WEEK 8

Abstract Base Classes, Virtual Functions

AGENDA

- Week 8-1
  - Pure Virtual Functions
  - Abstract Base Class (ABCs)
  - Array of Pointers
- Week 8-2
  - Types
  - Function Bindings

WEEK 8-1

POLYMORPHISM

- The third pillar of OOP is Polymorphism
- Previously we had touched on this in very brief but now we can talk about it in more detail
- We'll be seeing how C++ implements this concept in this week and next week's materials

POLYMORPHISM

- Let's firstly describe polymorphism as:
  - Some code or operations will behave differently in different contexts

OR

  - In C++ this often ends up to mean when we call a member function of an object, the function that is ends up being called will depend on the type of the object.
- There are a few different types of polymorphism but we'll see one of them today

FUNCTIONS

- So far we have seen functions that do various operations and actions
- Some of these functions are attributed to a class, member functions
- Some functions are global scope and perhaps they also help a particular class, helpers

PURE VIRTUAL FUNCTIONS

- Let's now introduce the concept of a pure virtual member function.
  - This is a member function
  - It's a member function that is pure. Pure refers to a function that lacks any implementation detail.
  - Typically a function has a declaration and definition, in this case only the declaration is present.
  - A pure function also needs to be virtual. We'll discuss a bit later what this keyword means.

PURE VIRTUAL FUNCTIONS

- The declaration of a pure virtual function looks like this:

```
virtual Type identifier(parameters) = 0;
```

```
virtual void display(std::ostream&) const = 0;
```

PURE VIRTUAL FUNCTIONS

- In essence the portion from the above that makes the function a pure function is the assignment to 0. This will then tell the compiler that this function only exists as a name/identifier to be referenced but won't have any implementation.

PURE VIRTUAL FUNCTIONS

- As mentioned previously a pure virtual member function is a member function. This implies then that it is a part of a class.
- A pure virtual function by definition doesn't have any implementation. Where then is its purpose?
- Within that class that has pure virtual functions there is also likely the presence of a hierarchy with a parent class and a derived. Recall how classes can make use of inheritance.

PURE VIRTUAL FUNCTIONS

- In a sense then, pure virtual member function (lacking definition/ defined behavior), act more like just an identifier. Something akin to a uninitialized variable perhaps?
- However within the aforementioned hierarchy of classes what would the interaction be if we had a pure virtual function in the parent class and a function whose signature matches it in the derived class?

PURE VIRTUAL FUNCTIONS

A hierarchy of classes that has a display function.

The signatures of each display match each other.

Eg. void display() const;

PURE VIRTUAL FUNCTIONS

If we called the parent class' display function, what do you think happens? Or maybe should happen?

Depending on the context we could either end up in Display A or Display B. What is that context?

PURE VIRTUAL FUNCTIONS

- The thing to keep in mind with these pure functions that in themselves they don't do anything. Their purpose is to expose a sort of generic entry point or portal that can lead us to different and mysterious lands of fantasy and where we end up may depend on the types of our object.
- These entry points also hide away those destinations from the client code. We'll see this a bit later.

ABSTRACT BASE CLASSES (ABCS)

- So far we have worked with structs then later classes as our method of enclosing data and logic into a crisp black box of functionality
- We have also begun to use the classes in conjunction with inheritance and we have seen briefly how that relationship works.
- We'll now introduce another kind of class called the abstract base class (ABC)

ABSTRACT BASE CLASSES (ABCS)

- The ABC has a direct relation to those previously described pure virtual functions.
- The definition for the ABC is that it is a class that contains or inherits at least one pure virtual function.
- Also it's a base class. Meaning they are meant to be inherited from (derived from)

ABSTRACT BASE CLASSES (ABCS)

- An ABC doesn't provide any implementation for its pure virtual functions. This follows the idea on what a pure function is.
- For this reason an ABC cannot be instantiated ie we can't create objects of this class.
- If you try to create one, you'll get a compilation error.

ABSTRACT BASE CLASSES (ABCS)

- If this ABC also doesn't contain any data members declared inside of it then it is also considered to be a pure interface: in other words all it comprises of is those pure virtual functions that may lead to other definitions?

