Class: A data block that can store different types of data as well as function members (actions available for the data). it is defined using the keyword, “class”. the class can further be used to create an object also known as the instance of a class.

Object members: contents of an object. There are two types of object members

1. Data members: represents encapsulated data, also known as properties.

2. Function members: functions associated with the class, also known as methods.

Special member functions:

Getters - member function that gets value

Setter - member function that sets values

Constructors/ destructors:

They are special function members that are called when an object is created or destroyed.

These member functions served a special purpose, they are used for allocation and deallocation of memory.

They do not have a name of their own but instead use the class name.

Constructors can be privatized, which ensure no objects of the class are created

Constructors:

A constructor only gets called when one creates an object

There may be several constructors defined but only one is called for each instance of an object

Named with the name of the class

Different types of constructors:

Default constructor: no parameter/ arguments list required.

Parameterized Constructors: this constructor takes parameters or arguments. They use a special class initializer.

Syntax example:

Animal::Animal (const string &type, const string &name, const string &sound):\_type (type), \_name (name), \_sound (sound){}

: introduces the intialization

Special class initializer: Name\_of\_data\_memeber (value)

Copy constructor: another of the same type to copy data from it to another constructor.

Implicit constructor: When a class does not have a constructor and an object is created in the main. The constructor used to create the object is called the implicit default constructor.

Ttpes of implicit constructors:

Implicit default constructor

Implicit copy constructor: c1 o2 = o3;

Implicit copy operator: o1.setvalue(72);

Explicit constructor:

Syntax:

explicit c1 (const in &value): \_value(value);

The use of the word “Explicit” ensures no implicit conversions take place throughout the program.

Parameterized constructors can be defined explicitly.

Copy operator

Destructor:

Destructor is called when the object is destroyed. Once the program ends a destructor for each constructor is being called automatically.

One destructor maybe be defined in a class

~class\_name();

Implementing function memener outside the class:

Vouid class\_name::function\_name (parameter) {

//code;

}

Object initalization:

class\_name object\_name;

Initializing data members through initializer list:

Class A{

Public:

Int x;

Int y;

Int z;

};

Int main (){

A a={1,2,3}; // initializer list for data memebers

}

Difference between classes and structure:

Everything in structure is by default public whereas in class everything is private.

Array Objects:

Class objects can be stored in an array.

Syntax:

Class\_name \* object\_name = new class\_name [number\_of\_elements];

Delete [ ] object\_name;

Class inheritance:

Reusing code by basing one class over another.

Ithe represents the ability to reuse code by deriving a class from a base class

Ther derived class typically inherits and builds upon some or all of the data and function members of the base class

Multi inheritance:

Class derived\_class : public base\_class1, public base\_Class2 {

//syntax of multiple inheritence

}

Base class:

Members of a base class are used in one or more derived classes.

This class is also known as the parent, super, or non-leaf class.

Derived class:

A class that is being created and is based upon the base class is sometimes called the subclass, child class, or leaf class.

Syntax:

Class Derived\_class:Public Base\_Class {

//code;

}

Access specifiers:

Protected: are only available to base and derived class

Public: are available to all objects that are base class, deviled class, and unrelated objects

Private: only available by the base class

Different types of member functions:

Freind function:

access to a class’s private variables, to another object, or even a few select is granted by the friend declaration.

This exposes all of the private members of the base class to the derived class

Friend classes do not have to be a derived class, they can be any classes that have access to all the private data of a class.

Syntax:

Class abc {

String \_name;

Animal();

Friend class xyz;

};

Virtual function:

This type of member function declared within a class and redefined (overridden) by a derived class

Functions are declared with a virtual keyword in the base class and resolving of a function call is done at run time

When you refer to a derived class object using a pointer or reference to the base class, you call a virtual function, for that object and execute the derived class’s version of the function

The virtual function ensures that the correct function is called for an object

They are mainly used to achieve runtime polymorphism

Adding virtual tells the compiler to look for the same function in the derived class, it might be overridden there.

It is defined in the base and contains no abstract class.

For virtual memner functions we need a virtual destructor

Virtual ~class\_name();

Pure virtual function:

These type of functions are not defined in the base class, and the class contain a pure virtual function is called an abstract class

Virtual function\_name () = 0;

Pure virtual functions do not give you the option of inheriting from the base class.

They do not have a body at all (no implementation.

Any derived class that inherits from a base class that a PVF in it must override the function in the derived class.

An abstract class is a class with a pure virtual function in it.

Polymorphism:

The same function is defined in the base class as well as the derived class. Hence when called from the derived class it calls the derived class one but you can use both the function as well.

