Operator Overloading

C++ allows us to overload operators. Let's find out how.

WE'LL COVER THE FOLLOWING

Definition

C++ allows us to define the behavior of operators for our own data types. This is known as **operator overloading**.

Operators vary in nature and therefore, require different operands. The number of operands for a particular operator depends on:

1. the kind of operator (infix, prefix, etc.)

2. whether the operator is a method or function.

```
struct Account{
  Account& operator += (double b){
    balance += b;
    return *this; }....
};

...
Account a;
a += 100.0;
```

We have already encountered function overloading. If the function is inside a class, it must be declared as a `friend` and all its arguments must be provided.

Rules

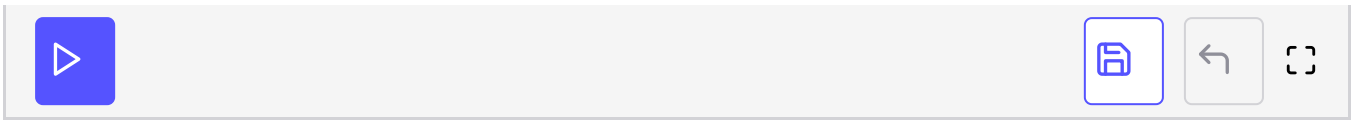
To achieve perfect operator overloading, there is a large set of rules we have to follow. Here are some of the important ones.

- **We cannot change the precedence of an operator**. The compiler computes all operators in order of precedence. We cannot alter this order. Hence, whatever operation our operator performs, it will be computed after the operator with higher precedence.
- **Derived classes inherit all the operators of their base classes except the assignment operator**. Each class needs to overload the `=` operator.
- **All operators other than the function call operator cannot have default arguments**.
- **Operators can be called explicitly**. A benefit of overloading an operator is that it can be used directly with its operands. However, the compiler may cause some implicit conversion in this process. We can make explicit calls to the overloaded operator in the following format: `a.operator += (b)`.

Example

```
#include <iostream>

class Account{

public:
  explicit Account(double b): balance(b){}

  Account& operator += (double b){
    balance += b;
    return * this;
  }

  friend Account& operator += (Account& a, Account& b);
  friend std::ostream& operator << (std::ostream& os, const Account& a);

private:
  double balance;
```

```
};


Account& operator += (Account& a, Account& b){
   a.balance += b.balance;
   return a;
 }

std::ostream& operator << (std::ostream& os, const Account& a){
   os << a.balance;
   return os;
}

int main(){

  std::cout << std::endl;

  Account acc1(100.0);
  Account acc2(100.0);
  Account acc3(100.0);

  acc1 += 50.0;
  acc1 += acc1;

  acc2 += 50.0;
  acc2 += acc2;

  acc3.operator += (50.0);
  //acc3.operator += (acc3);   ERROR

  std::cout << "acc1: " << acc1 << std::endl;
  std::cout << "acc2: " << acc2 << std::endl;

  std::cout << std::endl;

}
```



- The `+=` operator is being overloaded for the `Account` class in lines 8 to 10. Now, we can add and assign a `double` to an `Account` object without any problems. However, this will not support `Account += Account`, where both operands are of the `Account` type.

- For this purpose, we declare a friend function in line 13 to support the additional assignment between two `Account` objects. The friend function can access the private member, `balance`. Hence, we can perform `a.balance + b.balance`. The function has been implemented in lines 21 to 24.

- The `<<` output operator has also been overloaded in a friend function on lines 26 to 29. Now, `std::cout << acc;` will print the balance of the `acc`

object.

- From lines 39 to 43, we can see examples of the additional assignment working between the `Account` and `double` types.
- One thing to note is that the explicit operator call works when the argument is a `double`, as seen in line 45.
- However, the call would not work for the `Account` argument in line 46. This is because the class doesn't support the conversion from `Account` to `double`.

Prohibited operators

The following operators cannot be overloaded:

- `.`
- `::`
- `?:`
- `sizeof`
- `.*`
- `typeof`

Assignment operators

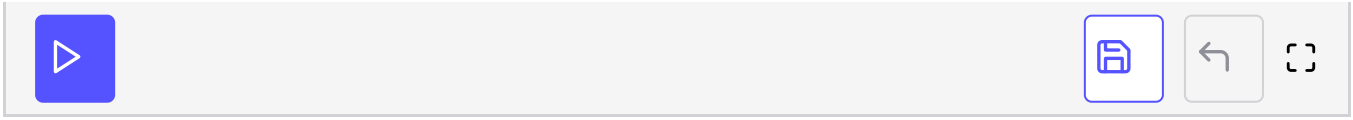
We can overload the assignment operator by implementing it as a **copy or move assignment** operator. It has to be implemented in a **class method**. The implementation is very similar to a copy or move constructor.

If the assignment operator is not overloaded, the compiler creates one implicitly. This operator performs a member-wise assignment of all the values from the object to be assigned. This is very similar to the behavior of the copy constructor, except that instead of a new object being created, the members of an existing object are updated.

Example

```
#include <algorithm>
#include <chrono>
#include <iomanip>
#include <iostream>
```

```
class Account{
public:

  Account()= default;
  Account(int numb): numberOf(numb), deposits(new double[numb]){}

  Account(const Account& other): numberOf(other.numberOf), deposits(new double[other.numberOf
    std::copy(other.deposits, other.deposits + other.numberOf, deposits);
  }

  Account& operator = (const Account& other){
    numberOf = other.numberOf;
    deposits = new double[other.numberOf];
    std::copy(other.deposits, other.deposits + other.numberOf, deposits);
  }

  Account(Account&& other):numberOf(other.numberOf), deposits(other.deposits){
    other.deposits = nullptr;
    other.numberOf = 0;
  }

  Account& operator =(Account&& other){
    numberOf = other.numberOf;
    deposits = other.deposits;
    other.deposits = nullptr;
    other.numberOf = 0;
  }

private:
  int numberOf;
  double * deposits;
};

int main(){

   std::cout << std::endl;
   std::cout << std::fixed << std::setprecision(10);

   Account account(200000000);
   Account account2(100000000);

   auto start = std::chrono::system_clock::now();
   account = account2;
   std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
   std::cout << "Account& operator = (const Account& other): " << dur.count() << " seconds" <

   start = std::chrono::system_clock::now();
   account = std::move(account2);
   dur = std::chrono::system_clock::now() - start;
   std::cout << "Account& operator=(Account&& other):" << dur.count() << " seconds" << std::e

   std::cout << std::endl;

}
```



- In this example, the assignment operator for the `Account` class is

- In this example, the assignment operator for the `Account` class is overloaded for both copy (line 16) and move (line 27) operations.
- If the argument is an lvalue, a copy is performed. A new array on the heap is created, called `deposits`, and the contents of `other`'s array is copied into it, as seen in lines 18 and 19.
- If the argument is an rvalue, a move is performed. In this case, a new array is not created. This makes the move operation much faster.
- This is evident in the `main` program.
- The `std::move` call on line 53 returns an rvalue, hence the assignment operator will move the data from `account2` to `account`.
- This is significantly more efficient than the copy assignment on line 48.

Further information

- function overloading

The next lesson deals with **explicit conversion operators**.