WEEK 3

Members, Privacy and I/O stream

AGENDA

- Week 3-1
  - Members (Functions)
  - Privacy
  - Empty State
- Week 3-2
  - I/O Stream

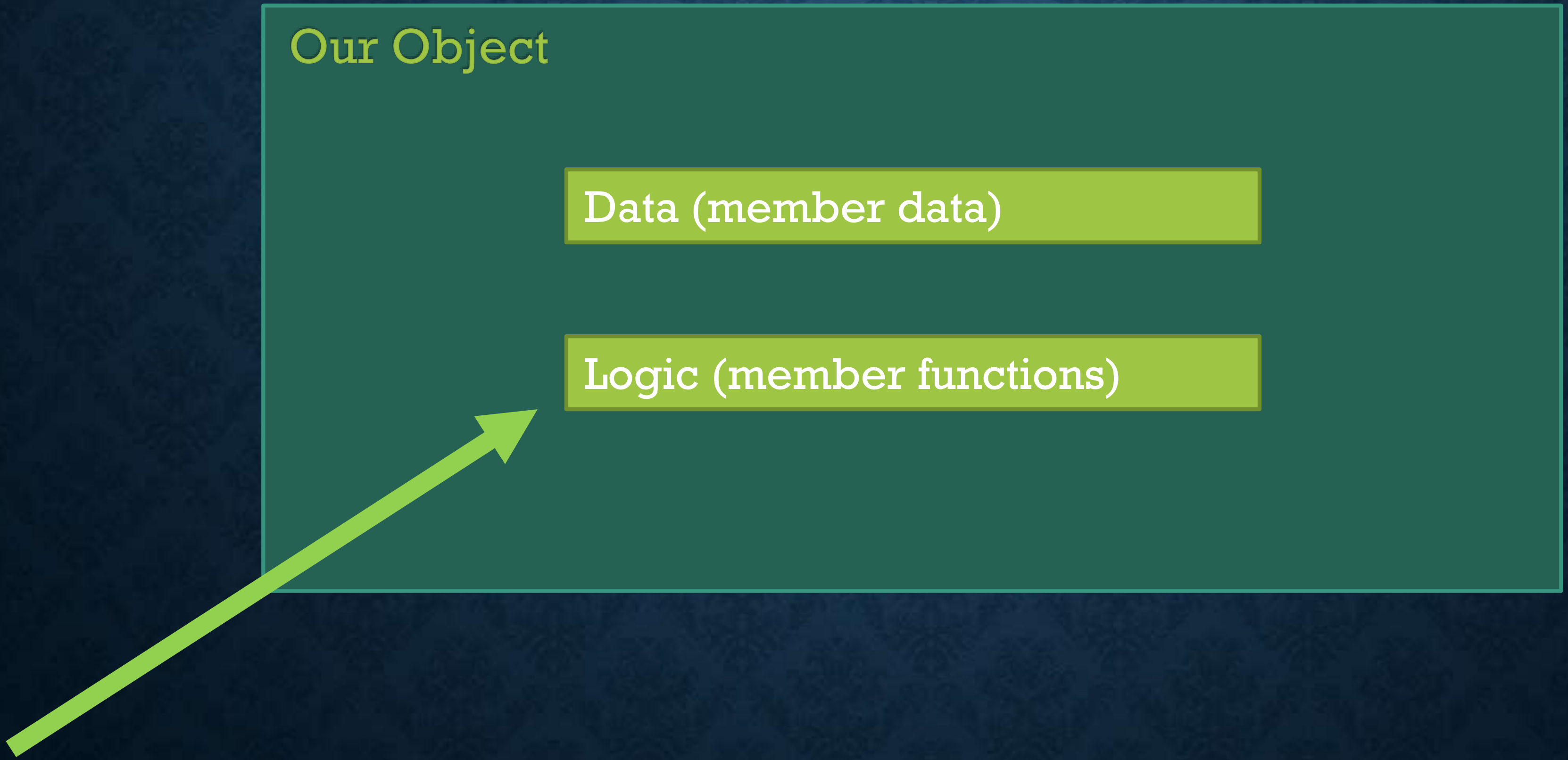
WEEK 3-1

Member Functions

ENCAPSULATION

- A reminder that Encapsulation is one of the main concepts behind object oriented programming
- The coupling of data and logic inside a structure that can cleanly hide the details from any clients that would use it, this sort of appliance like interface is Encapsulation
- This structure in C++ takes the form of structs and going forward will mainly constitute of classes.

