

# **WEEK 4**

Constructors, Destructors, Current Object

# AGENDA

- Week 4-1
  - Class scope/privacy
  - Constructors
  - Destructors
  - Overloading Constructors
- Week 4-2
  - The Current Object
    - `this` keyword

# **WEEK 4-1**

**Classes, Constructors, Destructors**

# CLASSES

- Last time we spoke about **member functions** in **structs** and subsequently **classes**.
- We touched on the notion of access levels/labels (**public** vs **private**)
- In conjunction with our discussions about objects and classes / structs with logic are in essence what comprises the **Encapsulation** idea of OOP. **Objects** are now more than a generic term but rather we can define it more accurate as **instances** of a **class**.
- Creating an instance of a class looks like the following:
  - **Type** identifier;
  - Eg. **Playdoh** p;

# CLASSES

- The **coupling of data** (member variables) and logic (member functions) together inside of a class is what makes up encapsulation.
- What's interesting to note is that when we create instances of a class like Playdoh while internally **each instance has its own member variables storing their own data, the member functions are only defined once in memory and each instance will share these functions.**
- This is like having a common tool box in which many workers share

# CLASS PRIVACY

- The **private** access label designates what isn't directly accessible within our class and typically we make it so the member data isn't.
- We then label some member functions as **public** allowing only those pathways as entry points into our object
- This is the standard practice.
- However we when are dealing with privacy, the C++ compiler actually applies it as class level privacy. In other words any member function can access any **private member of its class** including the data member of any instance of the class.

# SPECIAL MEMBER FUNCTIONS

- Recall in the previous week we made mention of the three categories that a class' member functions could be.
- One of those categories was regarded as '**special**'
- Those **special functions** are relegated to dealing with the creation, destruction and assignment of objects (of that class)
- These functions will allow for a more fluid and crisp mechanism to create objects
- To review an object is an **instance** of a **class**.

# CONSTRUCTORS

- Up to this point what we've done is instantiate an object and then use public member functions to set the values of the member data:

```
playdoh p; // Instantiate  
p.setWeight(100); // Set  
...
```

- This creates a problem. Because we aren't setting the value of the member data during instantiation we can reach a situation where we attempt to use data that has not yet been defined. What if we had called the display function before setting our data? What would happen?

# CONSTRUCTORS

- Having a mechanism to set our objects at creation time would solve this issue
- The first of those special member functions, the constructor fulfills this purpose
- The declaration for the default constructor function looks like the following:
  - Type();
  - Given a class of playdoh it would look like: playdoh();

# CONSTRUCTORS

- Some additional points on the default constructor – `playdoh()`;
  - It's a zero argument/parameter function.
  - We use this constructor to set an object's default/empty state and any other preliminary behavior
  - Note that this function doesn't have a return type (not even void)
  - If we don't include a default constructor explicitly the compiler will insert an empty one in for us

# CONSTRUCTORS / SAFE EMPTY STATE

- Utilizing the **default** constructor we can control the **safe empty state** and by having every object start in that state can allow us the ability to set up default behavior for our other functions such as `display()` functions
- These `display()` functions can now then differentiate between an **empty object** and a **non empty one**. Perhaps we want to show different things for each of these cases (eg. '`data not available`' for an empty object)

# DESTRUCTORS

- Complimentary to the constructor, there is the **destructor** which handles objects that **reach the end of their life time** (essentially meaning objects that reach the end of their scope/go out of scope)
- The destructor is mainly used to do any kind of clean up with respect to an object
  - This basically means handling any kind of deallocation of memory for dynamically allocated resources.

# DESTRUCTORS

- The declaration for the destructor looks similar to the constructor:
  - `~Type();` // Note the ~ symbol
  - Given a class of playdoh it would look like: `~playdoh();`
  - A destructor has **no return type** like the constructor and **never has any parameters/arguments**

# DESTRUCTORS

- A few points about destructors:
  - They are called **automatically** when an object reaches the end of their life time
  - As such we should **never explicitly call the destructor** (as it is supposed to handle clean up of objects we don't want to unintentionally clean up)
  - **Cannot be overloaded.** There can only ever be one destructor function.
  - If we **don't include a destructor explicitly** the compiler will insert an **empty one in for us**

# CONSTRUCTORS AND DESTRUCTORS

- While the constructors and destructors have a certain purpose (creation and clean up of objects), they are just functions at the end of the day and as such we can put whatever we want in them.
- Of course we should take care to know when these functions are called