ABSTRACT BASE CLASSES (ABCS)

- To review:
  - We have worked with classes as our method of enclosing data and logic into a crisp black box of functionality
  - We have also begun to use the classes in conjunction with inheritance and we have seen briefly how that relationship works.
  - We now also know about ABCs
- Where do the ABCs fit into our usage of classes?

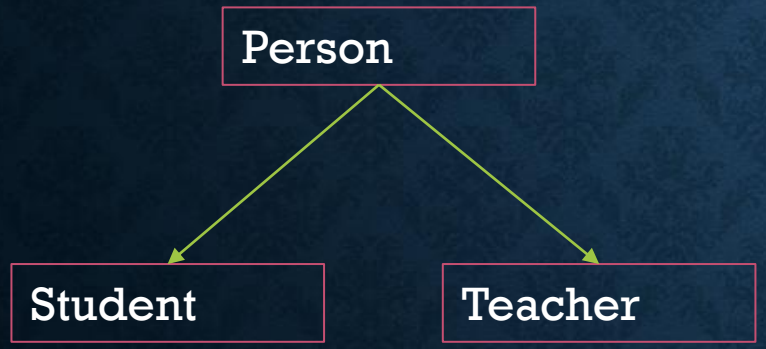
ABSTRACT BASE CLASSES (ABCS)

To expose the purpose of ABCs, recall that both pure functions and ABCs seem to relate to the idea of generic identifiers that may act one way or another depending on the context.

Let's now put it in terms of client code visibility in a class hierarchy.

ABSTRACT BASE CLASSES (ABCS)

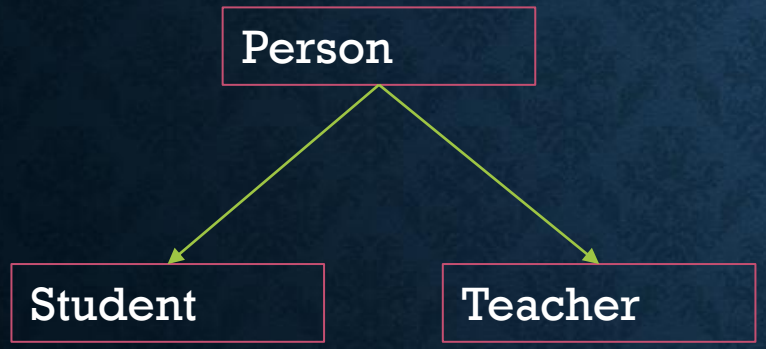
If we consider the idea of a normal class hierarchy:



Person is the parent class that derives into Student and Teacher.

Suppose they all have a function called void display() const;

ABSTRACT BASE CLASSES (ABCS)

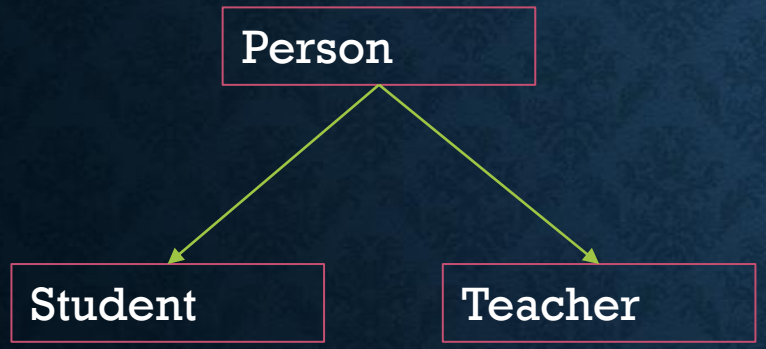


If we wanted to access the Person functions, we would have to create a Person object.
`Person p; p.display();`

If we wanted to access the Student functions, we would have to create a Student object:
`Student s; s.display();`

These lines of code would be in our main function / client code. Meaning our whole class hierarchy is client code visible.

