CS 211 Final

Stable Marriage

Stable marriage is a problem solved through the use of backtracking. You are given an equal number of women and men, and your goal is to pair them in such a way that no pair prefers another pair of individuals to each other.

In this example, we define the preferences of the men using 2D arrays, where each row corresponds to an individual, and the column in that row represents their preference for the other individual.

```
    static int mp[3][3] = { {0, 2, 1},
                            {0, 2, 1},
                            {1, 2, 0} };
    static int wp[3][3] = { {2, 1, 0},
                            {0, 1, 2},
                            {2, 0, 1} };
```

Here their level of likeness towards another person is measured from 0-2 as there are 3 different possible individuals to match with (index 0, don't forget). For example, `mp[0]` is Man#0 , and they prefer Woman#0 as 0/2, meaning the most, then Women #2 as 1/2 meaning he perfers her the second most, and lastly Woman #1 the least at 2/2 (this is using index 0 for the women.) Here is a full breakdown:

```
-   Man#0: Prefers woman#0 the most, woman#2 next, and woman#1 the least.
-   Man#1: Prefers woman#0 the most, woman#2 next, and woman#1 the least.
-   Man#2: Prefers woman#1 the most, woman#2 next, and woman#0 the least.
```

```
-   Woman#0: Prefers man#2 the most, man#1 next, and man#0 the least.
-   Woman#1: Prefers man#0 the most, man#1 next, and man#2 the least.
-   Woman#2: Prefers man#2 the most, man#0 next, and man#1 the least.
```

We check for conflicts using the `ok` function, it verifies whether a match is valid or not, this is done by iterating over all the men who have been matched and checking two conditions:

- If the current man and another man are married to the same woman, it will be conflicitng so it returns an false. This is checked through `if (q[i] == q[col])`.

- If the current man and the other man's wife prefer each other over their other partners, it's also a conflit, so it returns false. Here is how the check is executed:
  - The first part `(mp[i][q[col]] < mp[i][q[i]] && wp[q[col]][i] < wp[q[col]][col])` checks if man i prefers the current man's wife over his own wife and if the current man's wife also prefers man i over her own husband.
  - The second part `(mp[col][q[i]] < mp[col][q[col]] && wp[q[i]][col] < wp[q[i]][i])` checks if the current man prefers man `i`'s wife over his own wife and if man `i`'s wife also prefers the current man over her own husband.

```
    for (int i = 0; i < col; i++) {
        // If man#i and man#col are married to the same woman OR
        // man#i and man#col's wife like each other more than their own spouse OR
        // man#col and man#i's wife like each other more than their own spouse,
        // then return false (conflict exists)
        if (q[i] == q[col] ||
            (mp[i][q[col]] < mp[i][q[i]] && wp[q[col]][i] < wp[q[col]][col]) ||
            (mp[col][q[i]] < mp[col][q[col]] && wp[q[i]][col] < wp[q[i]][i])) {
            return false;
        }
    }
```

`main` uses the backtracking algorithm to find the solutions. Array `q` is initialized at `{0,0,0}`, meaning that all men are married to woman with index 0.

```
    while (col >= 0) {
        q[col]++;
        if (q[col] == 3) {
            col--;
        } else if (ok(q, col)) {
            if (col == 2) {
                print(q);
                col--;
            } else {
                q[++col] = -1;
            }
        }
    }
```

Here is a more in-depth breakdown of the main function:

The function starts by initializing an array `q` and a column index `col`, where each index of the array `q` represents a man, and the value at that index represents the woman he is married to.

```
int q[3] = {}; // Array to store the current solution (the marriages)
int col = 0; // Column (man) index
```

The while loops begin until all men have been checked.

```
while(col >= 0){
        ...
}
```

The first course of action in this loop is moving on to check the next woman for the current man

```
q[col]++; // This moves to the next woman
```

