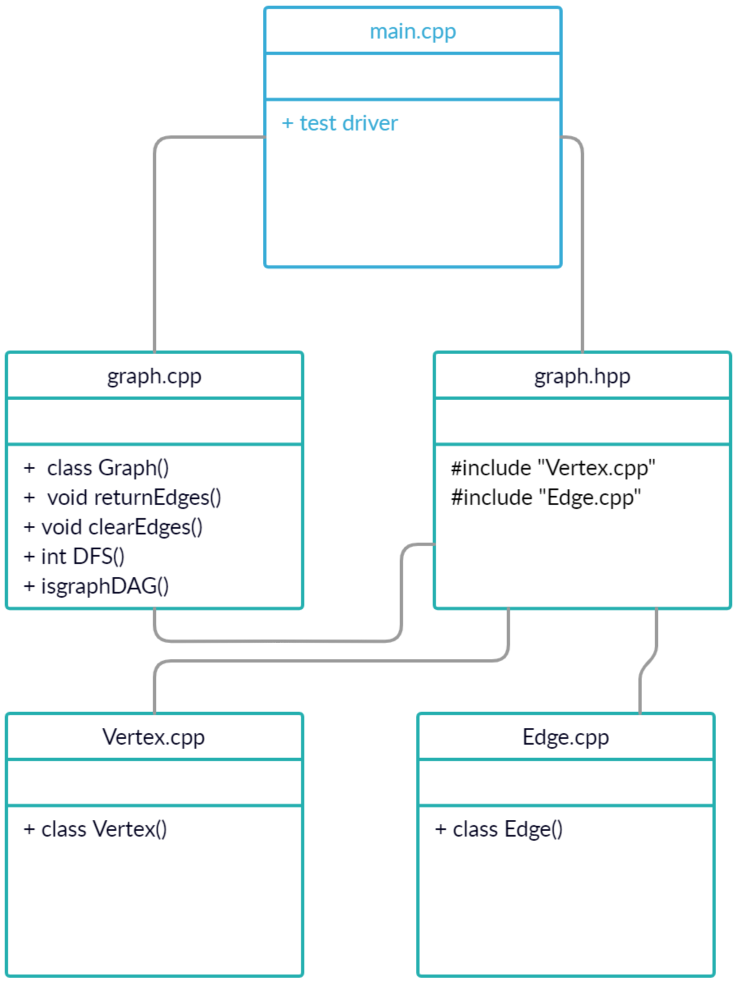
CPS2004 Assignment Report

Github url: *https://github.com/aidansey164/OOP_Assignment.git*
Last commit hash : *b479f5c8d6c1bd634051714f0f6495c84c60baae*

Task 1: DAG

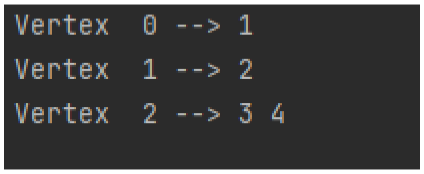
UML Diagram of Task 1



To implement the DAG, 3 main classes were used. The vertex class contains the parameterized constructor for vertices with the nodename and the default constructor. The edge class contains the parameterized constructor with the source and path for edges of vertices and the default constructor. The header file graph is used to link the vertex and edge classes to the graph class. The graph class contains all the functions and methods to build the DAG.

Firstly, the class Graph is initialized with the adjacency list, path nodes and source nodes. The graph constructor takes parameters, the list of edges and the number of vertices that will be included in the DAG. The adjacency list is used to declare the number of nodes in the DAG structure whilst also defining the Edge parameters.

The returnedges() function returns the structure of the directed acyclic graph. It loops through the adjacency list to print the DAG vertices and edges correctly.



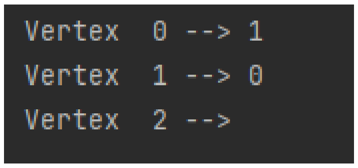
```
Vertex  0 --> 1
Vertex  1 --> 2
Vertex  2 --> 3 4
```

Output of the above function.

The DFS() function is used to traverse the DAG structure using depth first search algorithm. This was implemented to check and assure that the DAG has no cycles, hence its a properly defined DAG structure. First a variable was set to true for the first node of the DAG. Then a for loop loops throughout the adjacent list and checks if the algorithm has reached node a. Time complexity is incremented when leaving node n.