ABSTRACT BASE CLASSES (ABCS)

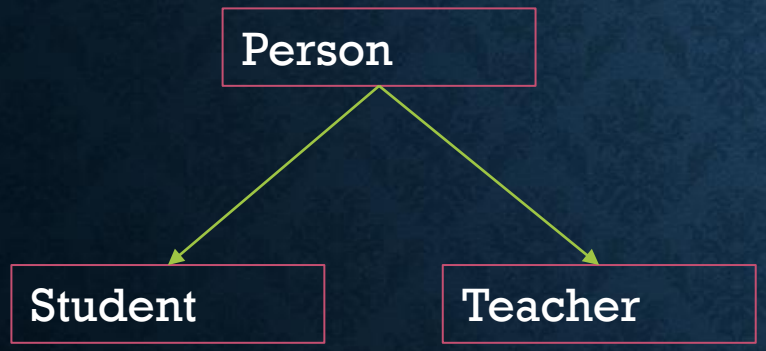


What if we wanted to do two things:

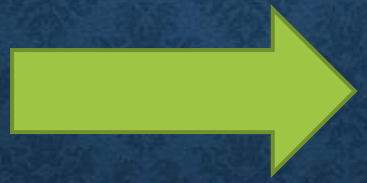
1. Have the ability to access the entire hierarchy from one starting point (versus specifically making instances of each part of the hierarchy)
2. Reduce the amount of exposure of our classes to the client code such that the concept of the black box / crisp implementation is even more crisp

ABSTRACT BASE CLASSES (ABCS)

