CS 7B - Spring 2019 - Final Exam Solutions

---

1. Write the number of the definition on the right next to the term it defines.

   (a) **copy** 3

   (b) **overload** 4

   (c) **container** 12

   (d) **pointer** 1

   (e) **reference** 8

   (f) **class** 5

   (g) **invariant** 10

   (h) **type** 9

   (i) **byte** 11

   (j) **constructor** 6

   (k) **scope** 7

   (l) **move** 2

   (1) (1) a value used to identify a typed object in memory; (2) a variable holding such a value.

   (2) an operation that transfers a value from one object to another, leaving behind a value representing "empty."

   (3) An operation that makes two objects have values that compare equal..

   (4) Define two functions or operators with the same name but different argument (operand) types.

   (5) A user-defined type that may contain data members, function members, and member types.

   (6) An operation that initializes an object. Typically establishes an invariant and often acquires resources needed for an object to be used (which are then typically released by a destructor).

   (7) The region of program text (source code) in which a name can be referred to.

   (8) (1) a value describing the location of a typed value in memory; (2) a variable holding such a value.

   (9) Something that defines a set of possible values and a set of operations for an object.

   (10) Something that must be always true at a given point (or points) of a program; typically used to describe the state (set of values) of an object or the state of a loop before entry into the repeated statement.

   (11) The basic unit of addressing in most computers.

   (12) An object that holds elements (other objects).

2. Write the output of the following C++ code, which uses several kinds of C++ parameter passing:

```
int mystery(int a, int* b, int& c) {
    a++;
    (*b)++;
    c++;
    return a;
}

int main() {
    int a{0}, b{0}, c{0}, d{0};
    mystery(a, &b, c);
    cout << a << " " << b << " " << c << " " << d << endl;
    mystery(c, &d, a);
    cout << a << " " << b << " " << c << " " << d << endl;
    c = mystery(b, &a, d);
    cout << a << " " << b << " " << c << " " << d << endl;
    a = mystery(a, &a, a);
    cout << a << " " << b << " " << c << " " << d << endl;
}
```

SOLN:
Variables `a, b, c,` and `d` are initialized to zero and then `a, b,` and `c` are passed to the function `mystery()`, with `a` passed by value, `&b` is passed by its address (which is kind of like being passed by reference, but `mystery()` then interprets it as a pointer variable) and `c` is passed by reference. So `mystery()` will leave `a` unchanged (it increments a copy of `a`, and that incremented copy is returned by `mystery()`, but the function call on line 10, does nothing with the returned value). So the value of `a` in the scope of `main()` is still 0 on line 11. On the other hand, `mystery()` dereferences the pointer to `b` and increments that value, so in the scope of `main()`, `b` is now 1, and since `c` is passed by reference and incremented in `mystery()`, `c` will have the value 1 on line 11. Nothing was done with `d`, so it's still 0 and the output from line 11 is "0 1 1 0".

The call to `mystery()` on line 12 passes `c=1` by value, which `mystery()` makes a local copy of (renamed `a`) increments to 2 and returns, but the call on line 12 does nothing with that returned value. The address of `d=0` is passed by value to a pointer to variable `mystery()` names `b`, which is dereferenced and incremented on line 3, so on line 13, the value of `d=1`. The variable `a` is passed by reference and renamed `c`, which is incremented so that on line 13, `a=1`. Since `b=1` was not affected by that, the output on line 13 is "1 1 1 1".

The call to `mystery()` on line 14 takes `b=1` by value, renames it `a`, increments that value and returns it, thereby assigning 2 to `c`. Meanwhile, the address of `a=1` is passed by value and renamed `b`, a pointer to an `int`, which is then dereferenced and incremented to 2, so on line 15, `b = 2`. Also, the call on line 14 passes `d=1` by references, renames is `c` and increments that value to 2, so that `d = 2`. The only variable unchanged then is `b=1`, which was passed in by value. So the output on line 15 is "2 1 2 2".

On line 16, the only thing that really matters is that `a=2` is passed in by value and the local copy is incremented and returned so that the value assigned to `a` is 3. This happens after `a` is incremented to 4 in the body of `mystery()`, so none of that matters and the output on line 17 is "3 1 2 2", giving the total output of

```
0 1 1 0
1 1 1 1
2 1 2 2
3 1 2 2
```

3. Consider the following complete program

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

int f(int);
int f(int num, vector<int>& memoizeF);
int64_t f(int64_t num, int64_t *memoizeF);

int main() {
    int num;
    int64_t num64{ 0 };
    vector<int> memoizeV{ 0,1 };
    // -1 indicates that the value is not within the array
    while (cin >> num) {
        int64_t* memoizeP = new int64_t[num + 1]{ 0, 1 };
        for (int i = 2; i <= num; i++) {
            memoizeV.push_back(-1); // -1 means not done
```

