CPS2004 Report

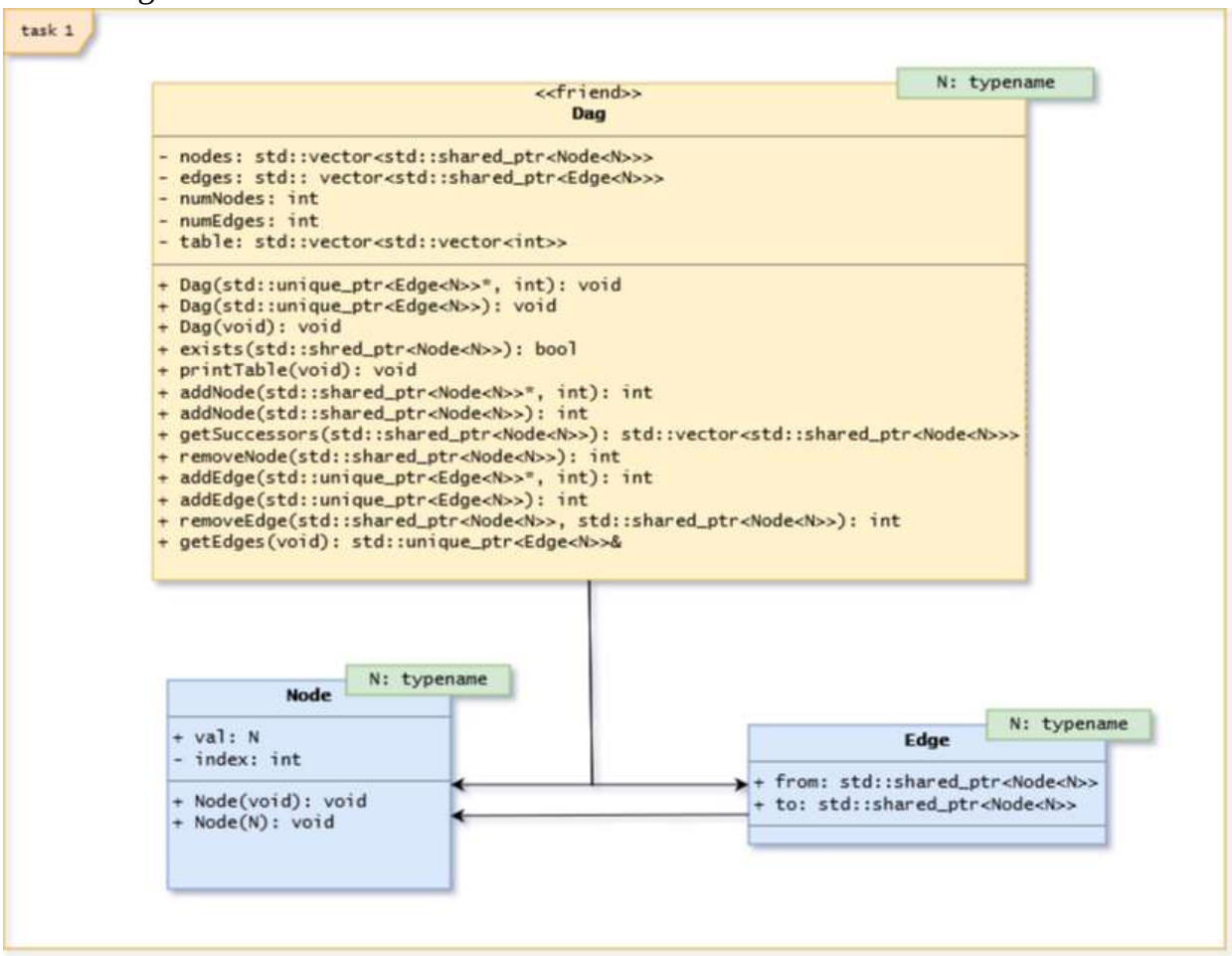
By Gabriel Hili

Contents

Running Tasks

Task 1

UML Diagram



Description

The Directed Acyclic Graph is implemented using 2 classes (Dag and Node) and 1 struct (Edge). Dag is a friend class to Edge and Node in order for the Dag Class to access private variables such as `val` and `index`. An edge consists of 2 nodes. The relationship of nodes and edges within the Directed Acyclic Graph is stored in an adjacency matrix. An element in the adjacency matrix is set to a non-zero value to represent an edge between 2 nodes.