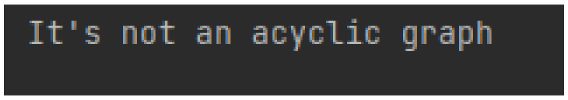
The isgraphDAG() function is linked to the previous function. The function checks if the graph is a DAG structure. Firstly, the function performs DFS traversal for all unvisited nodes in the graph. Then, it would check if the DAG is actually directed or not. It firstly checks if there is a back-edge between two different nodes. There exists a cycle and the graph cannot be directed, therefore false is returned. Otherwise the function returns true.

If the user defined a graph like this:



```
Vertex  0 --> 1
Vertex  1 --> 0
Vertex  2 -->
```

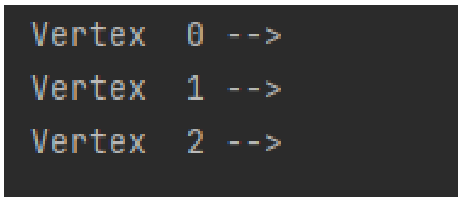
If isgraphDAG() returns false the function would return:



```
It's not an acyclic graph
```

Since there is a cycle between node 0 and 1, the graph inputted by the user is not a directed acyclic graph.

A limitation of the program is the implementation of removing edges from the directed acyclic graph; instead of removing one edge as instructed by the user, all edges defined are removed. This is achieved by clearing the whole adjacency list .clear().

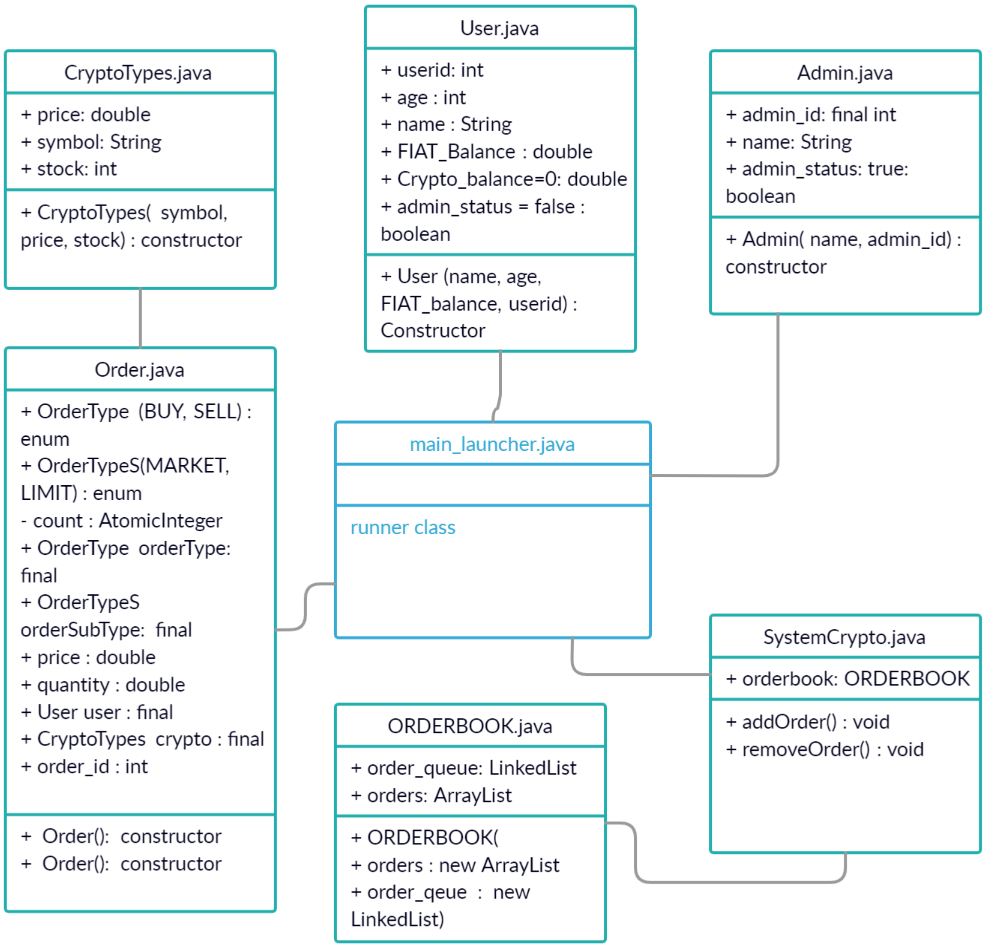


```
Vertex  0 -->
Vertex  1 -->
Vertex  2 -->
```

Output when clearing the adjacency list.

