Name: Soham Thaker SID: 011-748-159 Email: [sdthaker1@myseneca.ca](mailto:sdthaker1@myseneca.ca)

This workshop was all about overloading existing operators so that we can perform similar operations on compound types as we do on fundamental types. C++ allows us to overload all the existing operators except for few like, scope resolution, member selector, conditional operator & member selector through pointer to member operator. Also, it does not allow us to introduce or define new operators.

The Operators are classified into three categories:

1. Unary operator, which performs operations on one operand, example, post-fix & pre-fix increment/decrement operators, pre-fix plus and pre-fix minus.
2. Binary operators, which perform operations on two operands, example, assignment, arithmetic, relational, logical, etc.
3. Ternary operator, which performs operations on three operands, example, conditional operator

The overloading of operators is done in two ways:

1. Member operators – They become part of the class definition, where they can directly access the private attributes or private methods of that class
2. Helper operators- They are not a part of a class definition and they need public accessor methods to access the private attributes or private methods of a class or public mutator methods to mutate a objects’ state.

If the operator is to change the value of left operands, we make it a member operator, whereas helper operators are such that do not change the value of either the left or the right operand.

The signature to overload an operator would be, the operator keyword, followed by the operation symbol, the type of the right operand if any(depends if it’s a unary, if unary then it would not have any parameters, if binary it depends if it is a member or helper, if member then it would have an explicit and an implicit parameter, whereas if helper than it would have two explicit parameters) & the const status of the owner of the operator or the current object or the implicit parameter.

In this workshop we were to create & overload 4 types of operators which are, Binary member operator, Unary Type Conversion member operator, Helper Binary Operator between classes, Helper Binary Operator between primitive and a class. All four are briefly explained below with examples.

1. Binary Member Operator(from WS5 Lab):

Account& Account::operator=(Account& Ro) {}

This is an example of an Assignment Binary member operator that has an implicit parameter, which is current object and an explicit parameter which is of the same type as the current instance. It returns the reference of the current object since within this operator, we performed changes on this object where we assigned the values of Ro to \*this and changed the state of Ro too and thus to update those changes in the client code we need to return a reference of the current object. If the condition within the operator does not match, we simply return the state of the instance as is, without any changes. Adding const at the end of this operator’s signature would mean that the current object/owner of this operator would not be able change its state thus resulting in an error, and same goes with the right operand too.

Account& Account::operator=(const int acc) {}

This is an example of an Assignment Binary member operator that has an implicit parameter, which is current object and an explicit parameter which is of type int. Also, the explicit parameter is a const thus the logic within this operator cannot the value of variable acc. It returns the reference of the current object since within this operator, we performed changes on this object where we assigned the value of acc to \*this’ state and thus to update those changes in the client code we need to return a reference of the current object. If the condition within the operator does not match, we simply return the state of the instance as is, without any changes. Adding const at the end of this operator’s signature would mean that the current object/owner of this operator would not be able change its state thus resulting in an error, and same goes with the right operand too.

1. Unary Type Conversion member operator(from WS5 Lab):

bool Account::operator ~() const {}

Here the bitwise NOT operator is overloaded to return a Boolean value to be evaluated by the client code upon operator call. It has a constant status meaning the owner who calls this operator, its state cannot be changed. The bitwise NOT operator has only one operand that executes it logic upon, so it is considered to be a unary operator. Since this operator does not change the value of the operand, it should be a helper operator however the design of this functions calls for direct access to the state of the current object and thus we make it a member operator.

Account::operator double() const {}

This operator does a type conversion. We have overloaded the type conversion operator to cast a compound type into a primitive, double type. It is unary operator which does not change the state the owner and returns a value of type double.

Account::operator bool() const {}

This operator does a type conversion. We have overloaded the type conversion operator to cast a compound type into a primitive, Boolean type. It is unary operator which does not change the state the owner and returns a value of type bool.

1. Helper Binary Operator between classes(from WS5 Lab):

double operator+(const Account& Lo, const Account& Ro) {}

This is an example of an Addition Binary Helper Operator which operates on two explicit parameters that are of same type. It adds two member variables of two similar types and returns a sum as a type double. This is a helper since it does not change any of the operands and adding it as a member will give error about too many arguments, since binary member operators have two parameters, an implicit and an explicit, whereas here the specification of this operator tells us to declare both as explicit parameters. It returns a value of type double.

1. Helper Binary Operator between primitive type and a class(from WS5 Lab):

double operator+=(double& val, const Account& Ro) {}

This is an example of an Addition Assignment Binary Helper Operator which operates on two explicit parameters that are of different type. It adds two member variables of two different types and returns a sum of type double. This is a helper since its first parameter is of fundamental type whereas second being a compound type, meaning that implicit parameters concept does not exist on primitive types, so we cannot make it a member operator, it must be a helper otherwise we will get an error of too many arguments and changing the design to make the first parameter an implicit one, does not exist for primitive types. It returns a value of type double.