Apart from the adjacency matrix, there also exists a vector of all nodes and a vector of all edges. The element in the adjacency matrix between 2 nodes is +1 of the index of the edge in this vector. Hence as an optimization when a node is trying to be removed, the row/column of the adjacency matrix will hold the index of all edges depending on that node.

Another optimization is in the function for checking cycles. If the user adds a list of edges, calling the cycle checking function for each one is slow and redundant. Hence the function is set to call only after the list of edges has been adding.

An instance of Dag can only be of a static type, but that type may be chosen through templates. Hence all instances of node and edges must match the template parameter of the corresponding Dag.

The user can manipulate the Dag class in many ways. The user can add Nodes, either singular or through a list. A list of edges can be passed to the Dag. An edge may refer a new node or a node already in the graph. In both cases the adjacency matrix is updated accordingly. **Removing a node will cause all edges dependent on that node to automatically be removed. Hence it is possible for the Dag Class to be composed of 2 disconnected subgraphs.**

The Dag accepts a list of unique pointer edges. Hence the edge ownership is safely moved to the Dag and the user cannot change any edges directly and must only interface with the Dag class to do so. The Dag accepts nodes in a shared pointer format.

The user can decide to instantiate a Dag with a list of edges, a singular edge or no edges at all, adding edges and nodes later on.

The Dag owns the individual edges by passing them as unique pointers. It has methods to remove individual nodes and edges. Since edges are unique pointers, edges are removed by passing in the from and to node to the function. Moreover the user can request a **reference** to the vector of edges in the Dag. However, the return type is set to `const` in order to prevent the user from directly altering the underlying data structure of the Dag without the proper checks and updates.

The Dag is declared as a friend class to the Node and Edge class. This is to prevent private members from being directly accessed from the launcher class but also allowing the Dag to manipulate its internal structure, such as accessing the index of a node.

Extra Features

- A formatted way to view the underlying adjacency matrix
- The user can initialize the Dag with a single edge, or no edges at all, instead of only a list of edge.
- Checking for cycles only happens when necessary and not for every added edge.
- Throws exception in case of cycle.

Test Cases

```
auto A = make_shared<Node<char>>('A');
auto D = make_shared<Node<char>>('D');
auto B = make_shared<Node<char>>('B');
auto C = make_shared<Node<char>>('C');
auto A_B = make_unique<Edge<char>>(Edge<char>{A, B});
auto B_C = make_unique<Edge<char>>(Edge<char>{B, C});
auto C_D = make_unique<Edge<char>>(Edge<char>{C, D});
auto C_A = make_unique<Edge<char>>(Edge<char>{C, A});
unique_ptr<Edge<char>> edges[2] = {move(A_B), move(B_C)};
Dag<char> dag(edges, sizeof(edges)/sizeof(edges[0]));
```

Test Case	Result
`assert(dag.addNode(A) == 3);`	**Match**
`assert(dag.addNode(B) == 3);`	**Match**
`assert(dag.addNode(C) == 3);`	**Match**
`assert(dag.addNode(D) == 4);`	**Match**
`assert(dag.getNumNodes() == 4);`	**Match**
`assert(dag.getSuccessors(A)[0] == B);`	**Match**
`assert(dag.getSuccessors(dag.getSuccessors(A)[0])[0] == C);`	**Match**
`assert(dag.getSuccessors(C).empty());`	**Match**
`assert(dag.getNumEdges() == 2);`	**Match**
`assert(dag.addEdge(move(C_D)) == 3);`	**Match**
`assert(dag.getNumEdges() == 3);`	**Match**
`assert(!dag.getSuccessors(C).empty());`	**Match**
`assert(dag.getSuccessors(C)[0] == D);`	**Match**
`assert(dag.getSuccessors(D).empty());`	**Match**
`assert(dag.removeNode(D) == 3);`	**Match**
`assert(dag.getNumEdges() == 2);`	**Match**