MEMBERS



MEMBER FUNCTIONS

- Member functions are functions with an association to an object.
- These functions allow for a client to access the object's data and perhaps also to change that data.
- There are three main types of member functions
  - Queries – These functions report the state of the object (eg. get functions). Queries make use of the const keyword because these functions don't change state but only report it we can make sure this isn't violated by telling the compiler that these functions are non modifying const functions. If we violate this rule then the compiler will spit errors at us.
  - Modifiers – These functions change the state of object (eg. set functions)
  - Special – These functions are used to create, assign or destroy objects
- These member functions have direct access to the object's data.

MEMBER FUNCTIONS

- Let’s recall our knowledge of structs and add a member function declaration to it. The main effect is now that the functions are enclosed in the playdoh struct much like the member data variables.

```
//playdoh.h
// global functions

struct playdoh {
  char colour;
  int weight;
};

void setColour(char c, playdoh* p);
void setWeight(int w, playdoh* p);
void display(playdoh* p);
```

```
//playdoh.h
// member functions

struct playdoh {
  char colour;
  int weight;

  void setColour(char c); // member
  void setWeight(int w); // member
  void display() const; // note the const
};
```

MEMBER FUNCTIONS

- Having put in declarations to the playdoh struct, they now require definitions to give them some meaning.
- When defining a member function, similar to our use of namespaces we have to specify the scope affiliation of the member function (as they are affiliated in particular to a struct or class)
- For example with the display function previously declared may have a definition like so:

```
// playdoh.cpp
…
void playdoh::display() const {
  cout << “Playdoh Info” << endl;
  cout << “Colour: “ << colour << endl;
  cout << “Weight: “ << weight << endl;
}
```

Note we can reference the name of the member data variables directly because this function is of the same scope

MEMBER FUNCTIONS

- Note that when working with member functions there is a sense that we are working also within a scope local to the struct/class that the function is associated with.
- When we access a name it will be local to the struct.
- This is an example of struct or class level scope. That is why when giving definitions to these functions we have to have a prefix of playdoh:: so we can explicitly be sure which display function we are giving meaning to (playdoh's display rather than any other)
- By having a shared scope within a struct we can directly access any of its members (both data and functions)

MEMBER FUNCTIONS

- Calling a member function is similar to accessing a member variable of the struct instance:

```
playdoh p;
p.colour = ‘r’;
p.weight = 100;

p.display(); // calling a member function
```

GLOBAL SCOPE VS CLASS (STRUCT) SCOPE

- When inside of a member function your scope will be local to the struct or class.
- Thus meaning any name you reference will match the entity that is most local. If there is a another entity let’s say a function that has the same name as one local to the struct but it exists in the global scope, it will be shadowed by the struct’s member function when we are working within.
- However we can still access this global version with the use of the scope resolution operator :: (the double colons)
- Using it in this way looks like so:
  - ::display() // This will refer to the global version of display rather than one local note that it looks an empty prefix as opposed to playdoh::display()

STRUCT ACCESS

- Access to the data members of the struct are directly accessible from any scope and as such any client code can manipulate this data as it pleases
- This kind of access is the default for structs
- This type of access is referred to public access
- However is this the type of access we'd want all the time? If any client code can manipulate the data without regulation, this may result in actions that violate the design of our structs/objects

ACCESS LEVELS / LABELS (PRIVATE VS PUBLIC)

- There are two keywords: public and private we can use to designate the level of access to a particular member of our struct/class.
  - public will indicate that these members of the object are accessible by 'anyone' or any client code
  - private will indicate that these members of the object are only accessible by members of the object or anything within the same 'class scope'. They are otherwise inaccessible by 'anyone'
- By doing this we can restrict the ways our structure will be accessed and this then can allow for a more managed object
- This managed object will hopefully also not allow for any interaction with it that violates its design

ACCESS LEVELS / LABELS

```
//playdoh.h
// member functions

struct playdoh {
  private:
  char colour;
  int weight;
  public:
  void setColour(char c); // member
  void setWeight(int w); // member
  void display() const; // note the const
};
```

