```

Therefore a subclass is only its self, when a pointer or reference of a matching type points to it. This is not what we want. We can ensure the correct version of functions are called by declaring the function `virtual` within the superclass with an `override` parameter in the subclass.

```
Class Book {
         string title, author;
         int numpages;
     protected :
         string getTitle() const;
         void setAuthor (string newAuthor);
     public :
         Book ( --- ) ;
         virtual bool isItHeavy () const;
         ...
};

Class Comic : public Book {
         ...
     pubic
         ...
         bool isItheavy() const override;

};

Comic c {"A big Comic", "_____" , 40, "_____"};
Book *pb = &c; //Comic functions are executed
Book *rb =  c; //Comic functions are executed
Book *pc = &c; //Comic functions are executed.
```

In essence, virtual methods choose which class method to run based on the actual type of the object at runtime.

This also helps us resolve our initial problem of storing multiple different types in an array, as we can declare an array of pointers with a type that matches the superclass. Then when the functions for the class are called, the virtual function will determine the type on run time. This results in a implementation that allows for different behaviour based on the type. This is called **Polymorphism**.

7.4 "Dangerous" Situation

Consider :

```
1  Class One{
2          int x, y;
3      public :
4          One (int x = 0, int y = 0) : x{x}, y{y}{}
5  };
6
7  Class Two : public One {
8          int z;
9      public :
10         Two (int x = 0, int y = 0, int z = 0) : One{x, y}, z {z} {}
11 }
12
13 void f(One *a){
14     a[0] = {6,7};
15     a[1] = {8,9};
16 }
17
18 Two myArray[2] = {{1,2,3} , {4,5,6}}
19 f(myArray}
```

The code above will compile, but since f thinks its working with one objects, the data will end up being misaligned. The resulting array will be $\big[6, 7, 8 \mid 9, 5, 6\big]$.

So never use arrays of objects, polymorphically. If you want a polymorphic array, use array of pointers.