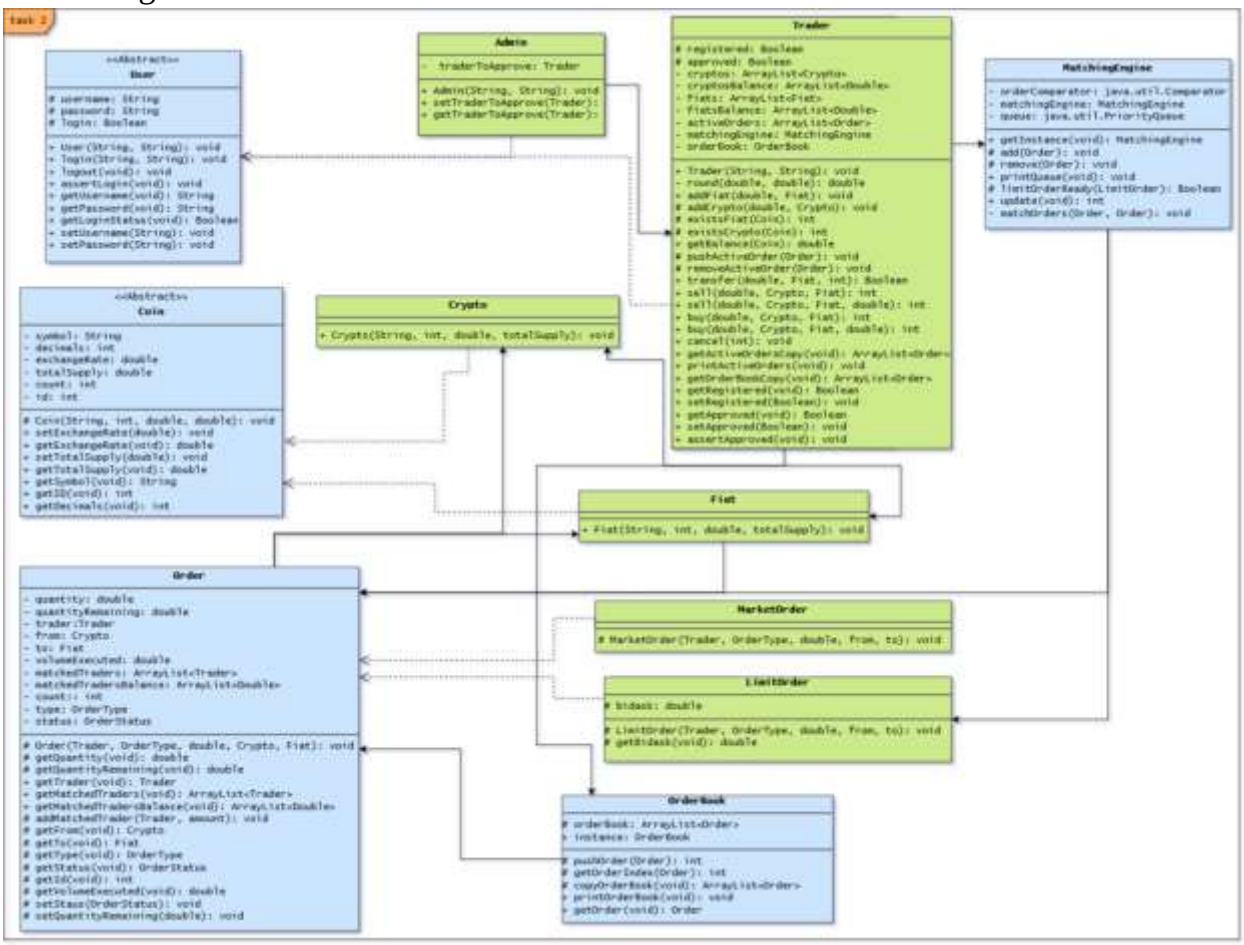
Critical Evaluation

The Dag was created to have the fastest look up time. Removing a node gives you the index of the edge to remove with it without performing a linear search through the vector of all edges. Moreover checking if an edge exists between 2 nodes is performed in O(1) time. The Dag has been optimized to have O(1) look up time. However this comes at a cost with respect to space complexity. The adjacency matrix has $O(n^2)$ best space complexity. Despite this, the choice was still preferable than using a vector of linked lists, which although would have better space complexity, would not have constant look up time.

Shared and Unique pointers were used to automatically allocate and delete their memory, as opposed to using new and delete. Unique pointers for edges met the constraint that the edges need to be owned by the Dag. Unique pointers were not used for nodes as multiple individual edges may reference the same singular node.

