CISS245: Advanced Programming
Assignment 12

Objectives

- Overload and use `operator()`; function objects.
- Write and use static instance variables (i.e., static member variables).
- Write and use static methods (i.e., static member functions).
- Prevent object construction by clients by making constructors private.
- Include objects as instance variables.
- Create names with external linkages; `extern`.

This assignment involves several major ideas: `operator()`, static members and methods, and objects as instance variables. You will also prevent clients from constructing objects by making constructors private (this might seem bizarre, but you'll see why you want to do this by the end of this assignment).

Function objects and function call operator

A **function object** is simply an object that looks like a function, i.e., if `a` is a function object, then `a` is an object (so it's constructed from a class) and you can execute for instance `a(42)`, i.e., you can execute `a.operator()(42)`, i.e., `a` has a method of the form `operator()(int)`.

You can also choose to make the `operator()` accept two integers, i.e., `operator()(int, int)`. In that case you can execute `a(42, 0)` which will result in the invocation of `a.operator()(42, 0)`.

In general you can `a.operator()([some prototype])`. The operator `operator()` is sometimes called the **function call operator**.

Of course if your `a` is simply a function (as in a regular CISS240 function), then there's no reason to make it an object. You want to create `a` to be a function object only when `a` behaves like a function ... and more.

For instance what if you want `a` to return the last value it computed? Suppose the last function call operator executed was `a(42)` and the value returned was `999`, and you want to know what was the last computed value. Then you need to store that `999` in object `a`, and maybe have a method called `last_result`, so that you can call `a.last_result()` to return `999`. Maybe you also want `a.last_input()` which will return `42`.

And what if you want to compute the most frequently returned result? Then you probably want to keep as many values computed by `a` as you can, and allow the execution of `a.most_common_result()`.

One common use of function objects is to store computations to *avoid re-computations*. This is a very common theme in computer science, especially when certain computations are expensive (time-wise or memory-wise) and can result in dramatic computational performance improvement. The storing of expensive computations to avoid re-computation is called **memoization** – i.e., create a *memorandum* or *memo* of results. The memorandum is called the **memo table** or informally the **lookup table**.

CISS240 Review: Recursive Functions

The Fibonacci sequence is the sequence of integers 1,1,2,3,5,8,13,... It starts with 1,1 and from there on, a term in the sequence is the sum of the previous two terms. Mathematically you can think of this as a function `fib()`:

- `fib(0)` = 1 (this is by definition)
- `fib(1)` = 1 (this is by definition)
- `fib(2)` = 2 (i.e. `fib(1)` + `fib(0)`)
- `fib(3)` = 3 (i.e. `fib(2)` + `fib(1)`)
- `fib(4)` = 5 (i.e. `fib(3)` + `fib(2)`)
- `fib(5)` = 8 (i.e. `fib(4)` + `fib(3)`)
- `fib(6)` = 13 (i.e. `fib(5)` + `fib(4)`)

Self-Exercise: Complete the following table. You will need the table for your test code:

n	$fib(n)$
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	

The following function `fib(n)` will return the n-th value of the Fibonacci sequence:

```
int fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

(For simplicity, we will return 1 whenever the argument is negative.)

Test this function by printing out `fib(n)` for a few values of n. Compare against the table.

CISS240 Review: Tracing a Recursive Function

How does our C/C++ function work? Let's do an example. Suppose you call `fib(3)`:

```
std::cout << fib(3);
```

The execution of `fib(3)` will call `fib(3 - 1)` i.e. `fib(2)` because of

```
running fib(3):
    ...
    returning fib(n - 1) + fib(n - 2);
```

Now the execution of `fib(2)` will call `fib(2 - 1)` i.e. `fib(1)` because of

```
running fib(2):
    ...
    returning fib(n - 1) + fib(n - 2);
```

The execution of `fib(1)` will execute this:

```
running fib(1):
    ...
    return 1;
```

On returning to `fib(2)` we have the value of `fib(n - 1)`, i.e. 1, and we still have to call `fib(n - 2)` i.e. `fib(0)`

```
running fib(2):
    ...
    returning fib(n - 1) + fib(n - 2);
```

This call will execute this:

```
running fib(0):
    ...
    return 1;
```

Returning to the execution of `fib(2)` with `fib(n - 1)` = 1 and `fib(n - 2)` = 1 we can now execute the addition in

```
    running fib(2):
        ...
        returning fib(n - 1) + fib(n - 2);
```

which will return 2 for the call for `fib(n - 1)` during the execution of `fib(3)` (did you even remember where that was??)