Then we check if all of the women for the current man have been checked, if they have, we backtrack. This means all the women have been considered for the current man (this is since women are indexed from 0 to 2), and the algorithm reduces column by one, therefore, moving to the previous man.

```
if (q[col] == 3) {
    col--; // Backtrack
    }
```

We then check if the marriage is conflict free through the ok function - if not all women have been checked for the current man as seen by the condition before.

```
else if (ok(q, col)) {
    ...
}
```

This will leave us with two possible outcomes, either all men have been assigned a marriage that is conflicty-freee, this is when `col` is equal to 2 since men are indexed from (0,2). Then we would print the solution and backtrack to figure out if we can find any other solutions.

```
if (col == 2) {
    print(q); // Print the solution
    col--; // Backtrack to find other solutions
}
```

The other outcome is that we are not on the last man, so it would move on to the next man and intiialize his marriage at -1.

```
else {
    q[++col] = -1; // Move to the next man and initialize his marriage to -1
}
```

The loop will then continue until col is less than 0 which means all solutions were explored. It cannot backtrack further than the first man `col = 0`

Towers of Hanoi (Recursive and Iteratively)

The rules of this game are simple:

1. Only one disc can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disc may be placed on top of a smaller disk.

Iterative

Here is the breakdown of the code that will solve the problem and place all disks in Tower B, they are arranged (A, B, C)

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

int main() {
    vector<int> t[3];
    int n;
    cout << "Please enter the number of rings to move: ";
    cin >> n;
    cout << endl;


    // Decide on the starting value whether there is an odd or even number of disks
    int first_to = (n % 2 == 0) ? 2 : 1;

    // Declare initial values
    int from = 0, to = first_to, candidate = 1, move = 0;

    // Initialize the towers
    for(int i = n + 1; i >= 1; --i)
        t[0].push_back(i);
    t[1].push_back(n+1);
    t[2].push_back(n+1);
```

```
    while (t[1].size() < n+1) { // while t[1] does not contain all of the rings
        cout << "Move #" << ++move << ": Transfer ring " << candidate << " from
tower " << char(from+'A') << " to tower " << char(to+'A') << "\n";

        // Move the ring from the "from tower" to the "to tower" (first copy it,
then delete it from the "from tower")
        t[to].push_back(t[from].back());
        t[from].pop_back();

        // from = the index of the tower with the smallest ring that has not just
been moved: (to+1)%3 or (to+2)%3
        if (t[(to + 1) % 3].back() < t[(to + 2) % 3].back()) {
            from = (to + 1) % 3;
        }
        else {
            from = (to + 2) % 3;
        }

        // candidate = the ring on top of the t[from] tower
        candidate = t[from].back();

        // to = the index of the closest tower on which the candidate can be
placed: (from+1)%3 or (from+2)%3
        // (compare the candidate with the ring on the closer tower; which tower is
"closer" depends on whether n is odd or even)
        if (n % 2 == 0) {
            to = (t[(from + 2) % 3].back() > candidate) ? (from + 2) % 3 : (from +
1) % 3;
        } else {
            to = (t[(from + 1) % 3].back() > candidate) ? (from + 1) % 3 : (from +
2) % 3;
        }
    }
    return 0;
}
```

First we need to know the number of rings.

```
    int n;
    cout << "Please enter the number of rings to move: ";
    cin >> n;
    cout << endl;
```

Then we need to decide on a starting value, the first move depends on whether there are an odd or even number of disks. If even, then the first move is to the third tower, if odd the first move is to the second tower, **this will change depending on what tower we want to stack the rings (keep on reading, I will explain it in further detail)**.

We intialize all of the towers. The second and third towers are initialized with a disk of size n+1, which is larger than any of the actual disks. This serves as a sentinel value, representing an "empty" tower (since any disk can be placed on top of it).