Task 2

UML Diagram



Description

The Crypto Exchange Platforms allows traders to place and cancel orders. For a trader to do so, he must be approved by an admin. This is done by first setting the `registered` flag to true. The trader instance is passed to an admin instance where the admin keeps a reference of the trader. The admin is then passed to the `approve` function in the Trader. If the admin's reference to the trader matches, then the `approved` flag is set to true. Admins and Traders are subclasses of User. For any user to perform actions, they must be logged in.

The trader may place buy/sell market/limit orders between a Fiat Currency and a Crypto Currency. Fiat and Crypto are subclasses of the Coin Class. A Coin holds information such as exchange rate, which is always set to Euros in each coin. Exchanging from one coin to another involves first exchanging to euros then from euros to the second coin. Before placing an order, the system checks whether the trader possesses enough coins to carry the order if it were to execute. Before executing the order, the balances of the 2 traders are checked again, as they might fluctuate between placement and execution.

Each Order a Trader places contains information such as the Traders which helped execute the order and the volume executed of the Order. Every Order contains a unique ID which may be used to get that Order

information from the OrderBook. The Trader can cancel any active order aslong as it hasn't been fulfilled. Partially filled Orders may be cancelled, but obviously the Trader cannot get back the already traded currency back. MarketOrder and LimitOrder are subclasses of Order. An instance of Order cannot be instantiated as its constructor is protected. Only MarketOrders and LimitOrders are allowed to be instantiated in the Launcher class.

As the order progresses the OrderBook is updated. The OrderBook makes use of an ArrayList of type Order to store all incoming Orders. Hence MarketOrders and LimitOrders are stored in the same list. A deep copy of this ArrayList may be obtained at any time by all users. The OrderBook class is a singleton instance, since all Orders should belong to the same OrderBook.

All non-filled orders are placed in a Priority Queue in the singleton MatchingEngine class. The Priority Queue is sorted by the quantity of an Order, buy, sell, market or limit. The Priority Queue is implemented using Java's inbuilt `java.util.PriorityQueue`. A static `java.util.Comparator` method is used to sort the Priority Queue by largest Quantity. When update is called the MatchingEngine attempts to find a pair of Buy and Sell Orders, which can be matched together. When a suitable pair is found, the Order's remaining quantity is updated, which updates the volume executed variable. When an Order's remaining quantity reaches 0 then the order is set to be fully filled, and removed from the Priority Queue.

Exceptions are promptly raised in case of illogical functionality, such as setting a negative quantity for an Order. Hence the Launcher file is required to catch these potential Exceptions and handle the flow of the program accordingly.

Extra Features

- All Users must log in first.
- A Trader can filter the OrderBook to view only his current active orders.
- A function exists to add Fiat Currency to a Trader Profile. The function takes a `bankid` variable. This is so a Bank API can be used within this function to actually transfer funds in a real world application.
- Order Quantities are rounded to the number of decimals the Coin has.
- Every Order returns an Order ID which may be used to find that Order in the OrderBook.
- Proper Exception Handling
- An Error class which can be implemented to take a particular action given a certain thrown Exception.
- The Crypto Exchange Platform is in a package.

Test Cases

Test Case	Result

`Trader trader = new Trader("trader","123");`
`Admin admin = new Admin("admin","456");`
 	
`assert(trader.getLoginStatus() == false);`
	Match

`trader.login("trader", "123");` `assert(trader.getLoginStatus() == true);`	Match
`assert(admin.getLoginStatus() == false);`	Match
`admin.login("admin", "456");` `assert(admin.getLoginStatus() == true);`	Match
`assert(trader.getRegistered() == false);`	Match
`trader.setRegistered(true);` `assert(trader.getRegistered() == true);`	Match
`assert(admin.getTraderToApprove() == null);`	Match
`admin.setTraderToApprove(trader);` `assert(admin.getTraderToApprove() == trader);`	Match

`assert(trader.getApproved() == false);`	Match
`trader.setApproved(true,admin);` `assert(trader.getApproved() == true);`	Match

`Trader joe = new Trader("joe","123");`
`joe.login("joe", "123");`
`joe.setRegistered(true);`
`admin.setTraderToApprove(joe);`
`joe.setApproved(true, admin);`
 	
`Fiat euro = new Fiat("€", 2, 1, 1000.0);`
`Crypto btc = new Crypto("$BTC", 8, 100, 1000.0);`
`MatchingEngine matchingEngine =`
`MatchingEngine.getInstance();`
`OrderBook orderBook = OrderBook.getInstance();`
 	