```
     running fib(3):
        ...
        returning fib(n - 1) + fib(n - 2);
```

which will result in the call to

```
    running fib(1):
        ...
        return 1;
```

which returns 1 to `fib(n - 2)` in the execution of `fib(3)`. Now that all calls have returned (`fib(n - 1)` returned 2 and `fib(n - 2)` returned 1), we can execute the addition in

```
    running fib(3):
        ...
        return fib(n - 1) + fib(n - 2);
```

which will return 3.

Phew!! Basically you can see the calls if you use mathematical equations:

```
fib(3):

= fib(2) + fib(1)           fib(3) calls fib(2) and fib(1)
= fib(1) + fib(0) + fib(1)  fib(2) calls fib(1) and fib(0)
= 1 + 1 + fib(1)            fib(1) returns 1 to fib(2), fib(0) returns 1 to fib(2)
= 2 + fib(1)                fib(2) returns 2 to fib(3), fib(3) calls fib(1)
= 2 + 1                     fib(1) returns 1 to fib(2)
= 3                         fib(3) returns 3
```

Of course if I'm not interested in keeping track of the number of function calls and I'm only interested in the result I can write

```
fib(3)     = fib(2) + fib(1)
           = fib(1) + fib(0) + 1
           = 1 + 1 + 1
           = 3
```

There ... I've explained it twice. Now I want to do it **again** with a different picture. The arrows denote function calls. The number above the arrows show you the value returned:

```
             2                 1
fib(3) ----+---> fib(2) ----+---> fib(1)
           |                | 1
           | 1              +---> fib(0)
           +---> fib(1)
```

There you go ... three explanations. Note that there were two (repeated) function calls of `fib(1)`:

```
             3                 2                 1
fib(4) ----+---> fib(3) ----+---> fib(2) ----+---> fib(1)
           |                |                | 1
           |                | 1              +---> fib(0)
           |                +---> fib(1)
           |
           | 2               1
           +---> fib(2) ----+---> fib(1)
                            | 1
                            +---> fib(0)
```

Obviously `fib(4)` returns $3 + 2 = 5$. Now in this case note that `fib(4)` makes the following calls (directly or indirectly):

- `fib(3)` – 1 call
- `fib(2)` – 2 calls
- `fib(1)` – 3 calls
- `fib(0)` – 2 calls

Hey this is fun! (For me ...). Let's do `fib(5)`. I know that `fib(5)` will call

```
             3                2                1
fib(4) ----+---> fib(3) ----+---> fib(2) ----+---> fib(1)
           |                |                | 1
           |                | 1              +---> fib(0)
           |                +---> fib(1)
           |
           | 2                1
           +---> fib(2) ----+---> fib(1)
                            | 1
                            +---> fib(0)
```

and

```
             2                1
fib(3) ----+---> fib(2) ----+---> fib(1)
           |                | 1
           | 1              +---> fib(0)
           +---> fib(1)
```

So putting them together I get:

```
         4               3                2              1
fib(5)-+--->fib(4) ----+---> fib(3) ----+---> fib(2)---+---> fib(1)
       |               |                |              | 1
       |               |                | 1            +---> fib(0)
       |               |                +---> fib(1)
       |               |
       |               | 2                1
       |               +---> fib(2) ----+---> fib(1)
       |                                | 1
       |                                +---> fib(0)
       |
       | 3               2                1
       +--->fib(3) ----+---> fib(2) ----+---> fib(1)
                       |                | 1
                       | 1              +---> fib(0)
                       +---> fib(1)
```

(Such a diagram is called a function call tree.)

The calls from `fib(5)` are:

- `fib(4)` – 1 call
- `fib(3)` – 2 calls
- `fib(2)` – 3 calls
- `fib(1)` – 5 calls
- `fib(0)` – 3 calls

Obviously there is a lot of repeated calls and it will grow as you increase with larger `n` for `fib(n)`. These are small numbers. But how fast do they grow?? Well you can do the math, but let's get the program to tell us:

```
int fib(int n)
{
    std::cout << "fib(" << n << ") ...\n";
    if (n <= 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

Now call `fib(10)`. Since I'm too lazy (or too smart) to count, I'll get the program to count for me:

```
int count[10]; // global variable, accessible to all functions in this file

void reset()
{
    for (int i = 0; i < 10; ++i)
    {
        count[i] = 0;
    }
}

void print()
{
    for (int i = 0; i < 10; ++i)
    {
        std::cout << count[i] << " ";
    }
}

int sum()
```

```
{
    int s = 0;
    for (int i = 0; i < 10; ++i)
    {
        s += count[i];
    }
}

int fib(int n)
{
    // std::cout << "fib(" << n << ") ...\n";
    count[n]++;
    if (n <= 1)
    {
        return 1;
    }
    else
    {
        return fib(n-1) + fib(n-2);
    }
}
```

with `main()` as:

```
for (int i = 0; i < 10; ++i)
{
    reset();
    fib(i);
    print();
    std::cout << "sum:" << sum() << std::endl;
}
```

You see that the number of calls grow very quickly and many of the calls are actually repeated calls. How does `fib` perform "in the long run"? Try this for your `main()`:

```
for (int i = 0; i < 100; ++i)
{
    std::cout << fib(i) << "\n";
}
```

Are you convinced that the implementation of `fib` is really bad?