```
    for(int i = n + 1; i >= 1; --i)
        t[0].push_back(i);
    t[1].push_back(n+1);
    t[2].push_back(n+1);
```

Here we break down the main function:

The while loop will check until all rings have been moved to the second tower (Tower B), which is also `t[1]`, don't forget, index 0. We also check it against n + 1 as the towers (B and C) have a base disk with size n+1 so we can stack any of our actual disks on top.

```
while (t[1].size() < n+1) {
...
}
```

We then print out the current Move#, the Tower it is being transferred from, and the Tower it is being transferred to. This is predetermined earlier based on whether the number of rings is odd or even, so the first move is already set. `char(from+'A')` has a `+A` to convert all of the indices into capital letters.

```
cout << "Move #" << ++move << ": Transfer ring " << candidate << " from tower " <<
char(from+'A') << " to tower " << char(to+'A') << "\n";
```

These two lines move the candidate ring from the "from" tower to the "to" tower. The `push_back` function places the ring on top of the target tower, and the `pop_back` function removes it from the original tower.

```
t[to].push_back(t[from].back());
t[from].pop_back();
```

These lines determine the new "from" tower for the next move. It should be the tower with the smallest ring on top, excluding the tower that we just moved a ring to. The modulus operation % 3 is used to wrap around the tower indices - that way we only check the 3 towers (0,1,2), as

any number above 3 is reduced to one of those 3 - modulus provides the remainder when divided by the number.

```
if (t[(to + 1) % 3].back() < t[(to + 2) % 3].back()) {
    from = (to + 1) % 3;
}
else {
    from = (to + 2) % 3;
}
```

This line updates the candidate ring to be moved next. It's always the ring on top of the "from" tower.

```
candidate = t[from].back();
```

These lines decide the "to" tower for the next move. The tower is chosen based on whether the number of rings `n` is even or odd. The new "to" tower must be the closest tower that the candidate ring can be placed on top of.

Again, the modulus operation % 3 is used to wrap around the tower indices so we don't go over, and the ternary operator ? : (a fancy if else) is used to choose between the two possible towers based on the size of the ring currently on top of them. If the ring on top of a potential "to" tower is larger than the candidate ring, the candidate ring can be placed on top of it; otherwise, it cannot.

```
if (n % 2 == 0) {
    to = (t[(from + 2) % 3].back() > candidate) ? (from + 2) % 3 : (from + 1) % 3;
} else {
    to = (t[(from + 1) % 3].back() > candidate) ? (from + 1) % 3 : (from + 2) % 3;
}
```

This wil lthen loop until all of the rings were moved to the target tower. Now what if I wanted all my rings in Tower C aka Tower 3. Then you would change the initialization to:

```
    int first_to = (n % 2 == 0) ? 1 : 2;
```

and change the while condition to check for the size of Tower C.

```
    while (t[2].size() < n+1) {
     ...
     }
```

Recursive

Solution (ripped from the internet):

```
using namespace std;

void towerOfHanoi(int n, char from_rod, char to_rod,
                  char aux_rod)
{
    if (n == 0) {
        return;
    }
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    cout << "Move disk " << n << " from rod " << from_rod
         << " to rod " << to_rod << endl;
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

// Driver code
int main()
{
    int N = 3;

    // A, B and C are names of rods
    towerOfHanoi(N, 'A', 'C', 'B');
    return 0;
}
```

Let's break down the main function:

```
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 0) {
        return;
    }
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    cout << "Move disk " << n << " from rod " << from_rod
         << " to rod " << to_rod << endl;
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}
```

These are the four arguments that the function accepts:

1. `n`: The number of disks.
2. `from_rod`: The rod currently holding the `n` disks.

3. `to_rod`: The rod to which we want to move the `n` disks.
4. `aux_rod`: The auxiliary rod we can use for temporary storage during the moves. So a tower to hold a ring so we can make a move.

Here is how it works:

1. `if (n == 0) { return;}`: This is the base case of the recursion. If there are no disks to move (i.e., `n` is 0), the function simply returns without doing anything.
2. `towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);`: This is the first recursive call. It moves `n-1` disks from the `from_rod` to the `aux_rod`, using `to_rod` as the auxiliary rod. After this call, the largest disk is left alone on the `from_rod`, and the remaining `n-1` disks are on the `aux_rod`, with `to_rod` empty.

Alright that's a lot to digest, so let's further break it down.

We are trying to move n-1 disks from the from_rod to the aux_rod. The function uses the to_rod as an auxiliary rod. You can think of this as trying to solve a smaller Towers of Hanoi problem, where we have n-1 disks instead of n disks. We ignore the largest disk for now and focus on moving the smaller n-1 disks.

Let's consider an example with n=3 disks, where the rods are 'A' (from_rod), 'B' (aux_rod), and 'C' (to_rod). At the start, all disks are on 'A', and we want to move them to 'C' using 'B' as an auxiliary. Here's the situation:

```
A: 3 2 1
B:
C:
```

The recursive call towerOfHanoi(n - 1, from_rod, aux_rod, to_rod); corresponds to towerOfHanoi(2, 'A', 'B', 'C'), which means we want to move 2 disks from 'A' to 'B', using 'C' as an auxiliary. This is the smaller Towers of Hanoi problem we're trying to solve first. After this call, the situation should look like this:

```
A: 3
B: 2 1
C:
```

This will continue until there are no more disks left, `n` = 0.

4. `cout << "Move disk " << n << " from rod " << from_rod << " to rod " << to_rod << endl;`: This line outputs the move that we are making. It's moving the `n`th disk (the largest one) from `from_rod` to `to_rod`.

5. `towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);` : This is the second recursive call. It moves the `n-1` disks from the `aux_rod` (where we moved them during the first recursive call) to the `to_rod`, over the largest disk. It uses `from_rod` as the auxiliary rod.

Feel free to send me edits for this, I yoinked it and did some modifications

Typedef

Definition
Typedef is a way to create an alias for an existing data type or complex type definitions. The primary use is to simplify type definitions so the code is easier to read and understand.

Syntax

```
typedef existing_type new_type_name;

// Example:
typedef int INTEGER;

// Here INTEGER would be an alias for the int type, so you can now declare an int variable as:

INTEGER a = 1;
```

Uses
Structs

```
typedef struct {
    int id;
    char name[50];
    float salary;
} Employee;

Employee e1;
```

Certainly! Typedef in C++ is used to create aliases for existing data types, making code more readable and providing an abstraction layer - "chat gpt ahhhh notes!💀" - daniel. Here are some examples of typedef usage in C++ along with explanations:

1. Alias for Primitive Data Types:

```
typedef int MyInt;
MyInt x = 10;
```

In this example, the `typedef` statement creates an alias `MyInt` for the `int` data type. The alias can be used interchangeably with `int`, providing a clearer and more descriptive name.

2. Alias for User-Defined Data Types:

```
typedef struct {
   int x;
   int y;
} Point;

Point p1 = { 10, 20 };
```

Here, the `typedef` statement is used to create an alias `Point` for a user-defined structure. This alias simplifies the declaration and usage of variables of that structure type.

3. Function Pointer Typedef:

```
typedef int (*MathFunction)(int, int);

int Add(int a, int b) { return a + b; }
int Subtract(int a, int b) { return a - b; }

MathFunction func = Add;
int result = func(5, 3);  // result = 8
```

In this example, `typedef` is used to create an alias `MathFunction` for a function pointer type. It allows for a more concise and intuitive declaration of function pointers, making the code easier to understand.

4. Typedef with Templates:

```
template<typename T>
struct MyContainer {
   typedef T value_type;
   // ...
};

MyContainer<int>::value_type x = 10;
```

This example demonstrates the usage of `typedef` within a template structure. It creates an alias `value_type` for the template parameter `T`. The alias provides a convenient way to access the underlying type used in the container.

5. Typedef Enum:

```
typedef enum {
   RED,
   GREEN,
   BLUE
} Color;

Color c = RED;
```

Here, `typedef` is used to create an alias `Color` for an enumeration. The alias simplifies the declaration and usage of variables of the enum type, enhancing code readability.

These examples showcase different use cases of `typedef` in C++, providing a way to define custom names for existing types, improve code clarity, and enhance code maintainability.

Pointers

Oof, this one here is a bitch, but lets get it. I am prone to making mistakes here, so please correct me if I do make any.

