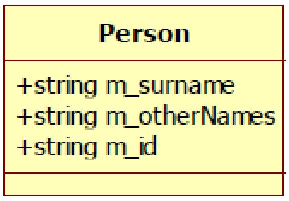
ICT283 Ans 1

Class types-

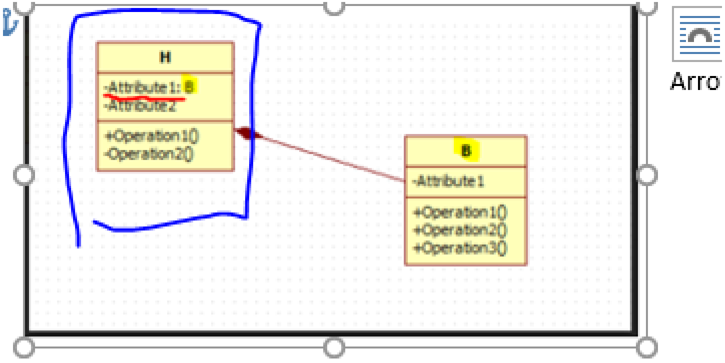
Basic:

- Simple classes whose member variables are all the types built into the language



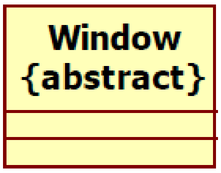
Composite:

- One member variable that is of another class type



Interface: (I know this. Refer to Java)

- Contains zero data members and no implementation (not for creating objects. Subclass it)
- Only consist of an interface (set of methods) that other classes must implement



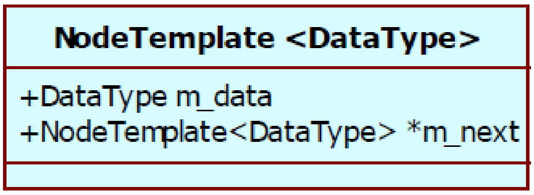
Abstract: (think interface class but with data members) *we use this a lot

- Those that provide an interface (set of methods) but may have data members and some implementation. But everything else must exist in the child class

Template: *we use this a lot

- Have data and complete implementation however the type of data is not instantiated until compile time

- The type that Is descried using a template and instantiating type if known as a bound element
- Template classes will be shown with the template parameter in dashed box



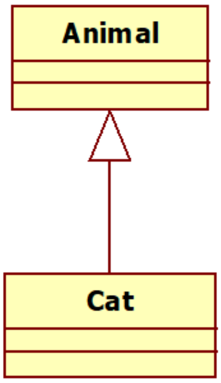
Class relationships-

Specialisation: (think a square is a rectangle or oval is a circle. BAD because rectangle has width but square doesn't square has 1 side length variable for all. Also rectangle has setLength and setWidth while square has setSideLength)

- Class *extends* another class (adds information)
- When a class (Cat) has all the characteristics of another class (Animal) but then adds information it is called specialisation
- Derive a child class (cat) from parent class (animal) and add to the child (cat)
- If you don't follow you break L in SOLID principle

- Rule: You can substitute objects of parents (animal) for → objects of child (cat) w/o effecting program behaviour
- For: The child class (cat) uses *every* member variable declared in the parent class (animal)
- For: The child class (cat) requires *every* method defined in the parent class (animal)
- For: you can relate them in English using x (Parent) 'is a' y (Child) in a behavioural sense
  - Animal is a cat
  - Transport vehicle is a car
  - NOT money is a wallet (makes no sense)

[Important rewatch Lec 3- 1:30:20]