# CONSTRUCTION AND DESTRUCTION ORDER

- The order of construction and destruction are **the inverse of one another**
- To be clear the order of construction/destruction means at what point the constructor/destructor is called for a given object
- The typical timeline is like so:
  1. **The constructor is called upon the creation of the object**
  2. **The destructor is called when that object reaches the end of its lifetime**
  3. Any memory allocated for that object is then released

# CONSTRUCTION AND DESTRUCTION ORDER (MULTIPLE)

- The question may then come up in that if we had created a bunch of objects which one of those is created first? Is it sequential? Which one is destroyed first? Does that follow the same order?
- The answer to this is that creation is sequential whereas destruction is sequential in reverse
- Given the following:

playdoh a;

playdoh b;

playdoh c;

Playdoh a is created **first** and destroyed **third**

Playdoh b is created **second** and destroyed **second**

Playdoh c is created **third** and destroyed **first**

# CONSTRUCTION AND DESTRUCTION ORDER (ARRAY)

- Similarly we consider if we were dealing with an array of playdoh rather than individual instances of playdoh, the idea is very similar
- Given the following:

**playdoh** a[3];

a[0] is created **first** and destroyed **third**  
a[1] is created **second** and destroyed **second**  
a[2] is created **third** and destroyed **first**

# CONSTRUCTION AND DESTRUCTION ORDER

- In summary the order of destruction is the reverse of the construction
- It's a first in, last out (FILO) mechanism

# OVERLOADING CONSTRUCTORS

- If we have the need to more than just the default constructor (very often we do) we can **overload the default constructor much like any other function**
- By doing we can then have a multitude of ways to safely create our objects
- A declaration for this overloaded constructor would look like so:
  - `Type(param1, param2 ...);`
  - For a class of playdoh it might look like: `playdoh(char colour, int weight);`
- Now these constructors can take the place of our set functions

# **WEEK 4-2**

Current Object, this

# CURRENT OBJECT

- Within the context of a member function of a class we are able to refer to the current object's data members without explicitly stating it
- Eg. playdoh's weight and colour can be accessed in reference to the current object inside of the display functions

# THIS

- The keyword **this**, can be used to refer to the memory in which the current object is stored **when you are inside of a member function**
- In other words it points to the **address** of where the object is stored
- Dereferencing it like so **\*this** will give you the actual object

# THIS

Current Object

```
int weight;  
char colour;  
...
```

Member functions  
...

# WHAT CAN WE DO WITH “THIS”

- Some of the uses of the **this** keyword can be:
  - To **differentiate** parameter names with member data names if we were to use similar naming
  - We can **return** the current object by returning **this**
  - We can **assign** to the current object by using **this**

# THIS

- Differentiating the parameters from the member data can be done though the **this** keyword:

```
void playdoh::setColour(char colour){  
    this->colour = colour;  
}
```

Without ‘**this**’ the compiler would not be able to tell the difference between the current object’s **colour** and the **parameter**. We would end up interacting with the parameter **colour** only.

# THIS

- Sometimes you might want to return the current object (either a copy or a reference) for one of your member functions and you can do so with **this**. Note that we're dereferencing before returning.

```
playdoh playdoh::retCopy(){
    return *this;
}
playdoh& playdoh::retRef(){
    return *this;
}
```

We'll see some more uses of returning **\*this** later in the course

# THIS

```
void playdoh::makePlaydoh(){

    char col;
    int wei = 0;

    cout << "Let's make playdoh" << endl;
    cout << "Enter a char for the colour: ";
    cin >> col;
    cout << "Enter a int for the weight: ";
    cin >> wei;

    playdoh temp(col, wei);
    *this = temp;
}
```

The use of **\*this** in conjunction with assignment allows for use to assign another same-type object to our current object.

In this case we're creating a temp playdoh that has gone through validation (via the constructor) then assigning it to our current object.

# THIS

```
void playdoh::makePlaydoh(){

    char col;
    int wei = 0;

    cout << "Let's make playdoh" << endl;
    cout << "Enter a char for the colour: ";
    cin >> col;
    cout << "Enter a int for the weight: ";
    cin >> wei;

    playdoh temp(col, wei);
    *this = temp;
}
```

While this is a technique to both validate and assign data to our objects, **do be mindful that this approach ends up creating additional temporaries / has extra copying of data involved.**