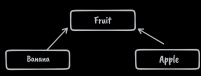
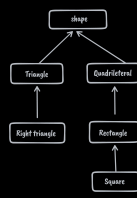
# INHERITENCE




This diagram represents a hierarchy




This diagram goes from general (top) to specific (bottom),
with each item in the hierarchy inheriting the properties and behaviors of the item above it.

A child class inherits both behaviors (member functions) and properties (data members)
from the parent.
Those vars and funcs become members of the derived class.

## Order of construction

When C++ constructs a derived object:
-The most-base class is constructed first,
-Then each child class is constructed in order,
-until the most-child class.

Base
int m_id;
int getId();

Derived
double m_cost;
double getCost();

### What happens when Derived class is instantiated:

1. Memory for derived is set aside (enough for both Derived and Base).
2. The appropriate derived constructor is called.
3. The Base object is constructed:
first: using the appropriate Base constructor
(if no constrctr is specified it uses the default one)
4. The member initializer list initializes the vars
5. The body of the constructor executes
6. Control is returned to the caller.

Creating an instance of Derived,
would construct the Base portion of Derived (using the Base default constructor)
Once finished, the Derived portion constructs (using the Derived default constructor)

NOTE: Before the Derived constructor can do anything, the Base constructor is called first.

## Initializing base class members

```
class Derived: public Base
{ public:
double m_cost {};

Derived(double cost=0.0, int id=0)
// does not work
: m_cost{ cost }
, m_id{ id }
{
}

double getCost() const { return m_cost; }
};
```

This wont work because:
-We need to add another parameter to our constructor (++ wouldn't know what value to initialize m_id to)
C++ prevents classes from initializing inherited member variables in the member initializer list of a constructor.

```
class Derived: public Base
{ public:
double m_cost {};

Derived(double cost=0.0, int id=0)
: Base{ id } //Call Base(int) constructor with value id!
, m_cost{ cost }
{
}

double getCost() const { return m_cost; }
};
```

Why?
If member variables were const or references they'd need to be initialized only once at the start of the program,
and also to prevent the member variables to be initialized twice.
we initialize this way:

In more detail here is what happened:
1. Memory for Derived is allocated,
2. The Derived(double, int) os called,
3. The compiler looks to see if we've asked for a particular Base class constructor,
4. The base class constructor member initializer list sets m_id to 5,
5. The Base class constructor body executes (does nothing, we can print a msg for that)
6. The Base class constructor returns,
7. The Derived class constructor member initializer list sets m_cost to 1.3
8. The Derived class constructor body executes
9. The Derived class constructor returns.

Note that if doesn't matter where in the Derived constructor member initializer list the Base constructor is called,
it will always execute first.

## Protected

allows the class the member belongs to, friends, and derived classes to access the member
protected members are not accessible from outside the class

## Inheritence and Access specifiers

The protected access specifier is only used in inheritence,
allows derived classes to access the base class members

### Public Inheritance

Nothing really changes, the accessibility in the members of the Base class
private stay private and public stay public.

| Access specifier in Base class | Access specifier when inherited publicly |
|---|---|
| Public | Public |
| Protected | Protected |
| Private | Inaccessible |

### Protected Inheritance

-the public and protected members become protected.
-private members stay inaccessible

| Access specifier in Base class | Access specifier when inherited protectedly |
|---|---|
| Public | Protected |
| Protected | Protected |
| Private | Inaccessible |

### Private Inheritance

all members from the base class are inherited as private.

| Access specifier in Base class | Access specifier when inherited privately |
|---|---|
| Public | Private |
| Protected | Private |
| Private | Inaccessible |

## Multiple inheritences

Ambiguity can happen where there are a lot of member
functions with the same name

## Diamond problem (diamond of doom)

Occurs when a class multiply inherits from two classes
which each inherit from a single base class
this leads to a diamond shaped inheritance pattern

### Example

```
class PoweredDevice
{ };
class Scanner: public PoweredDevice
{ };
class Printer: public PoweredDevice
{ };
class Copier: public Scanner, public Printer
{ };
```

Powered device

Scanner

Printer

Copier

## Virtual Base classes

means that there is only one Base object (so the constructor for the base class wouldnt be called twice)

```
class PoweredDevice
{ };
class Scanner: virtual public PoweredDevice
{ };
class Printer: virtual public PoweredDevice
{ };
class Copier: public Scanner, public Printer
{ };
```

Since both derived classes are "virtual", Copier is responsible
for creating PoweredDevice

1. Virtual base classes are always created before non-virtual base classes.
ensures all bases get created before their derived classes.
2. Scanner and Copier do issue calls to the PowDev constructor, it's just ignored
because Copier is responsible for creating it.
If we create an instance of Scanner and Printer their constructors would be called and normal inheritence rules apply
3. The most derived class is responsible for constructing the virtual base class.

## Shadowing Flags

shadowing occurs in C++ when a local variable or function parameter has the same name as a global variable or
another local variable that is in scope. When this happens, the local variable or function parameter "shadows" the other variable,
which means that it hides the other variable and can cause confusion and make the code harder to understand.
The -Wshadow compiler flag enables warning messages for shadowing.
This can be helpful when you are developing code, because if you alert you to potential issues with shadowing that might cause problems later on.
The -Wno-shadow compiler flag disables warning messages for shadowing.
This can be useful if you want to suppress warning messages for shadowing in your code, or if you are working with code that uses shadowing in a way that you are comfortable with.