`trader.transfer(10,euro,123);`
`assert(trader.getBalance(euro) == 10);`
	Match

`joe._addCrypto(20, btc);` `assert(joe.getBalance(btc) == 20);`	Match
`int id1 = trader.buy(2,btc,euro);` `assert(orderBook.getOrder(id1).getId() == id1);`	Match
`int id2 = joe.sell(2,btc,euro);` `assert(orderBook.getOrder(id2).getId() == id2);`	Match
`assert(orderBook.getOrder(id1).getVolumeExecuted()` `== 0.0);`	Match
`assert(orderBook.getOrder(id2).getVolumeExecuted()` `== 0.0);`	Match

`matchingEngine.update();`
`assert(orderBook.getOrder(id1).getVolumeExecuted()`
`== 100.0);`
	Match

`assert(orderBook.getOrder(id2).getVolumeExecuted()` `== 100.0);`	Match

Critical Evaluation

My implementation makes use of object oriented features such as inheritance, polymorphism, encapsulation and abstraction in order to create the structure a real-world crypto exchange platform would use. The User and Coin class are abstract classes and both are super classes to the Admin and Trader classes, and the Fiat and Crypto classes respectively. This removes any repeated and redundant code which were to be present in the subclasses.

Moreover, variables protection levels are set accordingly. Many of the variables are set to protected and private. Protected variables are hidden from the Launcher file, but are still allowed access within the package for more complex functionality. All private variables, those worth accessing or not, have getter and setter functions. This is to allow further development to take places using these variables without implementing any extra functionality. Calling the setter for the private variable relating to an order's remaining quantity also recomputes the volume executed variable. Hence the remaining quantity cannot be changed without also changing the volume executed variable.

There also exists the Error class. The aim of the creation of the error class was to setup the foundations of an error handling system. A real world application could add code to the already existing class in order to catch exceptions and take appropriate actions. Due to the scope of this assignment, currently, handling an error only prints the message to the error stream and exits the program.

The Matching Engine and OrderBook class are implemented as singleton classes. This is to ensure that there only exists one instance of each across all orders. I preferred singleton classes over static ones in

order to allow these classes to be passed to other functions in future development, despite not being so right now.

Audit Trails

A Logger Class can be created. This singleton Logger class will keep a reference of any active trader in the platform. This can be done by adding a trader to the Logger in the trader's constructor. The Logger can check if any trader is logged in, and if so can periodically serialise the instance of the Trader class into a log alongside the current time, every fixed interval. Changes to the Trader profile will result in discrepancies between serialisations in the logs. Moreover the serialized profiles can be reinstantiated and the scenario taking place may be reconstructed.

Orders which a trader may place are always kept in the Order Book. The OrderBook may be modified to also keep track of the current time which an order was put. This then can be synced with a Trader's logs to see if he placed any orders.

The logs may be saved in the company's storage and encrypted, in order to avoid tampering.

Task 3

Description

The `myuint` class uses an array of bytes as the underlying data structure. The size is specified using a template parameter S. If the user wants to instantiate a 128-bit integer, than an array of 16 `uint8_t` elements is instantiated. A `static_assert` is used within the constructor to ensure the size is in the range [1, 2048] and a power of 2. AND, NOT and OR logic gates were fairly simple to implement as it consisted of just performing the gate on every bit. If the size is 64 or less, the bits are copied to a uint64_t integer, where the built in gate is directly applied, as can be seen in the following snippet.

```
if (S < 128 && U < 128) {
    uint64_t x = 0;
    uint64_t y = 0;
    memcpy(&x, data, get_size());
    memcpy(&y, rhs.data, rhs.get_size());
    x &= y;
    memcpy(data, &x, get_size());
    return *this;
}
```

Addition, subtraction, division, modulo and multiplication were all performed using the 3 aforementioned gates. Division and modulo implement the restoring division algorithm for unsigned integers and return the quotient and the remainder of the algorithm, respectively.

It is also possible to perform operations on built-in integers to myuint. Under the hood, a myuint class of 64 bits representing the built-in integer is created and used instead. Moreover all operators feature their assignment equivalent (eg. +=, -=, *=). As a matter of fact, the functionality of all operators is implemented in their assignment equivalent and regular operators simply call the assignment equivalent function, as can be seen in the following code snippet.

Comparism and equality were also implemented as well as post/pre fix increment/decrement. The `myuint` class also supports typecasting between different sizes/template parameters, were the `myuint` is either truncated or padded with 0s.