Q1. The Fib1 Class: `operator()` and Fibonacci Computations

The aim is to create a class whose objects behave like functions. In particular when we run the following program:

```
#include <iostream>
#include "Fib1.h"

int main()
{
    Fib1 fib1;
    for (int i = 0; i < 6; ++i)
    {
        std::cout << fib1(i) << std::endl;
    }

    return 0;
}
```

we have the same output as before.

Check your notes and textbook on `operator()`. If `obj` is an object, then

`obj(x)`	is the same as	`obj.operator()(x)`
`obj(x,y)`	is the same as	`obj.operator()(x,y)`
`obj(x,y,z)`	is the same as	`obj.operator()(x,y,z)`

The following test code must be included:

```
#include <iostream>
#include "Fib1.h"

int main()
{
    int correct[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
    std::cout << "Testing Fib1:" << std::endl;
    Fib1 fib1;

    for (int i = 0; i < 10; ++i)
    {
        std::cout << "Test " << i + 1
                  << (fib1(i) == correct[i] ? "pass" : "FAIL")
                  << std::endl;
    }

    return 0;
}
```

Test the speed of your `Fib1` class by printing out the Fibonacci numbers for i = 49. Is it fast??? (Yeah ...) How much time did it take? (No, don't answer). By the way, since an int is probably 32-bit, you will have an overflow and get a negative integer. Don't worry about that.

Q2. The Fib2 Class: Fibonacci Computations with Table Lookup

In this question you will develop another class for Fibonacci computation. The name of the class is `Fib2`.

If a function is used frequently, it makes sense to keep previously computed values. In your next Fibonacci computation class `Fib2`, each object should have an array, called `table`, of 20 integers to keep computed Fibonacci numbers. You should initialize all the integers in the array to -1, except for

```
table[0] = 1, table[1] = 1
```

Suppose you call `fib2(0)` where `fib2` is a `Fib2` object. Then `table` should be used. (You can also initialize `table` with `0` since the fibonacci numbers are all at least `1`.)

However suppose you call `fib2(2)`. You check that `table[2]` is -1 which means that this is the first time `fib2(2)` is being computed. So you use the formula `fib2(1) + fib2(2)`. This will call `fib2(1)` which will return 1 and `fib2(0)` which will return 1. So you get 2 for `fib2(1) + fib2(0)`. Before you return this value, you should set `table[2]` to 2.

In general each time `fib2(n)` is executed, the table should be used as a lookup table for previously computed values. Once `fib2(n)` is known, it should be kept in the object at `table[n]`, if `table[n]` is still -1.

So the pseudocode looks like this:

```
if n is negative return 1
if table[n] is -1, set table[n] to fib(n-1) + fib(n-2)
return table[n]
```

But wait. Note that the `table` has maximum size of 20 (so the maximum lookup is `table[19]`, i.e., `fib(19)`). So for the computation of `fib(25)` for instance you should not refer to the table when you need `fib(24)` and `fib(23)`. You should compute using the above formula, i.e. compute `fib(24) + fib(23)`. The pseudocode looks like this:

```
if n is negative return 1
else if n < 20
    // use the table
    if table[n] is -1
        set table[n] to fib(n - 1) + fib(n - 2)
    return table[n]
else
    // do not use the table
    return fib(n - 1) + fib(n - 2)
```

For testing purposes, you must also include `operator[]` so that if `fib2` is a `Fib2` object, then `fib2[4]` returns the value of `fib2.table[4]`.

Test your new and improved `Fib2`:

```
#include <iostream>
#include "Fib1.h"
#include "Fib2.h"

int main()
{
    int correct[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};

    ... test code for Fib1 ...

    std::cout << "Testing Fib2:" << std::endl;
    Fib2 fib2;
    std::cout << "Test 1 " << (fib2[2] == -1 ? "pass" : "FAIL")
              << std::endl;

    for (int i = 0; i < 10; ++i)
    {
        std::cout << "Test " << i + 2 << ' '
                  << (fib2(i) == correct[i] ? "pass" : "FAIL")
                  << std::endl;
    }

    std::cout << "Test 12 " << (fib2[2] == 2 ? "pass" : "FAIL")
              << std::endl;

    return 0;
}
```

To understand the improvement made in `Fib2`, perform the following experiment. Print `fib1(i)` for i = 0, ..., 39 and then print `fib2(i)` for i = 0, ..., 39. The second object should compute a lot faster.

