COEN 79

Object-Oriented Programming and Advanced Data Structures

Practice questions for Quiz 4

1. Write the *pseudo-code* of an algorithm which evaluates a *fully parenthesized mathematical expression* using *stack* data structure. (calculator).

```
main() {
    while (there is an input character)
    {
        if (the input character is a digit || it is a decimal point) {
            push the number into the “numbers” stack;
        }
        else if (the input character is “+”, “-”, “*” or “/”) {
            push the number into the “operators” stack;
        }
        else if ( the input character is a “)” ) {
              evaluate the operation on top of the “operations” stack on the two numbers
               on top of the “numbers” stack;
        }
        else ignore the input;
    }
}
```

2. For the queue class given in Appendix 1 (cf. end of this assignment), implement the *copy constructor*.
   Note that the class uses a dynamic array. Also please note that you should not use the copy function (copy only the *valid entries* of one array to the new array).

```
template <class Item>
queue<Item>::queue (const queue <Item>& source)
{
    data = new value_type[source.capacity];  // Allocate an array

    capacity = source.capacity;

    count = source.count;
    first = source.first;
    last  = source.last;

    if (source.count != 0)  // Copy the elements from source to the newly created array
    {
        size_type tmpCount = count;
        size_type tmpCursor = first;

        while (tmpCount > 0) {
            data[tmpCursor] = source.data[tmpCursor];
            tmpCursor = (tmpCursor + 1) % capacity;
            --tmpCount;
        }
    }
}
```

3. For the queue class given in Appendix 1 (cf. end of this assignment), implement the following function, which increases the size of the dynamic array used to store items. Please note that you should not use the copy function (copy only the valid entries of one array to the new array).

```
template<class Item>
void queue<Item>::reserve (size_type new_capacity)
{
        value_type* larger_array;

        if (new_capacity == capacity) return;

        if (new_capacity < count)
            new_capacity = count;

        larger_array = new value_type[new_capacity];  // Allocate a new array


        if (count == 0) {
            first = 0;
            last = new_capacity - 1;
        }
        else  // Copy the elements from old array to the new array
        {
            size_type tmpCount = count;
            size_type new_last = new_capacity - 1;

            while (tmpCount > 0) {
                new_last = (new_last + 1) % new_capacity;
                larger_array[new_last] = data[first];
                first = (first + 1) % capacity;
                --tmpCount;
            }

            first = 0;
            last = new_last;
        }

        capacity = new_capacity;
        delete[] data;  // Delete the old array
        data = larger_array;
}
```

4. For the deque class given in Appendix 2 (cf. end of this assignment), implement the following constructor. The constructor allocates an array of block pointers and initializes all of its entries with NULL. The initial size of the array is `init_bp_array_size`.

```
template < class Item >
deque<Item>::deque( int init_bp_array_size, int init_block_size )
{
        bp_array_size = init_bp_array_size;
        block_size = init_block_size;

        //set a pointer the start of the array of block pointers
        block_pointers = new value_type* [bp_array_size];

        for (size_type index = 0; index < bp_array_size; ++index) {
            block_pointers[index] = NULL;
        }

        //set a pointer to the end of the array of block pointers
        block_pointers_end = block_pointers + (bp_array_size - 1);

        first_bp = last_bp = NULL;
        front_ptr = back_ptr = NULL;
}

```

5. For the deque class given in Appendix 2 (cf. end of this assignment), write the full implementation of the following function.

```
template <class Item>
void deque <Item>::pop_front()
// Precondition: There is at least one entry in the deque
// Postcondition: Removes an item from the front of the deque
{
    assert(!isEmpty());
    if (back_ptr  ==  front_ptr)
    {
        // This is the only element, clear the deque
        for (size_type index = 0; index < bp_array_size; ++index)
        {
            delete[] block_pointers[index];
            block_pointers[index] = NULL;
        }

        first_bp = last_bp = NULL;
        front_ptr = back_ptr = NULL;
    }

    // The front element is the last element of the first block
    else if (front_ptr == (( *first_bp)  +  block_size - 1))
    {
        delete[] *first_bp;
        *first_bp = NULL;
        ++first_bp;
        front_ptr = *first_bp;
    }

    else
    {
       ++front_ptr;
    }
}
```

6. What are the outputs of the following programs?

```
#include < iostream >
using namespace std;

class Base1 {
    public:
        ~Base1()  {
        cout << " Base1's destructor" << endl;  }
};
class Base2 {
    public:
        ~Base2()  {
        cout << " Base2's destructor" << endl;  }
};
class Derived: public Base1, public Base2 {
    public:
        ~Derived()  {
        cout << " Derived's destructor" << endl; }
};

int main() {
    Derived d;
    return 0;
}
```

Output: Destructors are always called in *reverse order of constructors*.