Syntax:

Void function\_name () const {base\_class::function\_name; cout<<”This function combines base and derived function;} //this statement creates a derived class memeber function which is overlaoded in the derived class and has a defination in base class

Overloaded members:

They are used for changing or customizing certain behaviors of the member function of the base class

Derived class overload members of the base class

Note: When a member function is overloaded while calling the member function you need to ensure which one are you calling by using scope resolution operation and class name before using the function name.

Class\_name:: function\_name();

**CppCon - Object Oriented Programing by Jon Kalb**

Oop: a programming paradigm in C++ using polymorphism based on runtime function dispatch using virtual functions.

Other alternatives:

Object-based programming

Static polymorphism

Functional programming

Derived object/ classes:

They are objects that are independent libraries that all happen to have the same API, and they do different things depending on the behaviour\state of the individual object.

It is tied with all these different implementations to the application but this commonality (API) is defined by the base class.

Their behavior can be changed at runtime

Base class:

API is defined by the base class, whereas the derived class gives different implementations of these libraries.

Libraries and objects can implement and extend API:

Compile-time: type of the derived object

Runtime: state of a derived class

Separation of concerns:

The idea is that each class is assigned one task that they deal with.

Example:

Logger class does not know anything about logging or how logging is done. Its only concern is when an event occurs it just logs it. It is the user/client or application code.

The derived class now takes care of how logging is done and knows how it is done.

Liskov Substitution Principle:

Ability to replace any instance of a parent class with an instance of one of its child classes without negative side effects

If D is a subtype of B then provable properties of B are also provable properties of type D. Hence, code written to work against the API defined by B will also work correctly for objects of type D, if D is a proper subtype.

Note: Not all derived types are subtypes.

Rule #38 - Safe overriding:

After the base class guarantees the preconditions and postconditions of an operation, any derived class must respect these guarantees

An override can ask for less and provide more but it must never require more and promise less because that would break the contract that was promised to calling code.

Inheritance:

Inheritance is a technique for enabling code reuse should be for two code paths being similar but not the same.

Public inheritance model, “is-a”: Base class defines an interface for an object that might provide a type of functionality. Derived classes provide implementations of different expressions of that object type.

Has-a model vs is-a model:

Composition and aggregation use as a model, for example: if a vehicle and engine are two different classes then the data members of an engine class are a part of the vehicle class. This model is a "has-a" model that states a vehicle has an engine. One class is composed inside another one.

Syntax:

Class vehicle{

Engine e;

};

Inheritance uses is a model that states that function members of a base class will be used by the derived class such that if the vehicle is the base class and car a derived class. A car is a vehicle.

Syntax:

Class car: public vehicle {

};

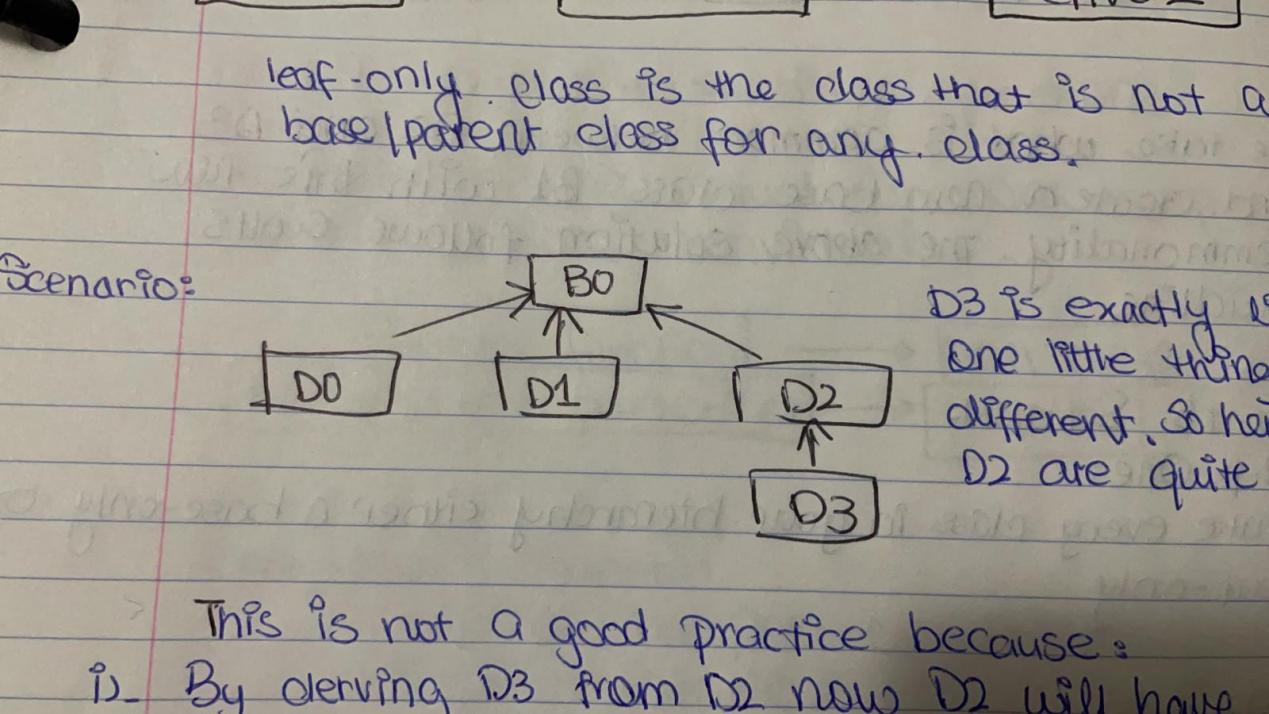
Note: Powerful hierarchy is built on well-defined abstraction.

