Q1. Pointer & Function Review (15 min) Write a function with the following header:

```
void sum(int* list1, int list1_size, int* list2, int list2_size);
```

`list1` and `list2` are two integer arrays representing numbers
`list1_size` is the number of digits in the first number
`list2_size` is the number of digits in the second number

`sum` prints the sum of the numbers.

Example:

```
int list1[] = { 8, 5, 3, 1 };
int list2[] = { 5, 3, 2, 9 };
sum(list1, 4, list2, 4);
// prints 13860, the sum of 8531 and 5329

int list3[] = {    5, 3, 1 };
int list4[] = { 5, 3, 2, 9 };
sum(list3, 3, list4, 4);
// prints 5860, the sum of 531 and 5329
```

Solution

```
void sum(int* list1, int list1_size, int* list2, int list2_size)

    int num1 = 0;
    int place = 1;
    for (int i = list1_size - 1; i >= 0; i--)
    {
        num1 += (list1[i] * place);
        place *= 10;
    }

    int num2 = 0;
    place = 1;
    for (int j = list2_size - 1; j >= 0; j--)
    {
        num2 += (list2[j] * place);
        place *= 10;
    }
    cout << num1 + num2 << endl;
```

```
}
```

Q2. (15 min) Write a function with the following header:

```
bool rangeSearch(int sorted_nums[], int n, int target,
                 int& start, int& end);
```

`sorted_nums` is an array of integers sorted in decreasing order
`n` is the number of elements in `sorted_nums`
`target` is a number to search for within `sorted_nums`

`rangeSearch` should return *true* if `target` is found in `sorted_nums` and *false* otherwise.

If `rangeSearch` returns *true*, `start` should be set to the first index where `target` appears and `end` should be set to the last index where `target` appears, so that the integers of indices from `start` to `end` should only contain `target`.

If `rangeSearch` returns *false*, `start` and `end` should not be altered.

Example:

```
int foo[7] = { 5, 4, 3, 3, 1, -2, -3 };
int s = 21;
int e = 14;
rangeSearch(foo, 7, 0, s, e); // returns false, s remains 21, e remains 14
rangeSearch(foo, 7, 3, s, e); // returns true, now s == 2 and e == 3
```

Solution

```
bool rangeSearch(int sorted_nums[], int n, int target, int& start, int& end){
     int s = 0;
     int e = n-1;
     while(s<n && sorted_nums[s]>target)
         s++;
    while(e>=0 && sorted_nums[e]<target)
        e--;
    if(sorted_nums[s] == target && sorted_nums[e] == target){
         start = s;
         end = e;
         return true;
```

```
    }
    return false;
}
```

Q3. Classes (15 min) Write a class called Complex, which represents a complex number. Complex should have a default constructor and the following constructor: (Michelle Lee)

```
Complex(int real, int imaginary);
// -3 + 8i would be represented as Complex(-3, 8)
```

Additionally, the class should contain two member functions: sum and print. Calling sum should set the calling object to the sum of the 2 complex numbers passed as arguments. Print should print the complex number that the object represents. You may declare any public getters/setters or private data members that you deem necessary. Your code should work with the example below.

```
int main() {
(1)   Complex c1(5, 6);
(2)   Complex c2(-2, 4);
(3)   Complex* c3 = new Complex();

(4)   c1.print();
(5)   c2.print();
(6)   cout << "The sum of the two complex numbers is:" << endl;
(7)   c3->sum(c1, c2);
(8)   c3->print();
(9)   delete c3;
}

// The output of the main program:
5+6i
-2+4i
The sum of the two complex numbers is:
3+10i
```

Bonus: What would happen if we swapped line (8) and (9)?

Solution

```
class Complex {
```

```
    int m_real;
    int m_imaginary;
public:
    Complex() {}  // Bonus question: Is this the best choice of implementation?
    Complex(int real, int imaginary) {
        m_real = real;
        m_imaginary = imaginary;
    }
    void print() {
        cout << m_real << "+" << m_imaginary << "i" << endl;
    }
    void sum(Complex c1, Complex c2) {
        m_real = c1.m_real + c2.m_real;
        m_imaginary = c1.m_imaginary + c2.m_imaginary;
    }
};
```

Bonus: What would happen if swapped the order of (8) and (9)? How would it change the output?

Solution

After deleting the object pointed to by c3, an attempt to follow the pointer c3 is undefined behavior. The program might crash, print weird values (perhaps because the memory used by the deleted object was overwritten with some bookkeeping information the storage manager uses), print 3+10i (if the memory used was not overwritten), or do something else.

Q4.

a. What's the main difference between declaring a type with the keyword **`struct`** and declaring it with the keyword **`class`**?

Where can you use public members? Where can you use private members?

Solution

When using `struct`, until you specify otherwise, the compiler treats members as if you said `public:`, whereas for `class`, the assumption is `private:`. (default access level)

Public members can be accessed from anywhere the object is visible. These are typically used to define the interface of a class—methods and data that users

of the class are allowed to interact with.

Private members can be accessed only within the class itself. Private member functions are often used as helper functions for public methods. Since we don't want these functions to be called outside the class or used in isolation, we make them private to enforce encapsulation and prevent misuse.

Q5.

a. When should you write a destructor for a class?
b. (True/False) A class may have more than one destructor.
c. What happens if you forget to deallocate dynamically allocated memory once you're done with it?

A destructor should be used when an object owns resources that need to be released when the object's lifetime ends. A common example of this would be dynamically allocated memory.

False. Since a destructor cannot have parameters or a return type, it is impossible to create more than one destructor for a class.

If you forget to deallocate memory or release resources, it results in a memory leak. Memory leaks cause memory to remain allocated even though it is no longer needed, reducing the available memory for the program. Over time, especially in long-running programs, this can lead to increased memory usage, degraded performance, or even program failure due to running out of memory.

Q6.

a. If you have an object pointed to by a pointer, which operator is used with the pointer to access the object's members?

Solution

You can either dereference the pointer and access the member with the . (dot) operator *(*p).m* or *(*p).f(args)*, or more compactly use the -> (arrow) operator *p->m* or *p->f(args)*

*(*pointer).member* means the same thing as *pointer->member*

b. What does the following program output?

```
int main() {
    int a = 100, b = 30;
    cout << a + b << endl;                  // (1) 130

    int* ptr = &a;
    cout << *ptr + b << endl;               // (2) 130

    *ptr = 10;
    cout << *ptr + b << endl;               // (3) 40  (a = 10, b = 30)

    ptr = &b;
    *ptr = -12;
    cout << *ptr + 2*b << endl;             // (4) -12 + 2(-12) = -36

    int c = a + *ptr;
    cout << c << endl;                      // (5) -2

    b = -5;
    cout << a + b << endl;                  // (6) 5

    int arr[5] = {4, 5, 10, 11, -1};
    ptr = arr + 1;
    cout << *arr + *ptr << endl;            // (7) 9
    int cs;
    int& pic = cs;
    ptr = &pic;
    pic = 31;
    cs++;
    cout << *ptr << endl;                   // (8) 32
}
```

Solution

1. 130
2. 130
3. 40
4. -12+2*(-12) = -36
5. 10-12 = -2
6. 10-5 = 5
7. 4+5 = 9
8. 31+1 = 32