```
            memoizeP[i] = -1;
        }
        num64 = num; //promotion
        auto t1 = std::chrono::system_clock::now();
        cout << "The time to compute fibonacci(" << num << ") = "
            << f(num) << " with dumb recursion: ";
        auto t2 = std::chrono::system_clock::now();
        cout << (t2 - t1).count() << endl;
        t1 = std::chrono::system_clock::now();
        cout << "The time to compute fibonacci(" << num << ") = "
            << f(num, memoizeV) << " with memoization vector: ";
        t2 = std::chrono::system_clock::now();
        cout << (t2 - t1).count() << endl;
        t1 = std::chrono::system_clock::now();
        cout << "The time to compute fibonacci(" << num << ") = "
             << f(num64, memoizeP) << " with memoization pointer: ";
        t2 = std::chrono::system_clock::now();
        cout << (t2 - t1).count() << endl;
        delete[] memoizeP;
    }
}

int64_t f(int64_t num, int64_t* memoizeP) {
    if (memoizeP[num] == -1) {
        memoizeP[num] = f(num-1, memoizeP) + f(num-2, memoizeP);
    }
    return memoizeP[num];
}

int f(int num, vector<int>& memoizeV) {
    if (memoizeV[num] == -1)    {
        memoizeV[num] = f(num-1, memoizeV) + f(num-2, memoizeV);
    }
    return memoizeV[num];
}

int f(int n) {
    if (n == 0 || n == 1) return n;
    else return f(n - 1) + f(n - 2);
}
```

(a) Describe the parameters that distinguish the three overloaded versions of `fibonacci()`.
SOLN: The first one just takes a single `int`. The sond takes an `int` and reference to a `vector` of `int`–the third a 64-bit `int` and a pointer to a 64-bit `int`.

(b) Describe what happens on line 13.
SOLN: A `vector` of two `int`s is created and initialized to contain {`0,1`}.

(c) Describe, in detail, the effect of the command on line 16. **SOLN:** Enough contiguous memory is allocated on the heap to contain `num+1` 64-bit `int`s and the memory address of the first one is assigned to the variable `momoizeP`. Also, the first to values are initialized as 0 and 1.

(d) Why does line 16 need to be inside the `while`-loop but line 13 does not?
SOLN: To avoid a memory leak, the memory allocated for the `memoizeP` array is freed at the end of the `while` loop and so needs to be reallocated and initialized. Note that the same issue kind of exists for the `memoizeV vector`, except that it's not dynamically allocated (at least not in a way that's transparent to the coder), so we can just let the accumulated Fibonacci values take a ride for the next cycle.

(e) Describe what the containers `memoizeV` and `memoizeP` do that makes the computation of `f()` more efficient.
SOLN: Lines 17, 18 and 19 fill the next `num-2` elements of the vector `memoizeV` and the array `memoizeP` with the flag value "`-1`", to indicate that these Fibonacci values have not yet been computed, then when the recursive functions `f()` are called, a new Fibonacci value is computed from previous values if, and only if, the Fibonacci value has not yet been computed. Since this leaves many fewer function calls on the call stack, and since looking up stored results and returning the cached result when the same inputs occur again is much less expensive in both time and space than expensive function calls, memoization is a great optimization.

(f) A sampling of output for this program as the user enters 20,30,40 and 50 is shown below. Note the use of the `chrono` library to calculate the times of executions (measured in ticks).

```
20
The time to compute fibonacci(20) = 6765 with dumb recursion: 69501
The time to compute fibonacci(20) = 6765 with memoization vector: 46258
The time to compute fibonacci(20) = 6765 with memoization pointer: 40211

30
The time to compute fibonacci(30) = 832040 with dumb recursion: 974679
The time to compute fibonacci(30) = 832040 with memoization vector: 60426
The time to compute fibonacci(30) = 832040 with memoization pointer: 61168

40
The time to compute fibonacci(40) = 102334155 with dumb recursion: 121545885
The time to compute fibonacci(40) = 102334155 with memoization vector: 64019
The time to compute fibonacci(40) = 102334155 with memoization pointer: 68222

50
The time to compute fibonacci(50) = -298632863 with dumb recursion: 15006249361
The time to compute fibonacci(50) = -298632863 with memoization vector: 50684
The time to compute fibonacci(50) = 12586269025 with memoization pointer: 47437
```

Compare the growth in the time of computation for the three overloaded versions of `f()`.
SOLN: With memoization, the time does not seem to grow with the size of the input, first going up by about 50%, then more like 10%, and then, strangely...going down! Without the memoization, however, the ratios of time are $\frac{974679}{69501} \approx 1400\%$, $\frac{121545885}{974679} \approx 12500\%$ and $\frac{15006249361}{121545885} \approx 12500\%$. Those are huuuge leaps!

(g) Why do the "dumb recursion" and "memoization vector" versions give an incorrect value for `f(50)` but the "memoization point" version gives the correct value?
SOLN: That's a memory overflow error, since $F_{50} = 12586269025 \geq 2^{31}$, a 32-bit `int` can't contain the value.

4. Suppose the following C++ classes have been declared:

```
class C1 {
public:
    void m1() {
        cout << "C1 m1  ";
    }
    void m2() {
        cout << "C1 m2  ";
        m3();
    }
    void m3() {
      cout << "C1 m3  ";
      m1();
    }
};
```

```
class C2 : public C1 {
public:
    void m1() {
        cout << "C2 m1  ";
    }
    void m2() {
        C1::m3();
        cout << "C2 m2  ";
    }
    void m3() {
        cout << "C2 m3  ";
        m1();
    }
};
```

and that a client program declares the following variables:

```
C1* var1 = new C2();
C2* var2 = new C2();
```

Write the output that would be produced by each of the following calls, as it would appear on the console.

```
var1->m1();     C1 m1
var1->m2();     C1 m2
                C1 m3
                C1 m1
var1->m3();     C1 m3
                C1 m1
var2->m1();     C2 m1
var2->m2();     C1 m3
                C1 m1
                C2 m2
var2->m3();     C2 m3
                C2 m1
```

None of these are terribly surprising. What if, however we make `m3()` a virtual function by modifying it in both class definitions. In `C1`:

```
virtual void m3() {
    cout << "C1 m3  " << endl;;
    m1();
}
```

...and in `C2`:

```
virtual void m3() {
    cout << "C2 m3  " << endl;;
    m1();
}
```

Then, since `C2`'s definition of `m3()` overrides `C1`'s defintion, we get

`var1->m3();` `C2 m3`
`C2 m1`

5. Assume that an integer and a pointer each takes 4 bytes. Also, assume that there is no alignment in objects.

   Predict the output following program. Why?

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

class Test {
    int x;
    int *ptr;
    int y;
};

int main() {
    Test t;
    cout << sizeof(t) << " ";
    cout << sizeof(Test *);
}
```

SOLN: The output is
`12 4`
The "12" indicates that an object of type `Test` has three member variables, each one is an `int` requiring 4 bytes, so, in all, 12 bytes is the sizeof a variable of type `Test`. A pointer to a variable of type `Test` is the address, which on a 64-bit machine, takes 8 bytes. On a 32-bit machine, or with a compiler using a 32-bit platform, this would be "4."

6. Write a program to print the area and perimeter of a triangle having sides of 3, 4 and 5 units by

   (a) creating a class named '`Triangle`' with a function to print the area and perimeter.
   SOLN:

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

class Triangle
{
public:
    void print_area(int s1, int s2, int s3)     {
        double s = (s1+s2+s3)/2.0;
        cout << "Area is " << sqrt((s-s1)*(s-s2)*(s-s3)*s) << endl; //Heron's
    formula
        cout << "Perimeter is " << s << endl;
    }
};

int main() {
    Triangle t;
    t.print_area(3,4,5);
    return 0;
}
```

(b) creating a class named '`Triangle`' with the constructor having the three sides as its parameters.
SOLN:

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

class Triangle {
public:
    int s1,s2,s3;
    Triangle(int a,int b,int c) : s1(a), s2(b), s3(c)  { }
    void print_area()     {
        double s = (s1+s2+s3)/2.0;
        cout << "Area is " << sqrt((s-s1)*(s-s2)*(s-s3)*s) << endl; //Heron's
    formula
        cout << "Perimeter is " << s << endl;
    }
};
int main() {
    Triangle t(3,4,5);
    t.print_area();
    return 0;
}
```

7. Print the sum, difference and product of two complex numbers by creating a class named '`Complex`' with separate functions for each operation whose real and imaginary parts are entered by the user.
SOLN: This is pretty bare bones–your fractal project contained more robust code!

```
#include <iostream>
using namespace std;
class Complex {
private:
    int real;
    int imag;
public:
    Complex(int r, int i) real(r), imag(i) {  }
    int get_real() {
        return real;
    }
    int get_imag() {
        return imag;
    }
    void add(Complex c1) {
        cout << c1.get_real() + real << "+i" << c1.get_imag() + imag << endl;
    }
    void difference(Complex c1) {
        cout << real - c1.get_real() << "+i" << imag - c1.get_imag() << endl;
    }
    void multiply(Complex c1) {
        cout << ((real*c1.get_real()) - (imag*c1.get_imag())) << "+i" <<
            ((real*c1.get_imag()) + (imag*c1.get_real())) << endl;
    }
};
int main() {
    Complex c1(4, 5);
    Complex c2(2, 3);
    c1.add(c2);
    c1.difference(c2);
    c1.multiply(c2);
}
```