WEEK 5

Operators, Helper Functions

AGENDA

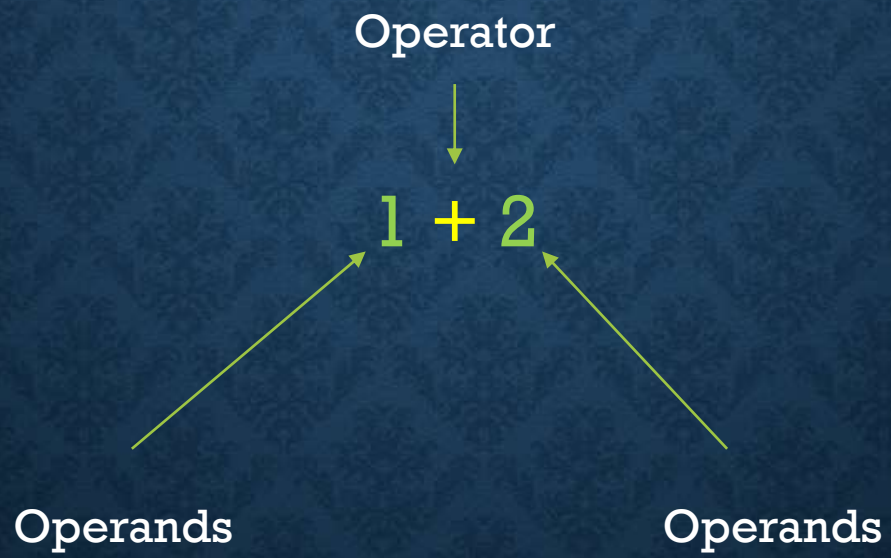
- Week 5-1
  - Operators
  - Operator Overloading
- Week 5-2
  - Helper Functions
  - Helper Operators
  - Friends

WEEK 5-1

Operators

OPERATORS

- When considering an expression such as 1 + 2, it can be broken down into operands and operators:



OPERATOR

- The operator defines the action to be enacted upon its operands (the target of those actions)
- In C++ there are many built-in operators that we can use and have used up to this point

OPERATORS

- There are a few types of operators:
  - unary - one operand - post-fix increment/decrement, pre-fix increment/decrement, pre-fix plus, pre-fix minus
  - binary - two operand - assignment, compound assignment, arithmetic, relational, logical
  - ternary - three operands - conditional operator
- C++ however does not allow for us to define new operators. We have to use what's available from the pre-existing ones.

OPERATORS

- The use of operators allow us to form expressions that can be evaluated and then perform some desired action.
- While they can be as simple as doing arithmetic with fundamental numeric types, C++ allows for more.
- Consider the following:

```
Playdoh p1;

p1 += 5;
```

OPERATORS

```
Playdoh p1;

p1 += 5;
```

- Does the above expression of adding two playdoh's together make sense? Is it something perhaps we should be able to do?
- The left operand in this case is our `p1`aydoh object (current object) and the right operand is a number

OPERATOR OVERLOADING

- When we want to define or describe how an operator should work with a non fundamental type (typically a user defined class) we can, in a similar way as with functions, overload an operator so it may then understand how to work with an operand of some user defined class type.
- Because we can not define new operators we have to overload existing ones to define this new behavior for our user defined objects

OPERATOR OVERLOADING

- The key to this is the use of the operator keyword.
- Similar to defining a function you have a declaration of the operator overload in the header file and the definition of it in the implementation file.
- An example of a operator overload for the previous playdoh addition operation could look like:
  - `Playdoh& operator+=(int w);`
  - Notice it looks like a function where the syntax can be described as:
  - `returnType [operator][OP_SYMBOL](parameters);`

OPERATOR OVERLOADING

```
Playdoh& operator+=(int w);
returnType [operator][OP_SYMBOL](parameters);
```

- The signature of the operator overload is akin to a function signature and can be described in the following parts:
  - the operator keyword
  - the operation symbol
  - the type of its right operand, if any (the parameters)
  - the const status of the operation

OPERATOR OVERLOADING

```
Playdoh& operator+=(int w);

returnType [operator][OP_SYMBOL](parameters);
```

- Note again how the return type isn't part of the signature like with functions. If we wanted to differentiate these overloads we have to differ the signature (like functions)
- Another thing to note is that if the compiler can't find the exact signature it will try to match it with the closest one (by promotion or narrowing of arguments)

OPERATOR OVERLOADING

- The following operators can be overloaded:
- binary arithmetic (+ - * / %)
- assignment - simple and compound (= += -= *= /= %=)
- unary - pre-fix post-fix plus minus (++ -- + -)
- relational (== < > <= >= !=)
- logical (&& || !)
- insertion, extraction (<< >>)

OPERATOR OVERLOADING

- The following operators can't be overloaded:
- the scope resolution operator (::)
- the member selection operator (.)
- the member selection through pointer to member operator (.*)
- the conditional operator (?:)

OPERATOR OVERLOADING DESIGN

- When designing a operator overload, we should consider making it so that the expression denoted by the operator makes 'sense'
- In other words if we are doing an addition operator (+), it should probably do something in the vein of adding two values together
- We can overload a operator to do whatever we want it to do but we should try to make things logical in the sense of it aligning with other uses of the + operator

