Abstract and concrete classes

1) What is a pure virtual function?:

- This virtual function is? Overridden in the base class by the derived class.
- A pure virtual function is declared with the? “virtual” keyword and is assigned with 0.
- Here’s an example: Ex: virtual double ComputerArea() const = 0;
- The above declares a pure virtual function ComputeArea()

2) What is an abstract class?:

- The abstract class is a base class. The abstract class is considered an abstract class when it has one or more pure virtual functions.
- The abstract class cannot be?: It cannot be instantiated as an object. The term instantiated refers to the process of creating an object for a class. Going back, the abstract class is still a superclass for a subclass, and the subclass will specify how the virtual function from the base class must be implemented (Just as how we stated earlier)

3) **What is a concrete class?:**

- This is a class that is not abstract. It can be instantiated.
- The concrete class will usually be a derived class to the abstract class. It will override and implement the virtual function

In short there are three things you need to know about a abstract class:
- **It has a pure virtual function (this is needed for run time polymorphism to work)**
- **It will usually not define or implement a function from the abstract class which is typically the virtual function**
- **You cannot create an object of the abstract class (or instance of the class, another way of saying object)**
- **This is where the derived class comes in. You will create a derived class which will override the virtual function.Of course this means the derived class will implement the function that wasn’t defined in the abstract class Because you cannot create an instance of the base class (abstract class) you must do so with the derived class in order to access the members of the abstract class**

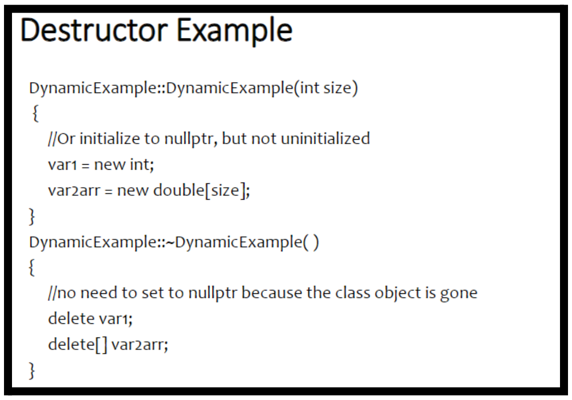
4 **What is a destructor and the purpose of it?: STATIC VS DYNAMIC**

- Destructors are special functions that don't have a return type
- The name of a destructor is the character '~' followed by class name. For example: ~clockType();
- A class can have only one destructor
- A destructor takes no parameters
- A destructor has no return type

- One purpose is when the class object goes out of scope. This happens when the class object is no longer accessible to the program, thus only occurring after the program is over. In other words, once the program is over, the object goes out of scope. However, this only goes for objects with automatic and static storage (essentially regular class objects without the new operator). Objects with automatic and static storage duration will always go out of scope and be destroyed by the end of the program. Okay so the destructor is called when the object goes out of scope within static storage, what happens next?: It basically deletes the class object.

- Another purpose is the destructor with

- **If you're dynamically allocating attributes of the class:** Make sure the desturctor's definition uses the delete operator on the pointer that stores memory address of whatever value you dynamically allocated.



Destructor Example

```
DynamicExample::DynamicExample(int size)
 {
    //Or initialize to nullptr, but not uninitialized
    var1 = new int;
    var2arr = new double[size];
 }
DynamicExample::~DynamicExample( )
{
    //no need to set to nullptr because the class object is gone
    delete var1;
    delete[] var2arr;
}
```

6) **If you're dynamically allocating memory of a class object type:** In the main function, towards the end of the program/main, make sure to deallocate the dynamically allocated memory of the class object type by the delete operator on the pointer. **(Example shown in**

5) **Are there anymore requirements for polymorphism?: Yes**

- **Derived/Base Class Pointer Conversion:** Remember a derived class object is also a base class object, if you make a pointer to said derived class object it also becomes a ptr to base class object. This occurs WITHOUT explicit cast.
- **The base class must have a virtual function:** I'm guessing it needs to be a pure virtual function, in that case there would be an abstract class of some sort.
- **Override key word for the function in the derived class.**

7) **What's the difference between shallow copy and deep copy?:**

- **Shallow copy:** Shallow copy is when you have two pointers pointing to the same memory address with the same content in said memory address. The issue here is the freeing or deleting one of the pointers can cause a memory leak as it causes the other pointer to be unable to access its content.

- **Deep copy:** Deep copy is the opposite of shallow copy. Deep copy is when both pointers have different memory addresses but the same content within that memory address. There is no issue here as deleting or freeing one of the pointers here will not cause a memory leak. Why? Because they both have different memory addresses.

- **What happens if I let class objectA = class objectB ?:** You're letting objects of the same type (class) to have the same memory address and the content stored at the memory address. In the back end, the compiler provides what's called the default copy constructor. Essentially, we just illustrated a shallow copy.

- **How can we override this shallow copy issue?:** You're asking for a deep copy. One method is to create a "copy constructor." The copy constructor is created in the class and then defined inside class. The copy constructor's fuel (parameter) is a object of the same class type. ***If it isn't obvious enough, the purpose of a copy constructor is to have one class object equal to the other class object but avoiding a shallow copy.***

- **Can you give me more characteristics about this copy constructor?:**

    a) Sure, as stated the purpose of the copy constructor is to avoid shallow copy. So in the definition, we should prioritize the attributes ***that are pointers*** to be dynamically allocated. Next, dereference said pointer(s) to access the contents of the memory address, from which will equal the parameter class object's respective value. (The concept is also similar for attributes that are non-pointers as well).

    b) There are at most three ways we can call this copy constructor, however, there is something common with all three ways. When we call the copy constructor on two class objects, we do so as we create one of the class objects.