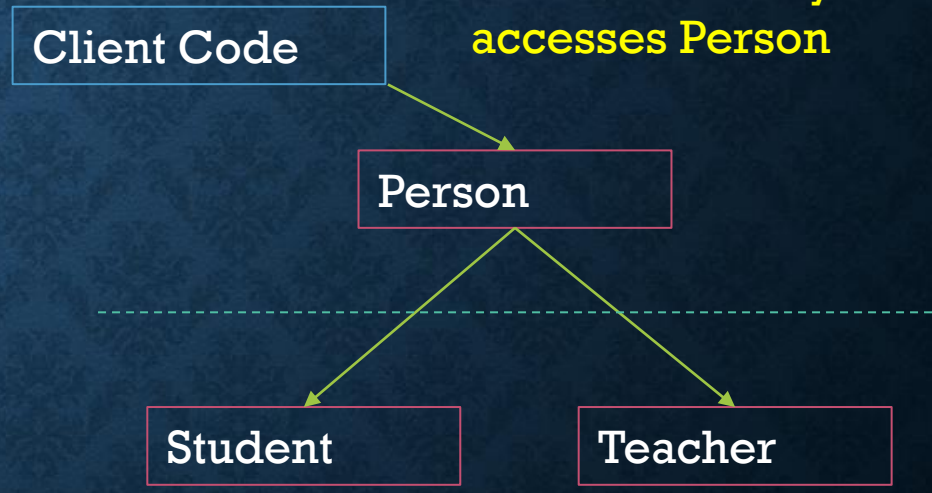
Client code visible



Go from this to this



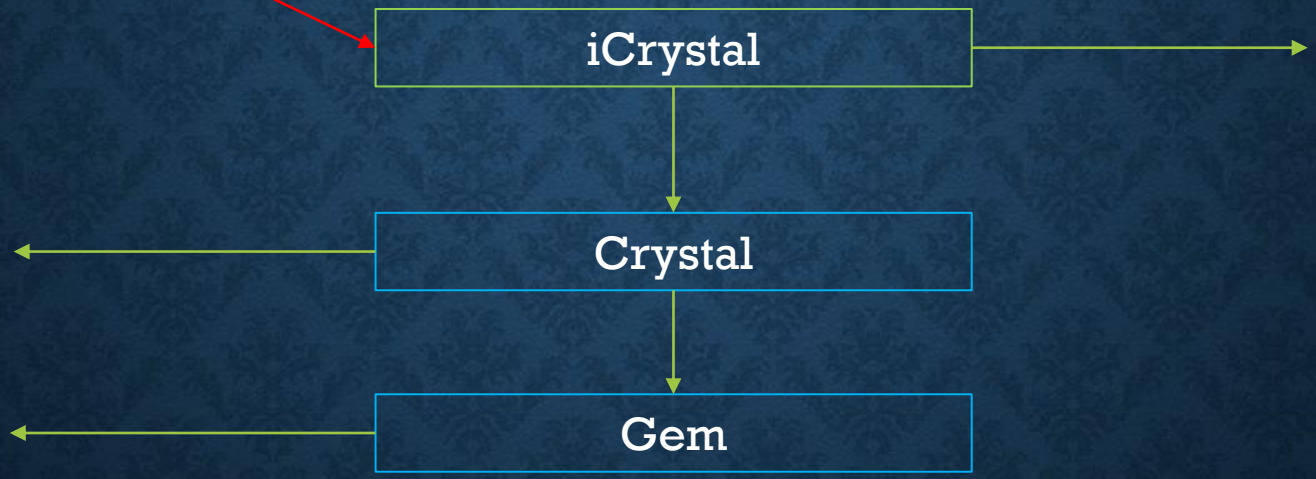
Client code only accesses Person



Hidden from client code

GEM KINGDOM EXAMPLE

User accesses the class hierarchy from here via pointers of iCrystal, the ABC and dynamic allocations of Crystal or Gem.



Abstract Base Class containing pure virtual functions

Derived classes will implement the pure virtual functions from the base

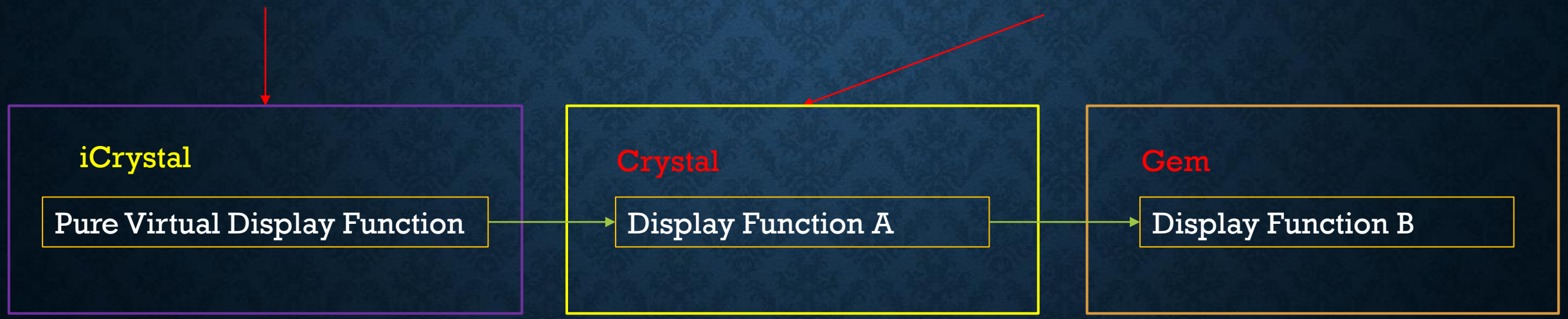
ABSTRACT BASE CLASSES (ABCS)

Consider our previous example with ABCs now in mind. Let's see if we can complete the picture in a generic way:

ABSTRACT BASE CLASSES (ABCS)

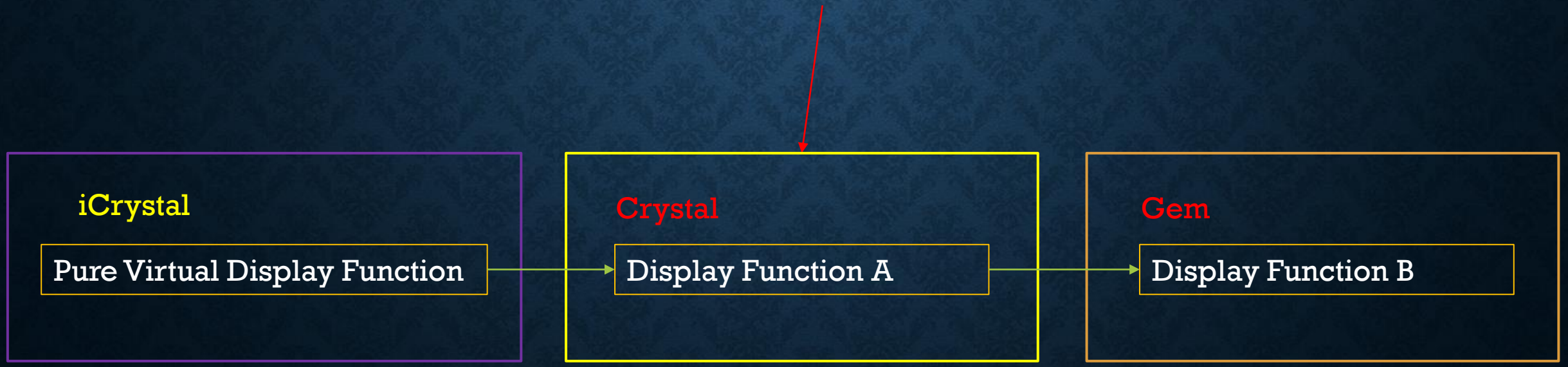
An ABC has at least one pure virtual function. Sometimes it also has no data members at all.