- `Derived's destructor`
- `Base2's destructor`
- `Base1's destructor`

```
#include < iostream >
using namespace std;

class Base {
    private:
        int i, j;
    public:
        Base (int _i = 0, int _j = 0): i(_i), j(_j) {}
};

class Derived: public Base {
    public:
        void show() { cout << " i = " << i << "  j = " << j;   }
};

int main(void) {
    Derived d;
    d.show();
    return 0;
}
```

Compiler Error: i and j are private in `Base`. We cannot access these variables inside `Derived`.

```cpp
#include < iostream >
using namespace std;

class P {
    public:
        void print()  {
        cout << " Inside P";
    }
};

class Q: public P {
    public:
        void print() {
        cout << " Inside Q";
    }
};

class R: public Q {};

int main(void) {
    R r;
    r.print();
    return 0;
}
```

Output:

```
Inside Q
```

The print function is not present in class R. So it is looked up in the inheritance hierarchy. `print()` is present in both classes P and Q, which of them should be called? If there is multilevel inheritance, then function is linearly searched up in the inheritance hierarchy until a matching function is found.

```cpp
#include < iostream >
using namespace std;

class Base {};

class Derived: public Base {};

int main() {
    Base * bp = new Derived;
    Derived * dp = new Base;
}
```

Compiler Error in line " `Derived *dp = new Base;`"
A Base class pointer/reference can point/refer to a derived class object, but the other way is not possible.

Appendix 1: queue class declaration:

```cpp
template < class Item >
class queue {
public:
    // TYPEDEFS and MEMBER CONSTANTS
    typedef std::size_t size_type;
    typedef Item value_type;

    static const size_type CAPACITY = 30;

    // CONSTRUCTOR and DESTRUCTOR
    queue(size_type initial_capacity = CAPACITY);
    queue(const queue& source);
    ~queue();

    // MODIFICATION MEMBER FUNCTIONS
    Item& front();
    void pop();
    void push(const Item & entry);
    void reserve(size_type new_capacity);

    // CONSTANT MEMBER FUNCTIONS
    bool empty() const {  return (count == 0);  }
    const Item & front() const;
    size_type size() const { return count;  }

private:
    Item* data;         // Circular array
    size_type first;     // Index of item at front of the queue
    size_type last;      // Index of item at rear of the queue
    size_type count;     // Total number of items in the queue
    size_type capacity;  // HELPER MEMBER FUNCTION

    size_type next_index(size_type i) const  { return (i + 1) % capacity; }
};
```

Appendix 2: deque class declaration:

```
template < class Item >
class deque {
public:
    // TYPEDEF
    static const size_t BLOCK_SIZE = 5; // Number of data items per block

    // Number of entries in the block of array pointers. The minimum acceptable value is 2
    static const size_t BLOCKPOINTER_ARRAY_SIZE = 5;

    typedef std::size_t size_type;
    typedef Item value_type;

    deque(int init_bp_array_size = BLOCKPOINTER_ARRAY_SIZE,
                                 int initi_block_size = BLOCK_SIZE);

    deque(const deque & source);
    ~deque();

    // CONST MEMBER FUNCTIONS
    bool isEmpty();
    value_type front();
    value_type back();

    // MODIFICATION MEMBER FUNCTIONS
    void operator = (const deque & source);
    void clear();
    void reserve();
    void push_front(const value_type & data);
    void push_back(const value_type & data);
    void pop_back();
    void pop_front();

private:
    // A pointer to the dynamic array of block pointers
    value_type** block_pointers;

    // A pointer to the final entry in the array of block pointers
    value_type** block_pointers_end;

    // A pointer to the first block pointer that's now being used
    value_type** first_bp;

    // A pointer to the last block pointer that's now being used
    value_type** last_bp;

    value_type* front_ptr; // A pointer to the front element of the whole deque
    value_type* back_ptr; // A pointer to the back element of the whole deque

    size_type bp_array_size; // Number of entries in the array of block pointers
    size_type block_size; // Number of entries in each block of items
};
```