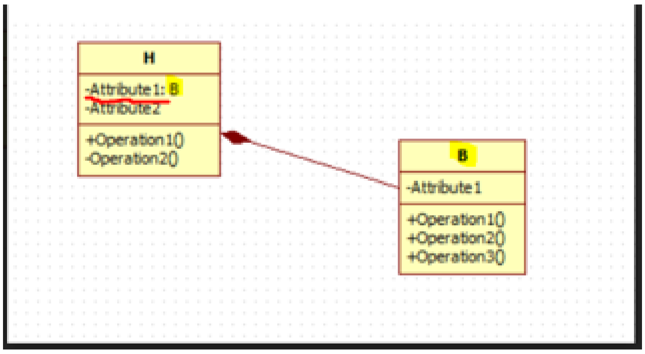


Arrow points to parent

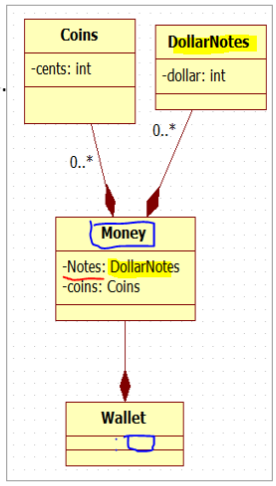
Composition: (logical design. Logical makes sense)

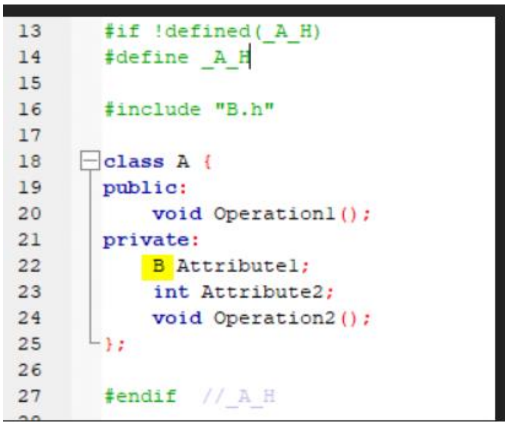
- When the object (parent class/wallet) disappears all the parts (child class/money) disappears
- Include multiplicity unless relationship 1 to 1 (most likely has multiplicity)

- For: you can relate them in English using x (Parent class) 'has a' y (Child class)
  - Wallet has money
  - Apartment has room
  - Class has student (although not a composition but aggregation)
- For: The parent class* has member variables (instance variables) that is the datatype child

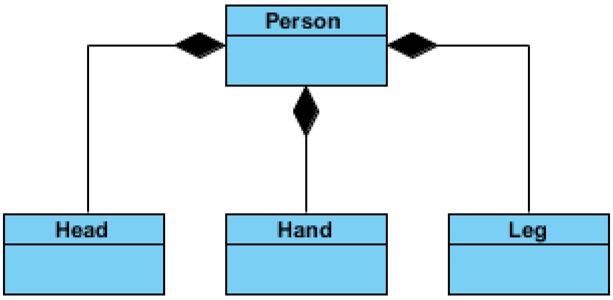


Arrow points to parent (bigger class)

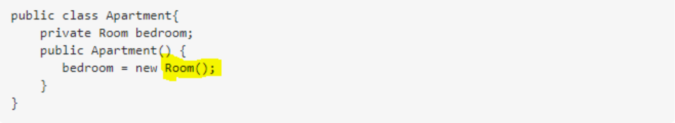




```
#if !defined(_A_H)
#define _A_H

#include "B.h"

class A {
public:
    void Operation1();
private:
    B Attribute1;
    int Attribute2;
    void Operation2();
};

#endif  //_A_H
```



Java version for understanding



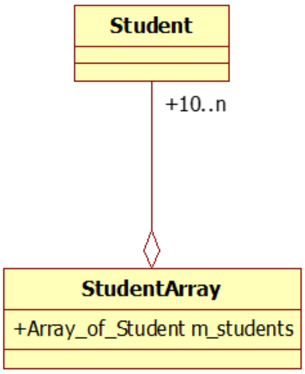
```
public class Apartment{
    private Room bedroom;
    public Apartment() {
       bedroom = new Room();
    }
}
```

***Note**: Don’t get confused not specialisation. There are no base class and parent class in terms of code

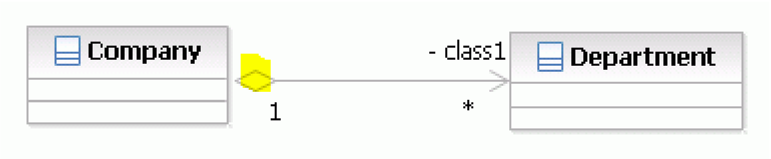
Aggregation: (still a container)