A class that is derived from an ABC also inherits its pure virtual functions. Does this then mean that it too is then a ABC?



ABSTRACT BASE CLASSES (ABCS)

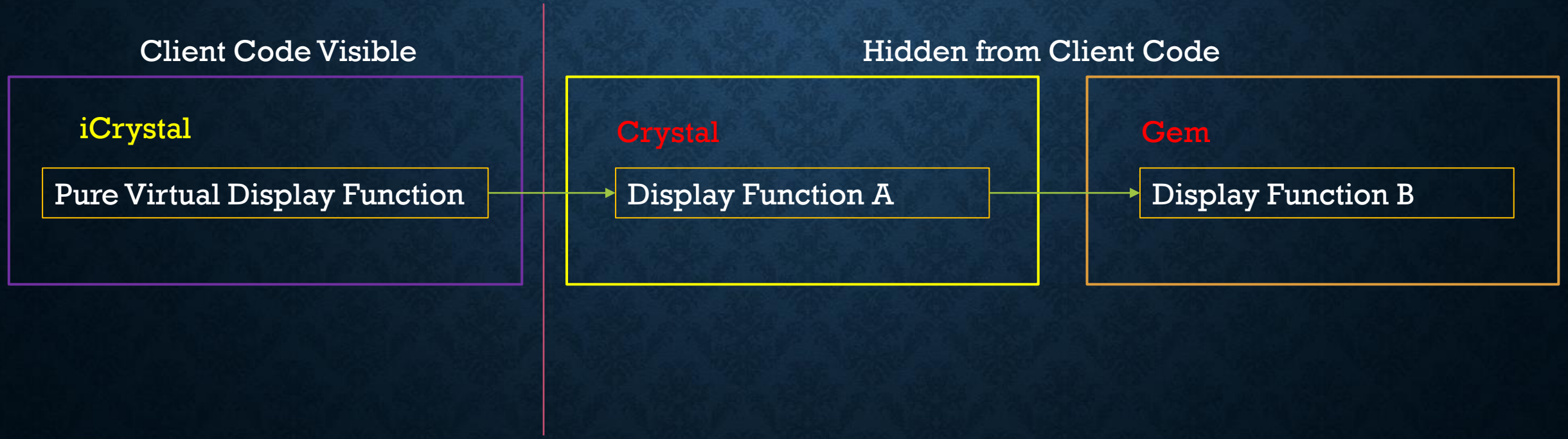
If Class A doesn't give implementation to the display function and continues to leave it virtual and pure then it too is an ABC. However it can implement display and if it does so then it is then called a concrete class. By definition a concrete class is one where it inherits from an ABC and provides implementation for its pure functions. Concrete classes are the same as normal classes so you can create objects of them.



ABSTRACT BASE CLASSES (ABCS)

Recall that both pure functions and ABCs seem to relate to the idea of generic identifiers that may act one way or another depending on the context.