Q3. The Fib3 Class: A Static Table Lookup

Obviously there's no reason for keeping different tables in different `Fib2` objects:

```
Fib2 fib2a;
std::cout << fib2a(5) << "\n"; // fib2a has a table

Fib2 fib2b;
std::cout << fib2b(5) << "\n"; // fib2b has a table too -- not cool!!!
```

Now:

- make the table member static so that they both share the same table. Run your test code to make sure that you haven't broken anything while making the changes. Think of a way of initializing `table`
- Provide a static method, `lookup()` so that `Fib3::lookup(n)` returns the value of `table[n]`.

Here's the test code:

```
#include <iostream>
#include "Fib1.h"
#include "Fib2.h"
#include "Fib3.h"

int main()
{
    int correct[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};

    ... test code for Fib1 ...
    ... test code for Fib2 ...

    std::cout << "Testing Fib3:" << std::endl;
    Fib3 fib3;
    std::cout << "Test 1 " << (Fib3::lookup(2) == -1 ? "pass" : "FAIL")
              << std::endl;

    for (int i = 0; i < 10; ++i)
    {
        std::cout << "Test " << i + 2 << ' '
                  << (fib3(i) == correct[i] ? "pass" : "FAIL")
                  << std::endl;
    }
    std::cout << "Test 12 "
              << (Fib3::lookup(2) == 2 ? "pass" : "FAIL") << std::endl;

    return 0;
}
```

Q4. The Fib4 Class: A Lookup Table with Variable Size

There's no reason why you should limit the size of the table member to size 20. Change your code so that you can specify the size of the table. The size of `table` is set when you create a `Fib4` object.

```
Fib4 fib4(30); // Now the static table is an array of 30 integers
```

For the above code, `fib4(30)` will point `table` to an array of integers of size 30. Use 20 for the default size. (Of course now `table` is an `int` pointer).

Note that your code must manage memory properly. For instance the following code tells you that it's possible to have two memory allocations for the static `table` member.

```
Fib4 fib4a(30); // static table is an array of 30 integers
Fib4 fib4b(40); // static table is an array of 40 integers and the
                // previous array of 30 integers are properly de-allocated
```

Hint:

- `table` should be initialized to `NULL`
- In the constructor, before allocating memory for `table`, the method should deallocate the memory `table` is pointing to if `table` is not `NULL`.

Besides allocating memory properly, previously computed values should be retained.

```
Fib4 fib4a(30); // static table is an array of 30 integers
Fib4 fib4b(40); // static table is an array of 40 integers and the
                // values in the previous array of 30 integers are
                // copied over to the new array of 40 integers before it
                // is de-allocated
```

The default constructor will allocate 20 integers for `table`:

```
Fib4 fib4c; // static table is an array of 20 integers
```

```
#include <iostream>
#include "Fib1.h"
#include "Fib2.h"
#include "Fib3.h"
#include "Fib4.h"

int main()
{
    int correct[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};

    ... test code for Fib1 ...
    ... test code for Fib2 ...
    ... test code for Fib3 ...

    std::cout << "Testing Fib4:" << std::endl;
    Fib4 fib4;
    std::cout << "Test 1 " << (Fib4::lookup(2) == -1 ? "pass" : "FAIL")
              << std::endl;

    for (int i = 0; i < 10; ++i)
    {
        std::cout << "Test " << i + 2 << ' '
                  << (fib4(i) == correct[i] ? "pass" : "FAIL")
                  << std::endl;
    }
    std::cout << "Test 12 "
              << (Fib4::lookup(2) == 2 ? "pass" : "FAIL") << std::endl;

    Fib4 fib4b(30);
    std::cout << "Test 13 "
              << (Fib4::lookup(2) == 2 ? "pass" : "FAIL") << std::endl;
    std::cout << "Test 14 "
              << (Fib4::lookup(25) == -1 ? "pass" : "FAIL") << std::endl;

    return 0;
}
```

Q5. The Fib5 class: Static Object in Fib5 Class and Private Constructors

It's annoying that we always have to create a Fibonacci object before using it. Furthermore ... do we really need to be able to create two Fibonacci objects?