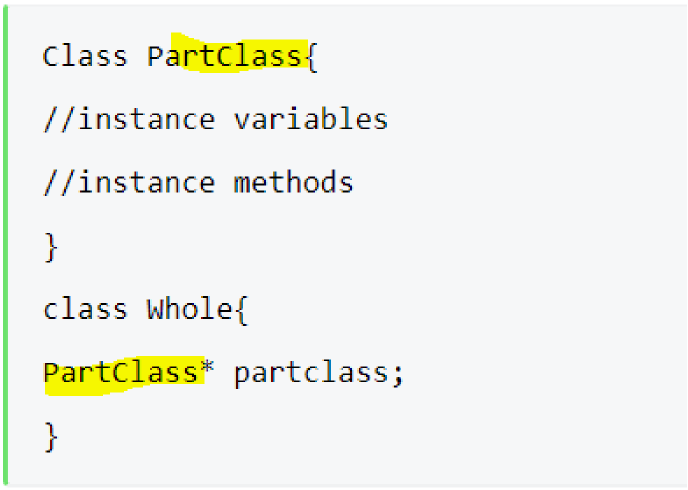
- When the <u>object</u> (parent class/Class) <u>disappears</u> all the parts (child class/student) <u>DOESN’T</u> disappears

- For: you can relate them in English using x (Parent class) ‘has a’ y (Child class)
    - Class has student (although not a composition but aggregation)
- For: one could say that one class *refers* to another class. I.e student → studentList or unit → unit coordinator Not studentlist →unit
- For: The parent class* has member variables (instance variables) that is the datatype child



Arrow points to parent (bigger class)



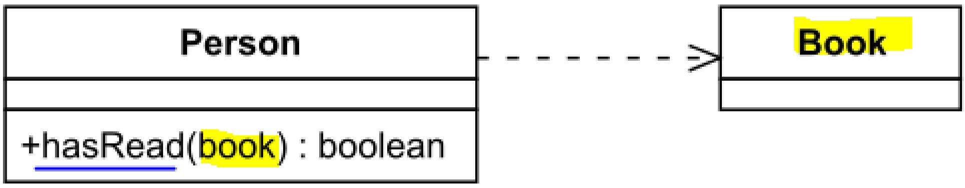


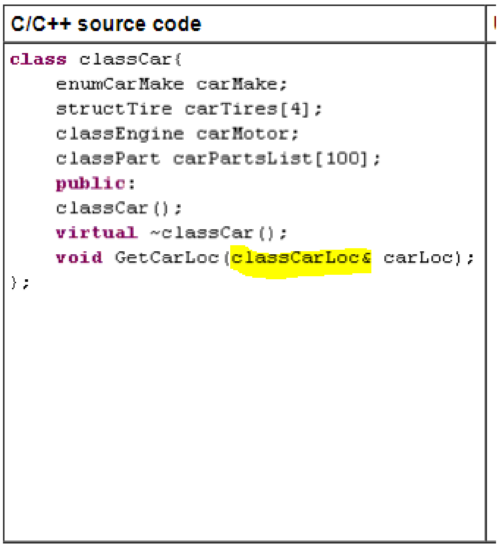
```
Class PartClass{
//instance variables
//instance methods
}
class Whole{
PartClass* partclass;
}
```

(inside private/public demo doesn’t show it. Also not sure about the *)

Dependency: (bad explanation). Bad practice since dependencies create high coupling. Aim to reduce

- Changes to the supplier affects the consumer since the consumer depends on the supplies but not vice versa
- One object invokes another object in order to complete a task (method)
- The method for one class (consumer) has the other class in its parameter
- ~~For: you can relate them in English using x (consumer) ‘uses’ → y (supplier)~~
    - ~~Car (consumer) uses a car lock (supplier)? (Why not has or aggregation? IDK just think object is not in method)~~
    - ~~Product uses a c~~