Consider the following scenario. The addition (+) operator is performed on 2048-bit version of a and b. All operators (apart from / & | ^ ~) typecast the operands to a 2048-bit version. This is to mimic built-in unsigned integers and also allow operations on `myuint` classes outside their specified range.

```
myuint<2> a = 2;

myuint<2> b = 3;

myuint<64> c = a + b; //5

myuint<2> d = a + b; //1 due to overflow/truncation

(a+b); //returns 5 implicitly despite being 2-bit numbers.
```

Extra Features

- Typecasting.
- Operator assignment (eg. +=).
- Implemented Move/Copy Constructors.
- Overloading comparism and equality operators.
- Using built-in integers as operands with `myuint` class.
- 2 example scripts apart from the launcher showcasing the `myuint` class.
- Functions to get/set a specific bit.
- Post/Pre-fix Increment/Decrement.
- Overloaded bool operator.
- Conversion constructor.
- Implicit return type of operators is not fixed to size of operands.
- Operands may have different sizes/template parameters.
- Ability to get/set a specific bit

Test Cases

```
myuint<128>  i(10);
myuint<2048> j(20);
myuint<16>   k(20);
```

Test Case	Result
i == 10	Match
j != 11	Match
j == 20	Match
j >= 20	Match
i <= 20	Match
i >= 5	Match
i <= 20	Match
j >= i	Match
j >= k	Match
k >= j	Match
(i|0) == i	Match
(i|1) == i+1	Match
(j|1) == 21	Match
(k|10) == 30	Match
(i|j|k) == 30)	Match
(k&1) == 0	Match
(i&i) == i	Match
(i&j) == 0	Match
(i&j&k) == 0	Match
(i&255) == i	Match
(~j) == ((j-j-1-j))	Match
(~k) == 65515	Match
(i^j) == 30	Match
(j^i) != 30	Match
(i+2093) == 2103	Match
(i+j+k) == 50	Match
(k+65535) == (k-1)	Match
(i-8) == 2	Match
((j-i)+10) == k	Match
(i+i-i+i-i+i-i) == i	Match
(i << 3) == 80	Match
(j << 1) == 40	Match
(k << 2) == 80	Match
(((k << k) >> k) +k) == (k+k-k)	Match
(i*j*k) == 4000	Match
(i*827) == 8270	Match
(j*k) == (i*2*j)	Match

`j/i) == 2`	Match
`(j/5) == 4`	Match
`(k/4) == ((j/5)+1)`	Match
`(k/4) == ((j/5)+1)`	Match
`(i%2) == 0`	Match
`((i+1)%2) == 1`	Match
`(j%k) == 0`	Match
`((j+i)%j) == i`	Match

Critical Evaluation

With regards to code size, although my implementation handles performing an operator on `myuint` with either an integer or a `myuint` instance of a different size, the operator is only implemented once and the various different variations simply cascade towards the function. Moreover, many operators are implemented using existing operators which further reduces code size. For example, addition is implemented using only the 3 basic logic gates and bit shifting.

```
inline myuint& operator += (const myuint& rhs) {
    //Addition is performed via a Full Adder on every bit
    uint8_t a = 0;
    uint8_t b = 0;
    uint8_t c = 0;

    for (uint16_t i = 0; i < S; i++) {
        a = get_bit(i);
        b = rhs.template get_bit(i);
        set_bit(i,a^b^c);
        c = (a&b)|(a&c)|(b&c);
    }


    return *this;
}
```

The `myuint` class has no additional overhead for storing the integers. An integer of 128 bits will take up exactly 128 bits in memory, in the form on 16 bytes stored in an array. To make the computation faster, during an operator, the bits of a `myuint` class are copied to a uint64_t integer, the operator is performed and the resulting bits are copied over to `myuint`. This only occurs when the size of the operands is less than 128. However the `myuint` class still proves to be slower when compared to built-in integers of the same size.

The move and copy assignments/constructors make sure follow best practices such as, realizing when the 2 objects are the same, deleting data from the heap before moving objects and setting freed pointers to `nullptr`.

In order to make the class more flexible in how it's used, operators are performed on a 2048-bit version of the operands so the return type will be able to be typecasted to instances of `myuint` with a bigger size

than the operands. Finally, since some operation depend on overflowing integers, I made sure to only use fixed size integers in order to be consistent across all systems.