OPERATOR OVERLOADING EXAMPLES

BINARY OPERATOR OVERLOADS

- Binary operators act on two operands
- The left operand is usually the current object that we're trying to perform this operation on and the right operand is usually what we're trying to apply on the current object
- These overloads take the form: return_type operator symbol (type [identifier])

BINARY OPERATOR OVERLOADS

- Consider the binary operator +=
  - playdoh pl;
  - pl += 5
  - Left operand: pl Right operand: 5
- Let's add numbers to playdoh:

```
playdoh& playdoh::operator+=(int w){
 weight += w;
 return *this;
}
```

UNARY OPERATOR OVERLOADS

- Unary operators act only on a single operand (the left operand)
- Typically these overloads take the form of: return_type operator symbol()
- For the prefix operator it looks like: Type& operator++()
- For the postfix operator it looks like: Type& operator++(int)
- The left operand being the current object
- Consider the unary operators ++ prefix and ++ postfix
  - Eg. ++p; or p++;
  - Eg --p; or p--;

UNARY OPERATOR OVERLOADS

```
// Unary Operator ++ pre fix
playdoh& playdoh::operator++(){
  weight += 1;
  return *this;
}
```

- For the pre fix operator we want to increment prior
- Note the difference in return type compared to post fix

```
// Unary Operator ++ post fix
playdoh playdoh::operator++(int){
  playdoh temp = *this;
  ++(*this); // Notice the ++ here
  return temp; // Note returning the original
}
```

- For the post fix operator we want to increment afterwards!
- Note the difference in return type compared to prefix

TYPE CONVERSION OPERATOR

- Type conversion operators are used to define how we can implicitly convert types to other types
- When we do something like: `int x = false;`
  - The value of `false` is implicitly converted to a value of 0.

- For our user defined types can we also define this behavior
  - Let's try defining how we can define this sort of behavior for playdoh

TYPE CONVERSION OPERATOR

- Consider the operator conversion where we define what the meaning a bool conversion of a playdoh object is like. If the weight of a playdoh is > 0 then it returns true.

```
// bool operator conversion

playdoh::operator bool() const{

  return (weight > 0);

}
```

We can use the following operator like so: if (p1)… and we can now assess the boolean value of a user defined type

CAST OPERATOR

- Cast operators defines the meaning behind what occurs when casting a value as something else
- In this case let's consider this:
  - `int x = 55;`
  - `int y = 4;`
  - `double div = (double) x / y; // Compare this to doing double div = x / y;`
- For our playdoh example we can consider this:
  - `playdoh p1 = (playdoh) 12345;`
  - What could this behavior mean? Or what do we want it to mean?

CAST OPERATOR

Consider this cast operator. What does it do?

```
playdoh::playdoh(int w){
  setWeight(w);
  colour = 'y';
}

playdoh p1 = (playdoh) 12345;
```

WEEK 5-2

Helpers

HELPER FUNCTIONS

- Helper functions are assistive functions that typically support an existing class
- They however exist outside of the class itself and only accesses that class by having it be one of its parameters
- These functions are also typically global
- In essence these functions work with a class without being very tightly coupled to that class
- These helpers that are loosely coupled can be considered free helpers

FREE HELPER

- A potential free helper function for our playdoh class could be something like this:
  - bool samePlaydoh(Playdoh& a, Playdoh& b);

  - It is a global function existing outside of the Playdoh class but has parameters of Playdoh type.
  - The purpose of this function is to assess whether two playdoh are equal
  - A free helper doesn't have private access to the class.
  - This helper has to access its parameters through public member functions
  - Internally this function will make use of getter functions to query the state of the parameter objects to see if things are equal

FREE HELPERS BLOAT

- Due to lacking private access, there is a potential for increasing bloat when using free helpers
- As free helpers require public member functions to access the state of the object, if the object were to be updated with additional member data, additional query (getter) functions would be needed to support our existing helpers
- This increase of getter functions is considered class bloat

HELPER OPERATORS

- Returning to the idea of operators and operator overloads we can consider two categorizations of operators:
  - Member operators – These operators exist internal to the class as a member and can modify the state of the left operand (ie the current object)
    - Unary - ++ -- - + ! & *
    - Binary: = += -= *= /= %=
  - Helper operators – These operators exist external to the class as a global function and do not change any operands
    - + - * / % == != >= <= > < << >>

HELPER OPERATOR

- Consider the samePlaydoh function earlier:
  - bool samePlaydoh(Playdoh& a, Playdoh& b);
- It's purpose is to assess the equivalency of two playdoh objects
- It's a logical comparison. Do we not have an operator to test equivalence?

HELPER OPERATOR

- To have the same function become an operator overload, it may be like so:
  - `bool operator==(Playdoh& a, Playdoh& b)`
- This might make for a more readable and crisp function
  - Eg. if (p1 == p2)
- Notice the left operand is the first parameter and the right operand is the second parameter
- As helper operators exist outside a class the left operand isn't assume to be the current object

FRIENDS

- The friend keyword grants a helper function the friendship status to a class
- Friendship allows for that helper to gain private access to that class' members
- This can reduce the previously mentioned class bloat
- However granting friendship pierces the encapsulated nature of our class
  - Thus it's advised to only grant friendship when it's truly needed for a helper function to have it (functions that require both read and write access).
  - It may be more advised to stick with public member function queries despite the class bloat