Task 2: Crypto Exchange

UML Diagram of Task 2



To implement the crypto exchange system in java, 6 classes were created and a main test driver class was also created to test the functions.

The class CryptoTypes is used to define a crypto type object constructor in the main_launcher class , defining the crypto's stock, price and symbol. Hence, the user can create many different types of cryptos in the system.

The class User is used to define a user that will be using the system. The user has attributes of userid, age, name, define his crypto balance and FIAT balance. The boolean admin status is set to false since the user should not access user only methods and functions. A constructor of type Admin is also defined in the class.

Similarly to the user class, the admin class is used to define a user object. An admin who has attributes of admin_id, name and where admin status is set to true. A constructor of type User is also defined within the class.

The class Order is used to define the order type. An order has attributes of OrderType either BUY or SELL and OrderTypeS which can be either a MARKET or LIMIT order. The count atomicinteger is used to keep a counter of the amount of orders made by one user. Each order will have a price, order_id, quantity and the cryptoType. Furthermore, the user making the order will be also included on the order, according to the userid as defined previously in the user object. Two order constructors are also defined within the class.

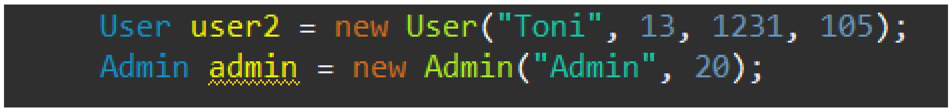
The class ORDERBOOK is used to define an order_queue of type linkedlist which holds a queue of the orders made by users. Another definition is of orders of the type of ArrayList that holds the history of order. Then new objects of orders and order_queue are created respectively.

The class SystemCrypto is where all the functions are found to process orders from users. The function add order is used to add an order to the linked list order_queue, whilst remove order is used to remove an order, if requested by a user.

Some limitations of the implementation include, a registration system of a user monitored by an admin, a system to process the different orders and a system to check if stock of a crypto is exceeded. Another system would be to sell orders of crypto as requested by a user.

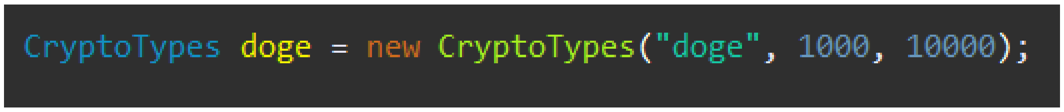
Technical solution for audit trails

To adhere with the audit trails and in order to not modify all classes and methods, a technical solution suitable would be to add a new variable order_id, a variable that accompanies each order made by the user. This way, one can identify every action that the user does by using the order_id to correspond to the order request made by the user. Using this technical solution, the deadline would be met and classes or methods would not be adjusted.



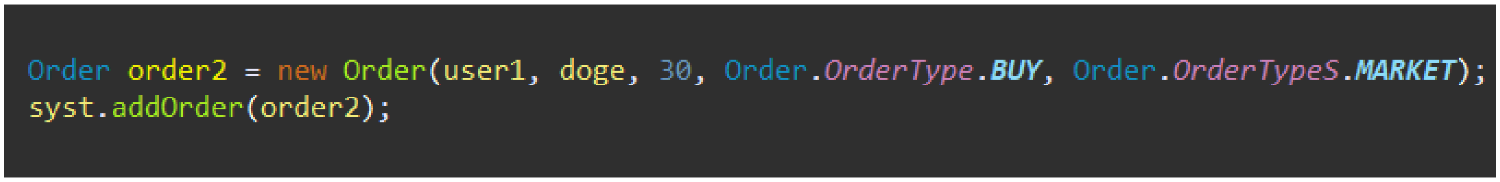
```
    User user2 = new User("Toni", 13, 1231, 105);
    Admin admin = new Admin("Admin", 20);
```

Declaring objects of type User and Admin respectively.



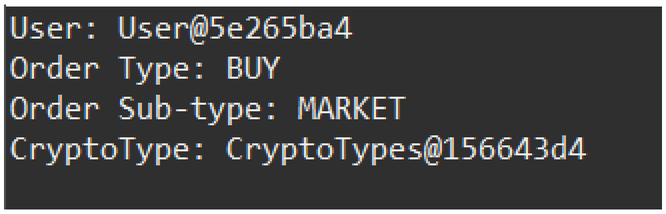
```
CryptoTypes doge = new CryptoTypes("doge", 1000, 10000);
```

Declaring object of type CrypoTypes.



```
Order order2 = new Order(user1, doge, 30, Order.OrderType.BUY, Order.OrderTypeS.MARKET);
syst.addOrder(order2);
```

User making a buy market order of crypto doge of 30 quantities. The order is then added to the order queue.

