# Member Operator

An expression consists of an operator and a set of operands.

Overloading an operator entails declaring a corresponding function in the class definition and defining its logic in the implementation file.

**Operations**

The keyword **operator** identifies an overloaded operation.

The signature of a member function that overloads an operator consists of the keyword, the symbol and the type of the right operand (if any, within parenthesis).

| **Student& operator=(const Student&);** |
| --- |

* Binary arithmetic: + - \* / %
* Assignment (simple and compound): = += -= \*= /= %=
* Unary (prefix postfix plus minus): ++ -- + -
* Relational: == < > <= >= !=
* Logical: && || !
* Insertion, extraction: << >>

**Classifying Operators**

* **Unary (One operand):** Postfix increment/decrement, prefix increment/decrement, pre-fix plus, prefix minus
* **Binary (Two operand):** Assignment, compound assignment, arithmetic, relational, logical
* **Ternary (Three operands):** Conditional operator

**Member and Helpers**

We overload operators in either two ways, as:

* **Member Operators:** Part of the class definition with direct access to the class representation.
* **Helper Operators:** Supporting the class, without direct access to its representation.

Prefer to declare operators that change the state of their left operand as member operators.

#### Overloading a Member Operator

**Signature** of an overloaded operator consist of:

* The **operator** keyword
* The operation **symbol**
* The type of its right operand, if any
* The **const** status of the operation

The compiler binds an expression to the member function with the signature that matches the operator symbol, the operand type and the **const** statues

**Type of the Evaluated Expression**,  
The return type fo the member function declaration identifies the type of the evaluated expression.

**Binary Operators**

* A binary operator consists of **one operator and two operands**.
* The left operand is the current object and the member function takes one explicit parameter: the right operand.

| **return\_type operator symbol (type [identifier])** |
| --- |

| **Student& operator+=(float g);** |
| --- |

| **Student& Student::operator+=(float g) {  if (no != 0 && ng < NG && g >= 0.f && g <= 100.f)  grade[ng++] = g;  return \*this; }** |
| --- |

* **return\_type** is the type of the evaluated expression.
* **operator** identifies the function as an operation. symbol specifies the kind of operation.
* **type** is the type of the right operand.
* **identifier** is the right operand's name.

**Unary Operators**

* A unary operation consist of **one operator and one operand**.
* The left operand of a unary member operator is the current object.
* The operator does not take any implicit parameter

| **return\_type operator symbol()** |
| --- |

* **return\_type** is the type of the evaluated expression.
* **operator** identifies an operation.
* **symbol** identifies the kind of operation.

Pre-Fix Operators

* We overload the pre-fix increment/decrement operators to increment/decrement the current object and **return a reference to its updated value**.

| **Type& operator++() or Type& operator--()** |
| --- |

| **Student& operator++();** |
| --- |

0

| **Student& Student::operator++() {  for (int i = 0; i < ng; i++)  if (grade[i] < 99.0f)  grade[i] += 1.f;  return \*this; }** |
| --- |

Post-Fix Operators

* We overload the post-fix operators to increment/decrement the current object after returning its value. The header for a post-fix operator takes the form.

| **return\_type operator++(int) or Type operator--(int)** |
| --- |

| **Student operator++(int);** |
| --- |

* The **int** type in the header distinguish the post-fix operators from their pre-fix counterparts.

| **Student Student::operator++(int) {  Student s = \*this; //Save the original  ++(\*this); //calls th pre-fix operator  return s; //returns the original }** |
| --- |

Return Types

* The **return type of the pre-fix and post-fix operator differ**.
* The **post-fix operator returns a copy of the current object** as it was **before any changes** took effect.
* The **pre-fix operator returns a reference to the current object**, which access the **data after the changes have taken effect**.

**Type Conversion Operators**

* Type Conversion operators define implicit conversions to different types, including fundamental types.
* For the following code to compile, the compiler needs information on how to convert a Student object to a bool value:

| **Student harry;  if (harry)  harry.display();** |
| --- |

Bool **operator**

* Let us define a conversion operator that returns true if the Student object has valid data and false if the object is in a safe empty state.

| **operator bool() const;  Student::operator bool() const {  return no != 0; }** |
| --- |

**Cast Operator**

* C++ defines the casting operation for a class type in terms of a single-argument constructor.
* This overloaded constructor defines the rule for casting a value of its parameter type to the class type, as well as constructing an object from an argument of the parameter type.