Best Practices:

Make sure the public inheritance model "is-a". It is substitutability, inherit not to reuse, but to be reused.

Do not reuse your base class code, but instead write a derived class code that can be reused.

Make non-leaf class abstract (has at least one pure virtual function).

Scenario:

D3 is exactly like D2 with one little difference. So hence D3 and D2 are quite similar.

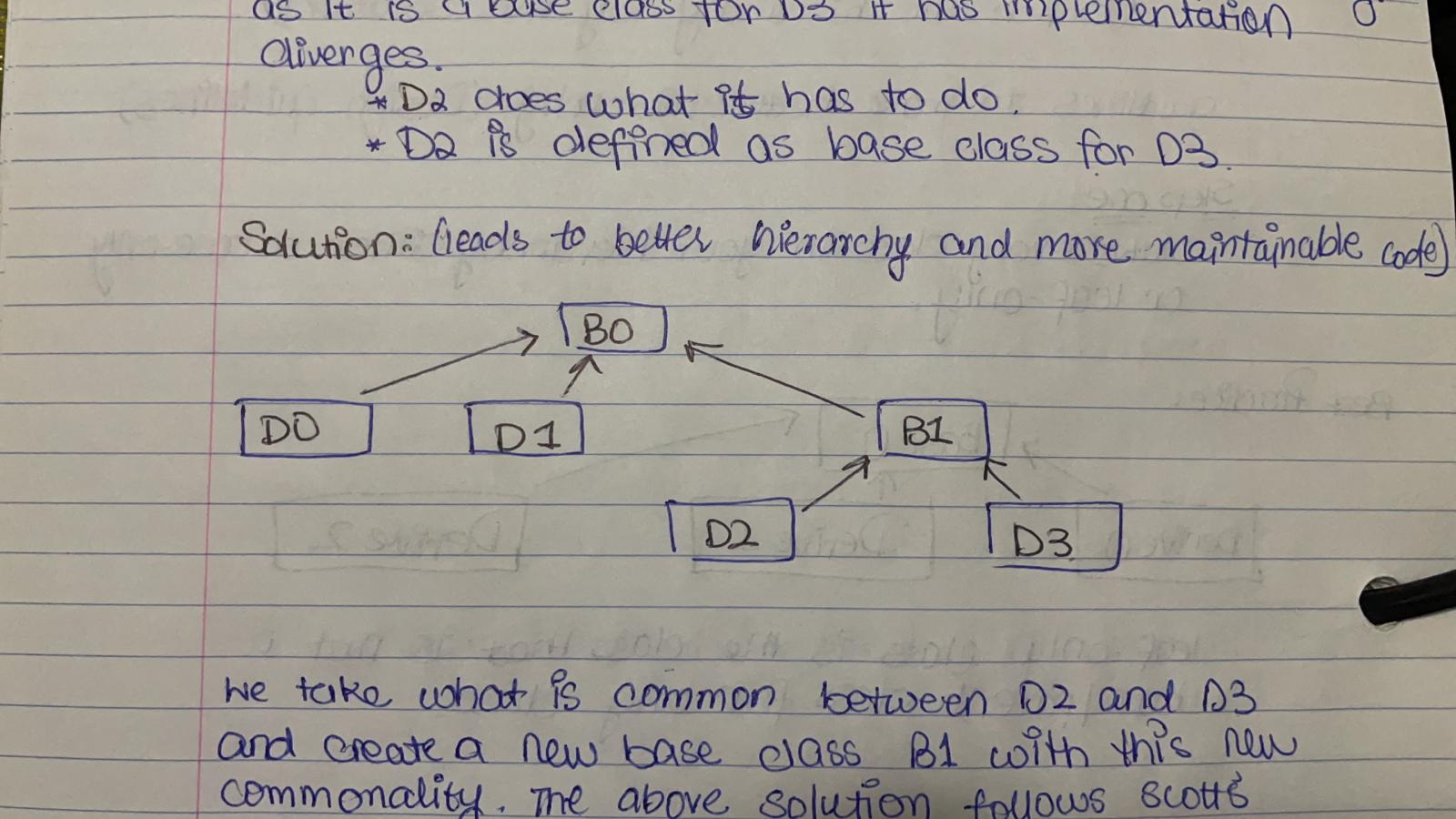
This is not a good practice for the following reasons:

By deriving D3 from D2 now D2 will have new requirements on itself. Hence resulting in the implementation of divergence.

This is a violation of rules number 5 of Herb and Andrei that states: give one entity, one cohesive responsibility.

In this case, D2 is not given one cohesive responsibility as it is a base class for D3 it has implementation diverges.

In this case, D2 does what has to do as well as it is the base class for D3.

Solution:

Leads to better hierarchy and more maintainability code.

We take what is common between D2 and D3 and create a new base class B1 with this new commonality. The above solution follows Scott's guidelines.

Scott’s Solution:

By implementing this solution you get better hierarchies.

Step one:

Make every class in your hierarchy either a base only or a leaf only.

Step two:

Make bases:

Abstract (add one or more pure virtual functions)

Protected assignment operators

Step 3:

Make leaf classes

Concrete (override all pure virtual functions)

Public assignment operator

Final (not added by Scott. To declare leaf classes came in C++11)

Note: you either instantiate it or you derive from it, you never do both from the same class.

Use the non-virtual interface (NVI) idiom:

Consider making virtual functions nonpublic, and public functions nonvirtual.

Virtual function ------------------> Non public (private)

Non Virtual functions ----------> Public

Base Class in control:

Enforce pre and post condtions

Instrumentation

Robust in the face of change: add/remove pre/post-processing without breaking caller or drivers.

Each interface can take its natural shape

Best Practice Build:

Overridden functions must be declared virtual

Always make the base class destructor virtual

Use “override” for overridden function

Do not mix overloading and overriding

Do not specify default values on function overrides

Don’t call virtual functions in constructor\ destructor

Use dynamic rather than static casts for downcasting but avoid casting refactoring where possible.

Virtual destructors:

State of data members of the derived class will not gonna get cleaned up

Undefined behavior to delete a derived class object with a base class pointer if the destructor is not virtual. Always make base class destructors virtual

Overriding keyword:

If you declare a function to be an override and it is not for any reason example if the signature does not match, you can use the keyword, “Override”. This communicates your intent and verifies with the compiler.

It is an override, it is a good practice to write it as an override.

Scoping:

Function overloading rules:

Look for a called name in scope, collect all candidates in the scope and once you find it in the scope you stop looking for it. Overloading does not happen across the scope

If not found in the first scope it starts looking at the next scope and keeps on looking till all scopes end.

Note: If the function is found in one scope it does not go to other scopes and avoid hiding inherited names.

Default parameter values:

Default parameter values are determined by the compile by examining the declaration of the called function.

They are inserted into the parameter list at the call site at the compile-time.

Because the actual function invoked is determined at runtime by address lookup and not by the compiler, the passed default value will also be statically determined as the default parameter in the base class’s function declaration

Upcasting:

Casting a derived type pointer or reference to a pointer or reference to a type further up the hierarchy (to a base type)

Always safe, can be implicit, done all the time

It works because of the Liskov substitution principle

Downcasting:

Casting a base type pointer or reference to a pointer or reference to a type further down the hierarchy

Problematic because the compiler cannot know at compile time that the object is what we are casting to and a base type object cannot, in general, be substituted for a derived type object.

Types of downcasting:

Unconditional Static Casting:

You have to be certain that the object in question is of a required type

Effective and efficient

The compiler cannot verify, undefined behavior if you are incorrect

Uncertain in the face of code maintenance

Condition Dynamic Casting:

A better option is a condition dynamic cast as it does not need to check

The compiler generates code to determine at runtime the object type, hence it is safe and no runtime overhead.

When dynamic cast fails it returns a null pointer rather than pointing to a function.

Note Both the casting need error code.