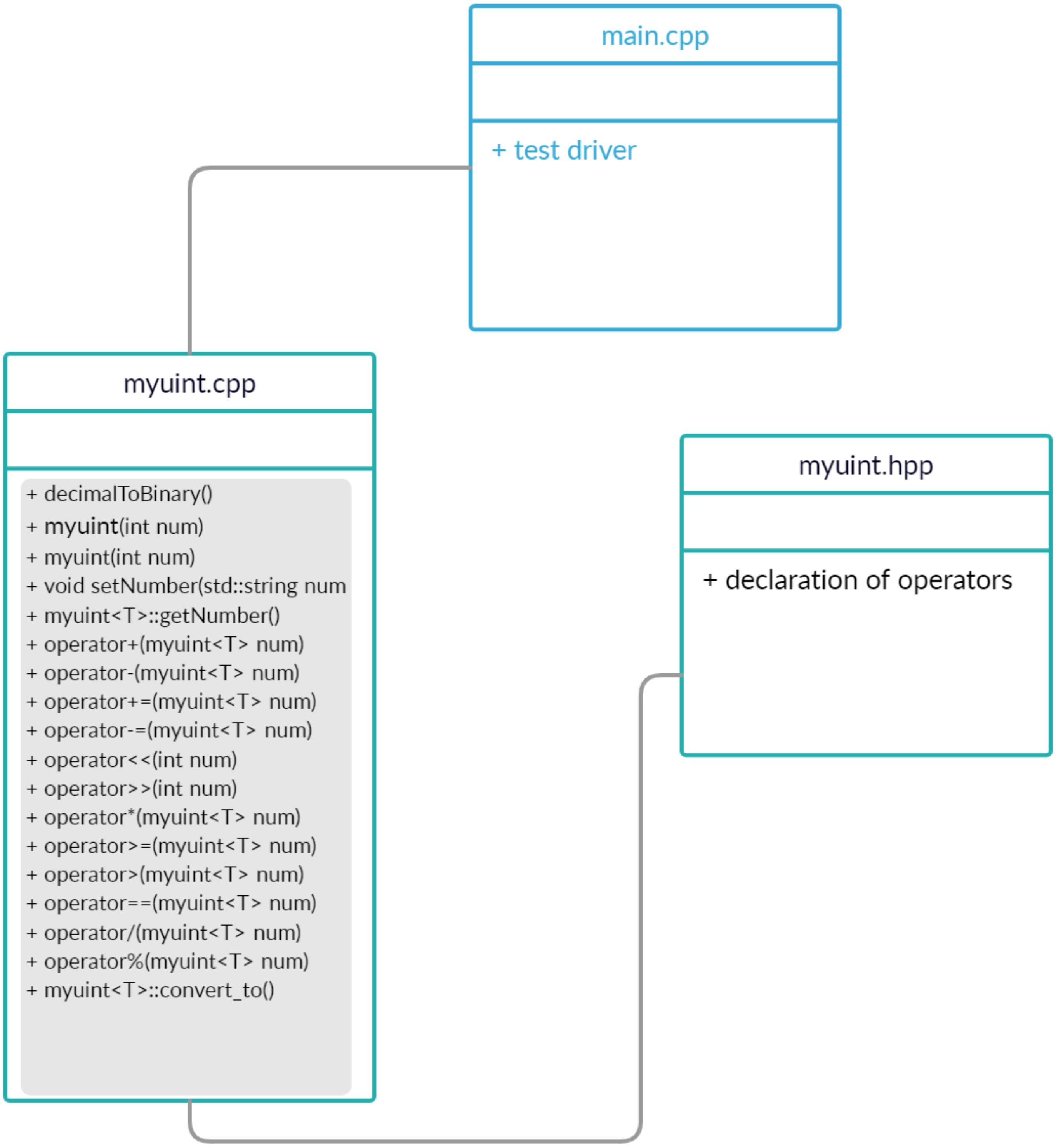


```
User: User@5e265ba4
Order Type: BUY
Order Sub-type: MARKET
CryptoType: CryptoTypes@156643d4
```

How an order looks like when outputted.