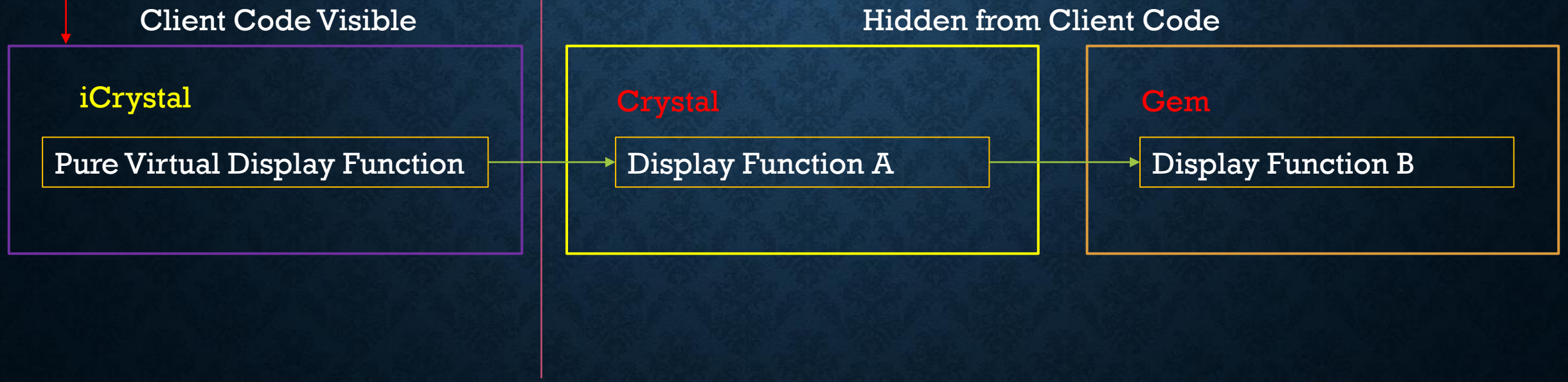
Put in terms of client code visibility.



ABSTRACT BASE CLASSES (ABCS)

Client Code

Now including the entry point from client code that can only access the hierarchy from the ABC.

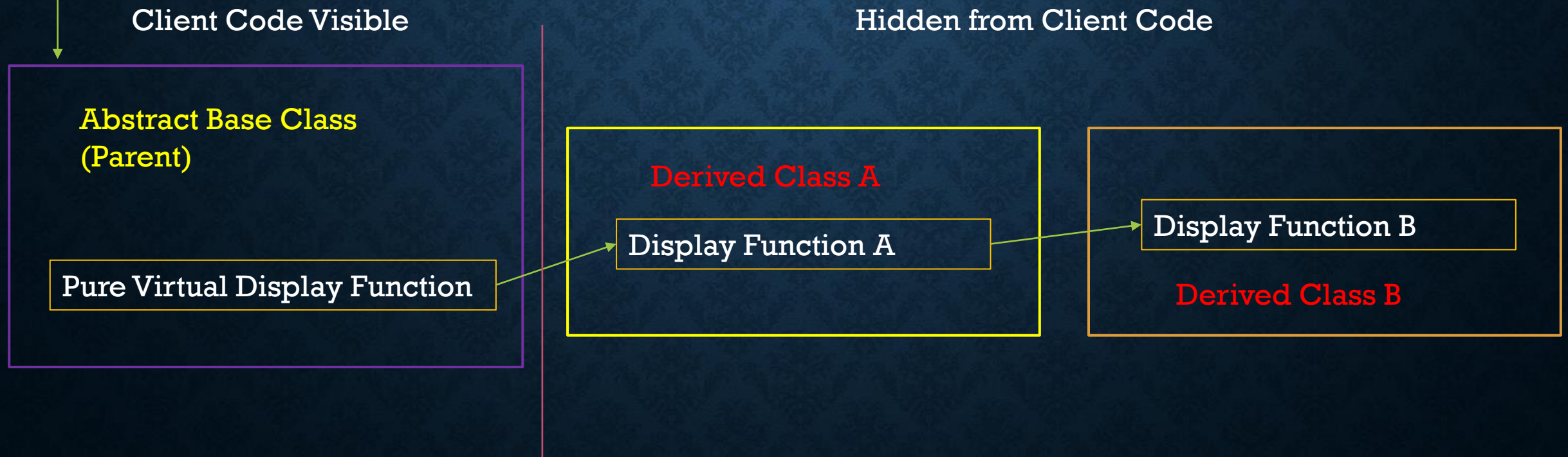


ABSTRACT BASE CLASSES (ABCS)

Client Code

Accessing the entire hierarchy is done through pointers of the abstract base class and dynamic memory:

```
iCrystal * c1 = new Crystal();
iCrystal * c2 = new Gem();
```



ABC ARRAY OF POINTERS

- Having established a hierarchy of classes, we can create classes of that hierarchy and make use of them.
- Through the use of pointers, we can more generically access objects of different dynamic type through the use of an array of pointers of their static type.