These variables are no longer accessible directly

CLASS ACCESS

- As compared with structs where the default access is public, the default access level for classes are private if not specified.
- Class will generally be used from this point on in the course due to their leaning towards a private access default
- The syntax for classes is in essence the same as structs, replacing the keyword for another

```
//playdoh.h
// member functions

class playdoh {
  char colour; // private by default
  int weight; // private by default
  public:
  void setColour(char c); // member
  void setWeight(int w); // member
  void display() const; // note the const
};
```

MODIFYING PRIVATE DATA

- Knowing now that private data can’t be accessed directly from the outside, how should we then deal with interaction with this privatized data?
- What can be done is the use of public member functions
  - As member functions exist in the same scope as the member data, they are able to access those variables or entities
  - By making the member functions public, client code can then access the object’s data via these functions
  - Doing it this in manner allows for a more controlled access to our objects (of our design)

EMPTY STATE

- Consider that now we have the ability to limit the kinds of the access to the object via specified functions (of our design), we should also start considering designing these objects so they are always in a valid state
- In particular we should have a state in which we can consider the object to be empty
- Knowing that an object is empty is quite useful as we can then have a set of default behaviors for that object
  - If our playdoh has a weight of 0 perhaps that's our empty state. If that's the case perhaps then we shouldn't display the object in the same way we always do.
  - If an object is empty perhaps some actions don't' make sense to do. Querying for an unset variable doesn't do much for us

EMPTY STATE

- Considering our playdoh once again, a negative weight doesn't really make any sense. If we were to say that a weight of 0 is our empty state, our setWeight function perhaps shouldn't then accept any negative numbers at all.
- Upgrading the setWeight function to do a bit of validation and set things to an empty state if the provided values for setting the weight aren't valid. This allows us to have greater control on what our object can and should be.

NEXT STEPS

- Those aforementioned special member functions that are focused on creation, destruction and assigning of objects alleviate some of the cruft of setting up the state of our objects
- This will be covered next week.

WEEK 3-2

I/O Stream

IOSTREAM

- The iostream library is the C++ library that provides a standardized mechanism for both outputting to a standard output device as well as accepting input from standard input devices
- The interface that this library utilizes for these features are the cin and cout objects.
- Recall that objects have types much like any entity, the type of objects from the iostream library are also of the iostream type (or a derivative of it).

INPUT / CIN

- The cin object is instance of the istream type.
- In latter weeks we'll revisit what meaning this typing holds
- The typical usage of cin is:
  - cin >> identifier
  - This usage has it so the input from the cin is fed into the identifier
  - cin unlike scanf doesn't require a format string to understand what type the identifier is

- When dealing with c-style strings, the cin object will also append a nullbyte at the end automatically.

INPUT / CIN

- When accepting input the cin object skips leading and trailing white space as it will use that white space character as a delimiter
  - In other words when inputting data a string of '   abc   ' will be the same as simply inputting 'abc'
  - When providing multiple values to the cin via multiple calls, the cin object knows when a value begins and ends by the white space that separates them

INPUT / CIN

- The cin object has some public member functions that may be useful when dealing with standard input:

- ignore(...) - ignores/discards character(s) from the input buffer
- get(...) - extracts a character or a string from the input buffer
- getline(...) - extracts a line of characters from the input buffer

- We'll be looking into these functions more in depth later on in the semester.

OUTPUT / COUT

- The cout object is of the ostream type
- The typical usage is:
  - cout << identifier
  - This usage has it so the data from the identifier on the right is fed into standard input on the left
  - This can be done on a individual identifier bases with multiple cout calls or it can be done in cascading fashion all in one line:
    - cout << identifer1 << identifier2 << identifier3 …. << endl;

- The endl inserts a newline into the output and then flushes the buffer

OUTPUT / COUT

- The cout object like the cin object offers some public member functions that can assist in formatting of outputs
- This formatting was accomplished via format codes in C
- Some of these functions are:
    - width(int) - sets the field width to the integer received
    - fill(char) - sets the padding character to the character received
    - setf(...) - sets a formatting flag to the flag received
    - unsetf(...) - unsets a formatting flag for the flag received
    - precision(int) - sets the decimal precision to the integer received
- We'll be looking into these functions more in depth later on in the semester as well as other ways to manipulate output