Task 3: Big integers library

UML Diagram of Task 3


main.cpp
+ test driver
myuint.cpp
+ decimalToBinary()
+ myuint(int num)
+ myuint(int num)
+ void setNumber(std::string num
+ myuint<T>::getNumber()
+ operator+(myuint<T> num)
+ operator-(myuint<T> num)
+ operator+=(myuint<T> num)
+ operator-=(myuint<T> num)
+ operator<<(int num)
+ operator>>(int num)
+ operator*(myuint<T> num)
+ operator>=(myuint<T> num)
+ operator>(myuint<T> num)
+ operator==(myuint<T> num)
+ operator/(myuint<T> num)
+ operator%(myuint<T> num)
+ myuint<T>::convert_to()
myuint.hpp
+ declaration of operators

To implement the big integers library, 2 classes and a header file were mainly used. The header file contains all of the declaration of operators that will be overloaded or constructed from the beginning. The myuint class contains all of the overloaded functions to implement the big integers library. The main class is used as a test driver to test all the operators.

In the header file myuint, firstly the convert_to(), getnumber(), setNumber() are defined and the myuint class is defined globally. Inside the class are defined all the operators that will be overloaded which are, +,-,+=,-=, <<, >>. The operators >=, > and == are defined in order to construct the division operator. Lastly, the *, % and / are also defined in the myunit class.

In myuint class all functions are defined. Firstly the function decimaltoBinary() takes as a parameter an integer, and converts this integer to a string number which will be used later overloaded in the myuint class constructor. A constructor for class myuint is created that takes in parameter an integer number. A try catch was implemented to assure that the correct number of bits would be present and an if function, if no bits are present which would print out an error message. If no conditions fail, then a number of type myuint is initialized. An empty constructor for myuint was also created with the checking conditions. The function setNumber() was created to set the integer value by overloading string num whilst the getNumber() function gets the integer from the user.

The function operator+() takes a num type of myuint, which is an overloaded string to overload the addition operator. Function declares the two numbers to be multiplied together and the resulting number, returning the result as type of myuint.

The function operator-() works similarly to the previous function, overloading the operator+() that is returned by the function.

The functions operator-=() and operator+=() are overloaded to cater for the corresponding operators.

The function operator<<(), overloads the left shift operator. Overloads a string value of type myuint. A for loops through the value and calculates the left shift of the value. The result is returned by the function. The function>>() works similarly to the previous function, taking an overloaded string value of type myuint and calculating/returning the right shift.

The function operator*() constructs the multiplication of two string values in which one is overloaded of type uint and the other is received by the user. Part of the answer is calculated and is pushed into a vector from the back. The other part of the answer is calculated and popped back to the other part of the answer. Ultimately the final answer is returned.

The function operator>() overloads the greater than operator that will be used to construct the division operator. Same with the function operator>=() and the function operator==(), are both overloaded with the respective operators in order to be used in constructing the division function.

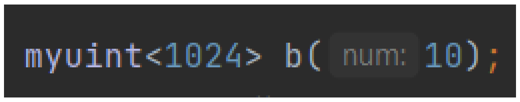
The function operator/() takes a num type of myuint, which is an overloaded string to overload the addition operator. Function returns a counter inside a while loop, which essentially checks how many times a number fits inside another number, incrementing the answer (i).

The function operator%() calculates the modulus of a number by subtracting the number by a variable which is set to 0 until the answer is no longer greater than or equal to the number inputted by the user. There is no counter as we do not care for the remainder of the number to calculate the modulus. The answer is finally returned by the user.

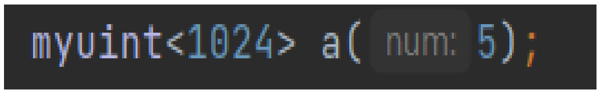
The function convert_to(), is used to convert from a type of myuint to any number type as specified by the user.