ABC ARRAY OF POINTERS

- Consider having an array of pointers of the abstract base class like this:

```
iCrystal* ic [3]; // The type of these pointers are iCrystal, the static type
ic[0] = new Crystal(); // Have this point to a Crystal – the dynamic type
ic[1] = new Gem(); // Have this point to a Gem – the dynamic type
```

This one single array can store different types in the hierarchy

ABC ARRAY OF POINTERS

- This feature allows a form of polymorphism (these objects here are considered to be polymorphic) as we have a single entry point but different behavior based on the 'true' type of the object

WEEK 8-2

TYPES

- Let's review types again briefly here.
- Types in C++ will determine how the compiler will interpret a particular set of data.
- Stating something like int x = 3; will have us interpret that the data like a integer versus a character as in char x = 3;

TYPES

- In our previous example with array of pointers we made mention of two different categories of types, a static type and a dynamic type that an object can have
- The static type can be said to be the type of the object that's known during compile time
- The dynamic type can be said to be the 'actual' type of the object at the time of its creation during run time

TYPES

- C++ allows for these two types of an object through the pointer syntax:
- Examining a previous example again: `iCrystal* c = new Crystal();`
  - The static type of c is the iCrystal class (left hand side)
  - The dynamic type of c is the Crystal class (right hand side)
  - It can be noted that we can deallocate the c pointer and reallocate it to perhaps a different type.
  - This means in essence that the dynamic type can change throughout a program whereas the static type will remain the same throughout.

TYPES

- Remembering Polymorphism, consider a function:
  `void show (const Crystal* c) { c.display() };`
  - This function takes in a Crystal pointer
- If we had these objects and passed them to the show function:
  - `Crystal* one = new Crystal(); show(*one);`
  - `Crystal* two = new Gem(); show(*two);`
- You might be inclined to think they will show different things (one is a Crystal and one is Gem). Do they show different things?

FUNCTION BINDING

- Function binding (which function is bound to a call) is the mechanism that determines which code path our object of a hierarchy should proceed towards (based on its type – the static type or the dynamic type)
- Again if we had the iCrystal => Crystal => Gem hierarchy how should we interpret a call to the display function for example if we were to call it on an iCrystal pointer. It could point to a Crystal object or a Gem object. We won't necessarily know at compile time.

FUNCTION BINDING

- The notion of function binding exists in two forms:
  - Early binding – The object's behavior will be determined through its static type
  - Dynamic dispatch - The objects behavior will be determined through its dynamic type

EARLY BINDING

- Early binding is the default mechanism in place for determining which function is bound to a particular call. It is the most efficient form of binding.
- Reviewing the show function again:
  `void show (const Crystal& c) { c.display() };`

  If we were to pass in `Crystal* one = new Crystal(); show(*one);`
  Do we expect that the Crystal display is called?

  If we were to pass in `Crystal* two = new Gem(); show(*two);`
  Do we expect that the Gem display is called?

EARLY BINDING

- Which one is the static type? That's what determines this call. The parameter is a Crystal type, that's all the show function knows of its parameter. It can't tell whether it's a Crystal or a Gem. Thus it can only rely on the static type of the object to decide what to do. So it will always call the Crystal display as that's what the parameter type specifies.

DYNAMIC BINDING

- Dynamic dispatch is the opposite to early binding, it determines the course of action to take with which function to bind to a call based on the dynamic type of a object.
- Reviewing the show function again:
  `void show (const Crystal& c) { c.display() };`
- We know that the show function can't determine what the pointer c points to at compile time. All it knows is that it is a "Crystal" but that could mean it's a Crystal or a Gem (the dynamic type)

DYNAMIC BINDING

- What if we could insert something that could make that type assessment before the function is called.
- This act of figuring out the 'actual' type of the object is what dynamic dispatch is (versus relying on the specification of the parameter – the static type).

DYNAMIC BINDING

- To enable dynamic dispatch (it's not done by default), we use the virtual keyword
- In essence the purpose of the virtual keyword is to just enable dynamic dispatch
- Given a Crystal class and a Gem that derives from Crystal both containing a display function, if we add the word virtual to the base classes display function this will enable the show function to now make an assessment on which display to call based on the dynamic type of the object

```
class Crystal {

…

virtual void display() const;

};


class Gem : public Crystal {

…

void display() const;

};
```

ANIMAL EXAMPLE