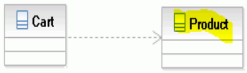


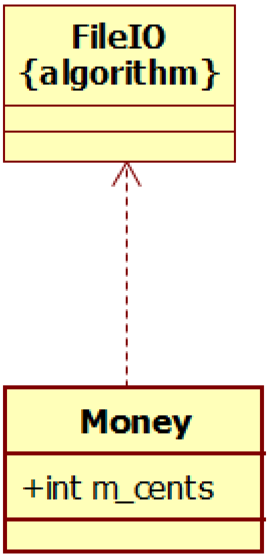


C/C++ source code

```
class classCar{
    enumCarMake carMake;
    structTire carTires[4];
    classEngine carMotor;
    classPart carPartsList[100];
    public:
    classCar();
    virtual ~classCar();
    void GetCarLoc(classCarLoc& carLoc);
};
```

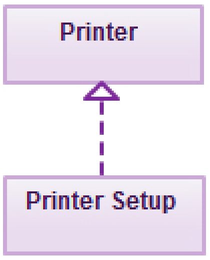
In an e-commerce application, a Cart class depends on a Product class because the Cart class uses the Product class as a parameter for an add operation. In a class diagram, a dependency relationship points from the Cart class to the Product class. As the following figure illustrates, the Cart class is, therefore, the client, and the Product class is the supplier.



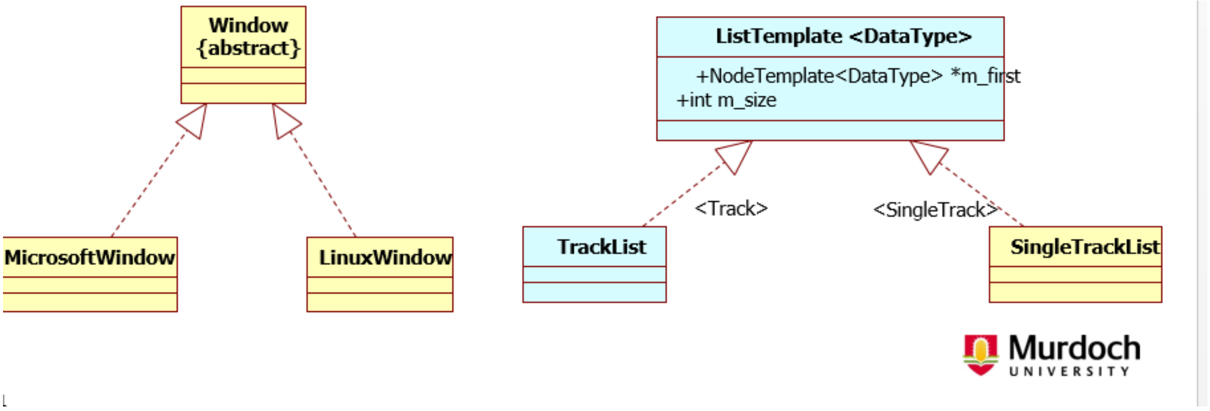


Realisation:

- Refers to abstract, template, and interface classes
- One class implementers the functionality belonging to another class



Realization



Information hiding/Abstraction: (aim to promote)

- Make client code less dependent on an entities implementation details. An appropriate concern for dependency management is reflected in avoiding shared state, applying information hiding and much more (don't expose implementation file .cpp (to the application) give them .h public specification/data structure) NOTE: main.cpp is the application/client program
- Don't put implementation code (.cpp) in specification file (.h) thus exposing it to the application. A single class will have both implementation file and specification file

ICT283 Ans 2

Arrow notation:

[Pointer] -> [Data member] OR (*Pointer).DataMember

So

[Pointer] -> [Data member] = [Pointer2] -> [Data member]

Think of it as referring to [data member] and assigning [pointer2] data member

Deep vs Shallow copying: ICT283Ans

- What is normal ~~copying/Deep~~ copying? Int x = int y
  - "Copy of harry potter book"
  - Means you have two INDEPENDENT variables (x, y)
  - Changes to x DON'T affect y
  - Copies y value → x (Independent + Changes of x don't affect y)