```
// Assume every pointer is 4 bytes
char* b[2][3]{

sizeof(b) = 24 // The array is 2x3 and each element takes up 4 bytes. So 2 x 3 x 4 = 24 bytes
sizeof(b+0) = 4 // This is the size of the address of the first row. Aka the size of the pointer b[0]

sizeof(*(b+0)) = 12 // This is the size of the first row of b. Each row has 3 columns, and each column has 4 bytes. 3 x 4 = 12

// Assume the address of b is 0034EF7D0

address of b+1: 0034EF7DC // This is the address of the second row, since each row has 3x4 bytes, the address for this would be 0034EF7D0 + 12 (decimal) -> 0034EF7D0 + C = 0034EF7DC

address of &b+1: 0034EF7E8 // &b is the address of the array itself and when adding 1 to this address, the result would be an address that is 1 whole array away from the starting address. 1 whole array in this case is 24 bytes. So 0034EF7D0 + 24 (decimal) -> 0034EF7D0 + 18 (hexadecimal) -> 0034EF7E8

}

// Different use cases of pointers, more specifically *
```

```
// Prints the product of all the elements in the array
int s(int *r, int n){
    if (n==1) {
        return *r;
    }
    else {
    /*This breaks down into the dereference of (r +n -1) which is then multiplied
      by the recursive call of s(r, n-1)
    */
        return (*(r+n-1)*s(r,n-1));
    }
};

int main(){
    int a[5]={1,2,3,4};
    cout<<s(a,4)<<endl;
    return 0;
}
```

Code to play around with and test pointer arithmetic:

```
// This is for a 64-bit system with 8 byte pointers.

#include <iostream>

int main() {
    char* b[2][3];

    // Size of the entire array: 2 rows * 3 columns * 8 bytes/pointer = 48 bytes
    std::cout << "sizeof(b): " << sizeof(b) << std::endl;  // Expected output: 48

    // Size of the first row: 3 columns * 8 bytes/pointer = 24 bytes
    std::cout << "sizeof(b[0]): " << sizeof(b[0]) << std::endl;  // Expected
output: 24

    // Size of a pointer: 8 bytes on a 64-bit system
    std::cout << "sizeof(b+0): " << sizeof(b+0) << std::endl;  // Expected output:
8

    // Size of the first element of the first row: 8 bytes on a 64-bit system
    std::cout << "sizeof(*(b+0)): " << sizeof(*(b+0)) << std::endl;  // Expected
output: 24

    // Size of a pointer: 8 bytes on a 64-bit system
    std::cout << "sizeof(b+1): " << sizeof(b+1) << std::endl;  // Expected output:
8
```

```
    // Size of the first element of the second row: 8 bytes on a 64-bit system
    std::cout << "sizeof(*(b+1)): " << sizeof(*(b+1)) << std::endl;  // Expected
output: 24

    // Size of a pointer: 8 bytes on a 64-bit system
    std::cout << "sizeof(*b+1): " << sizeof(*b+1) << std::endl;  // Expected
output: 8

    // Address of the array
    std::cout << "b: " << b << std::endl;

    // Address of the first row
    std::cout << "b[0]: " << b[0] << std::endl;

    // Address of the second row
    std::cout << "b+1: " << b+1 << std::endl;

    // Address of the second element of the first row
    std::cout << "b[0]+1: " << b[0]+1 << std::endl;

    // Address of the second row
    std::cout << "*(b+1): " << *(b+1) << std::endl;