Testing of some of the operators

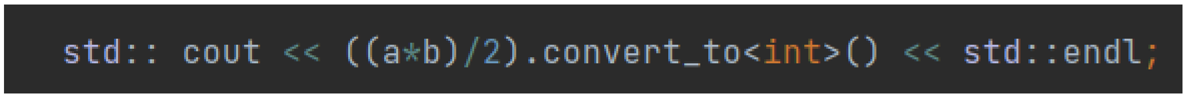
Multiplication and division



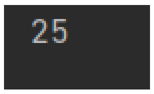
```
myuint<1024> b( num: 10);
```



```
myuint<1024> a( num: 5);
```

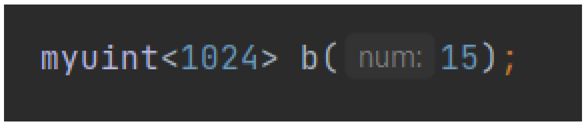


```
std:: cout << ((a*b)/2).convert_to<int>() << std::endl;
```

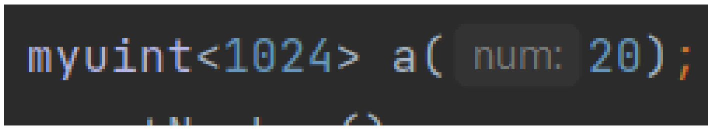
Output: 

```
25
```

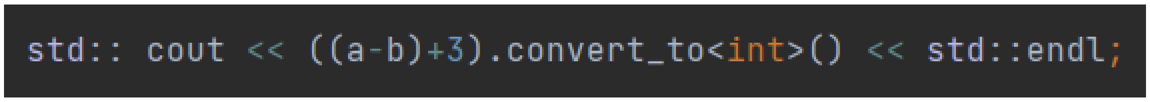
Addition and Subtraction



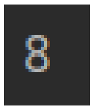
```
myuint<1024> b( num: 15);
```



```
myuint<1024> a( num: 20);
```

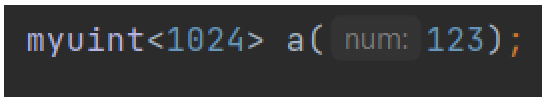


```
std:: cout << ((a-b)+3).convert_to<int>() << std::endl;
```

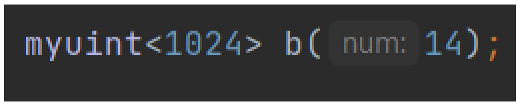
Output: 

```
8
```

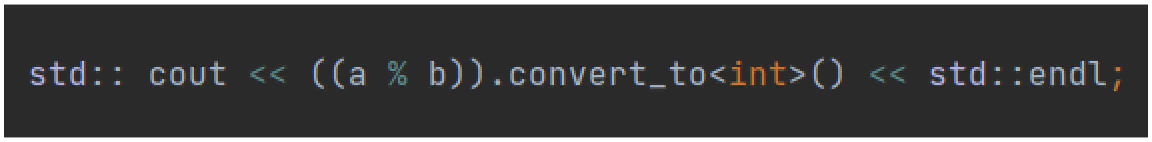
Modulus



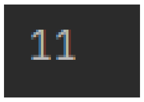
```
myuint<1024> a( num: 123);
```



```
myuint<1024> b( num: 14);
```

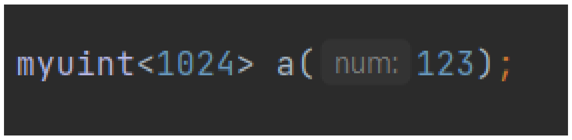


```
std:: cout << ((a % b)).convert_to<int>() << std::endl;
```

Output: 

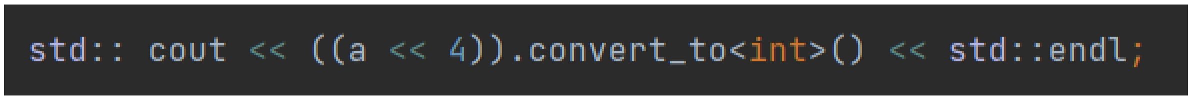
```
11
```

Left and right shift of 4 bits

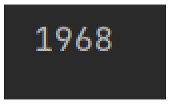


```
myuint<1024> a( num: 123);
```

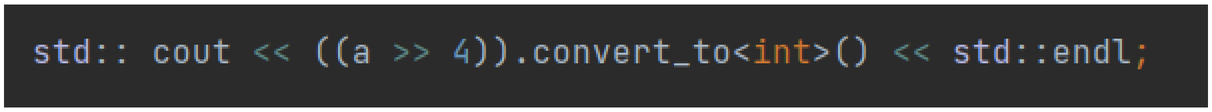
Output:



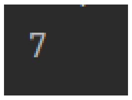
```
std:: cout << ((a << 4)).convert_to<int>() << std::endl;
```



```
1968
```



```
std:: cout << ((a >> 4)).convert_to<int>() << std::endl;
```



```
7
```