- When you have two same type objects and you assign RIGHT object to LEFT object the POINTER DATA MEMBER by default don't copy across (shallow copy). Ptr x = Ptr y
  - Shallow copy-
    - Means the Ptr x copies the memory address of Ptr y. So essentially Ptr x and ptr y share the same memory address
    - Changes to the content of X or Y affect each other since they point to same memory
    - Also, the memory Ptr x was pointing to originally is not released when discarded because the memory was actually lost. This leads to memory leak
  - Deep copy- (process)
    - DELETE/release memory of Ptr x to stop (build-up of unreleased memory i.e memory leak)
    - Create a NEW memory for Ptr x of the same size as Ptr y
    - Definition: Means instead of copying the memory address the actually memory content/elements/nodes of Ptr y are copied across to Ptr x
- **Destructor for pointer member variable:** is needed because when the object goes out of scope the pointer member which dynamically created an array isn't deallocated. So the array is still allocated even though pointer may be destroyed. Thus update destructor to deal with class object when it goes out of scope

- When class has pointer data member must create ALL 3-
    - Assignment operator
    - Copy Constructor

    - Destructor (delete shared memory address for pointer. Shallow copy)

ICT283 Ans 3

Memory leak is this the build up of unreleased memory that isn't deleted resulting in a decrease in performance. The decrease in performance is because less memory is available. Often with memory leaks it is unintended

Dangling pointer is the situation where a pointer points to a memory location that is deleted or deallocated. This can lead to unpredictable outcomes which is bad. So it's best to allocate the dangling pointers to a nullptr to reduce to unpredictable affects a dangling pointer may have. This means the dangling pointer points is not pointing to any memory location

Struct vs Class

- Struct only public + protected
- Struct contains no invariants properties maintained? (*Think one member affects another member OR one member has restrictions)
    - So does any member in struct depend on each other? Day, Month, Year?
        - Month determines amount of days. So in class
        - Windspeed has restriction so in class
        - Struct can't contain month and day variable as one determines another so put in a class (encapsulate) then put in struct
- Important: Class use
    - Protecting/providing controlled access to data members (**Encapsulation**) is valid ONLY
        - Setters and Getters check for something in regards to data members
            - Having no setters checking anything is stupid and roadblock
        - And invariant is maintained (think as a collection of data members like day,month,year or as a single data member)
- Struct controls access vs long list of pass by parameter
    - OutputResult(int avgWindSpeed, int sdWindspeed, int avgTemp) Vs Struct

Friend declaration input stream: Appears in the class body and grants: - a function or another class access to its private and protected members

```
Input >> [CurrentObj].InstanceMember //Type: class
```

Non-Friend declaration input stream:

```
static [ClassName] tempObject;

Input >> tempObject;

[CurrentObj].SetOBJMethod(tempObject);
```

Friend declaration output stream:

```
Os << [CurrentObj].InstanceMember;
```

Non-Friend declaration output stream:

```
[ClassName] tempObject;

tempObject = [CurrentObj].GetOBJMethod();

Os << tempObject
```

Operator overloading:

- Allows us to extend the definition of operators such as- relational operators, arithmetic operator, insertion operator >>, and extraction operator << for objects

Functional overloading:

[Refer to chapter 5]

ICT283 Ans 5

Encapsulation:

- When data member has invariant meaning there is controlled access to data members. Through setter that check input

Problem with global variables is it provides uncontrolled access to data. Solution

Function encapsulation

File encapsulation

Data member private encapsulation

Useful when you have invariants you need to maintain that is because the data members had invariants and needed controlled access. The month data member needed protection in that it would not accept months greater than 12 and less than 1. So, putting the month and year data member in a struct would not be suitable.

ICT283 Ans 6

Iteration

- Uses less memory because only one call to routine and no need to maintain stack frames
- Does not use the stack so faster than recursion
- Often iteration involves more lines to code

Recursion

- Reduces code complexity and often easier to understand and code
- More memory and clock cycles needed to maintain stack frames. So, more machine instructions executed
- (Very slow in terms of Fibonacci)

Linear search algorithm:

Linear search

Iterative Binary search algorithm: (must be sorted) [linear structure]

[Done]

Recursive Binary search algorithm: (must be sorted) [linear structure]

Sorting algorithms- (Heap, merge sort, then quicksort)

Insertion sort

Selection sort

Bubble sort (<1000 elements)

Merge sort/ Merge sorted containers algorithms: (same as set) O(n log n)

[DONE]

Quick sort O(n log n) BUT if sorted O(n^2)

[DONE]

Heap sort