Create a static object called `fib5` in `Fib5`. Initialize the lookup table with a size of 20. Next, since we want users to use the static `Fib5::fib5` object and not create their own objects, ... **for the first time** ... make the constructors **private** (both the default and the copy constructor). In other words make the following code impossible to compile:

```
Fib5 fib5;                  // won't compile ... yeah!
Fib5 fib5(Fib5::fib5);      // STILL won't compile ... :)
```

Include a static method to resize the size of the table. Call it `resize()` (duh). So `Fib5::fib5(5)` computes the 5 th Fibonacci number while `Fib5::resize(100)` resizes the table lookup to a size of 100.

(Note: Although you don't have to do this, you can use your `IntDynArr` class instead of `int*` for `table`. You also realize that earlier we made `table` static in order to prevent objects from having their own `table`. Now that the constructors are private, and there is only one public static `Fib5` object, this is not an issue anymore.)

```
#include <iostream>
#include "Fib1.h"
#include "Fib2.h"
#include "Fib3.h"
#include "Fib4.h"
#include "Fib5.h"

int main()
{
    int correct[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};

    ...test code for Fib1 ...
    ...test code for Fib2 ...
    ...test code for Fib3 ...
    ...test code for Fib4 ...

    std::cout << "Testing Fib5:" << std::endl;
    std::cout << "Test 1 " << (Fib5::lookup(2) == -1 ? "pass" : "FAIL")
              << std::endl;

    for (int i = 0; i < 10; ++i)
    {
        std::cout << "Test " << i + 2
                  << (Fib5::fib5(i) == correct[i] ? "pass" : "FAIL")
                  << std::endl;
    }
    std::cout << "Test 12 "
              << (Fib5::lookup(2) == 2 ? "pass" : "FAIL") << std::endl;

    Fib5::resize(30);
    std::cout << "Test 13 "
              << (Fib5::lookup(2) == 2 ? "pass" : "FAIL") << std::endl;
    std::cout << "Test 14 "
              << (Fib5::lookup(25) == -1 ? "pass" : "FAIL") << std::endl;

    return 0;
}
```

Q6. The Math header and cpp File

The next step in this assignment is written in two parts to make it easier for your to accomplish the task.

We now have a class `Fib5` and there's only one object in it called `Fib5::fib5`.

In the same manner, suppose you have another mathematical function that you want to write, say the factorial function. Does that mean we need to write `Factorial::factorial` if we use the same technique of a table lookup?

That's cumbersome and a little artificial. Why do I need to say **`Fib5` and `fib5`**? Or **`Factorial` and `factorial`**? How silly can this get ... ?!? ...

```
std::cout << Fib5::fib5(5) << "\n";
Fib5::fib5.resize(100);
std::cout << Factorial::factorial(6) << "\n";
std::cout << Log::log(7.4) << "\n";
```

Wouldn't it be more natural to say

```
std::cout << fib(5) << "\n";
fib.resize(100);
std::cout << factorial(6) << "\n";
std::cout << log(7.4) << "\n";
```

Do the following:

- In your `Fib5.cpp`, declare a `Fib5` reference (i.e. `Fib5 &`) called `fib` and initialize it to `Fib5::fib5`.
- To access `fib` from `Fib5`, in your cpp containing `main()` you add this statement: `extern Fib5 & fib;`. This tells your compiler that the identifier `fib` is in some cpp file. Of course `fib` is really the same as `Fib5::fib5`. At this point your should test that you can execute this in your `main()`: `std::cout << fib(7) << std::endl;`
- To make this even easier for users of your Fibonacci computations as well as other mathematical functions, you now move your external declarations into a header file. Create a header file called `Math.h`. Move your external declaration of `fib` to the file `Math.h`.

(Now ... keep this header file carefully. Build upon it by including all the useful things you might need. For instance you can declare a constant `PI` as the double 3.14159365, etc. in your `Math.cpp` and create an external declaration for `PI`, etc. in your `Math.h`.)

```
#include <iostream>
#include "Fib1.h"
#include "Fib2.h"
#include "Fib3.h"
#include "Fib4.h"
#include "Fib5.h"
#include "Math.h"

int main()
{
    int correct[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};

    ... test code for Fib1 ...
    ... test code for Fib2 ...
    ... test code for Fib3 ...
    ... test code for Fib4 ...
    ... test code for Fib5 ...

    std::cout << "Testing fib:" << std::endl;
    for (int i = 0; i < 10; ++i)
    {
        std::cout << "Test " << i + 1 << ' '
                  << (fib(i) == correct[i] ? "pass" : "FAIL")
                  << std::endl;
    }

    return 0;
}
```

Note that written in this fashion, the user does not need to know anything about any Fibonacci computation class. All of that is taken care of by `Math.h`.