    // Address of the third element of the second row
    std::cout << "*(b+1)+2: " << *(b+1)+2 << std::endl;

    // Address of the array
    std::cout << "&b: " << &b << std::endl;

    // Address immediately following the array
    std::cout << "&b+1: " << &b+1 << std::endl;

    return 0;
}
```

Detailed Pointer Rules (the examples below are using 4 byte pointers with address 0x100 for b)

- **Array Pointers and Sizeof Operator**: The `sizeof` operator gives the size of the type or object in bytes. When used with an array, it gives the total size of the array (number of elements * size of each element). When used with a pointer, it gives the size of the pointer (which is typically 4 bytes on a 32-bit system and 8 bytes on a 64-bit system).

```
char* b[2][3];
sizeof(b);  // Size of the entire array: 2 rows * 3 columns * size of pointer = 24 bytes
sizeof(b+0);  // Size of a pointer: 4 bytes
sizeof(*(b+0));  // Size of the first row: 3 columns * size of pointer = 12 bytes
```

- **Pointer Addition**: When you add an integer to a pointer, the pointer moves forward by that many elements of its type. In the case of a 2D array, each "element" is actually a row of the array.

```
char* b[2][3];
b+1;  // Pointer to the second row of the array: 0x100 (start of array) + 12 (size of one row) = 0x10C
*(b+1) + 2;  // Pointer to the third element of the second row: 0x10C (start of second row) + 8 (size of two pointers) = 0x114
```

- **Pointer to Array**: The name of an array is a pointer to the first element of the array. When you take the address of an array with the `&` operator, you get a pointer to the array itself, which is a different type than a pointer to the first element.

```
char* b[2][3];
&b;  // Pointer to the array itself: 0x100
&b + 1;  // Pointer to the memory location immediately following the array: 0x100 (start of array) + 24 (size of array) = 0x118
```

- **Dereferencing a Pointer**: The `*` operator dereferences a pointer, giving the object that the pointer points to. When used with a pointer to an array, it gives the first element of the array.

```
char* b[2][3];
*(b+0);  // First row of the array: 0x100
*(*(b + 1) + 2);  // Third element of the second row: 0x114
```

- **Array Subscripting and Pointers**: Array subscripting `[]` is a shorthand for pointer arithmetic and dereferencing. `b[i]` is equivalent to `*(b+i)`, and `b[i][j]` is equivalent to `*(*(b+i)+j)`.

```
char* b[2][3];
b[0];  // First row of the array: 0x100
b[0] + 1;  // Second element of the first row: 0x100 (start of first row) + 4 (size of one pointer) = 0x104
```

- **Pointer Difference**: The difference between two pointers is the number of elements of the pointer's type between the two memory locations. This can be useful for finding the length of an array if you have pointers to the start and end.

```
int arr[5] = {1, 2, 3, 4, 5};
int* start = arr;
int* end = arr + 5;
int length = end - start;  // length is 5
```

- **Pointer Comparison**: Pointers can be compared using relational operators like `<`, `<=`, `>`, `>=`, `==`, and `!=`. This can be useful for checking if a pointer is within the bounds of an array.

```
int arr[5] = {1, 2, 3, 4, 5};
int* p = arr + 2;
if (p >= arr && p < arr + 5) {
    // p points to an element within arr
}
```

- **Pointer to Pointer**: A pointer can point to another pointer, creating what's known as a double pointer. This can be useful for dynamic 2D arrays, among other things.

```
int x = 5;
int* p = &x;
int** pp = &p;
// Now *p is x, and **pp is also x
```

- **Void Pointers**: A void pointer is a special type of pointer that can point to objects of any type. A void pointer must be cast to another pointer type before it can be dereferenced.

```
int x = 5;
void* p = &x;
int* ip = static_cast<int*>(p);
// Now *ip is x
```

- **Function Pointers**: A function pointer is a pointer that points to a function. This can be useful for passing functions as arguments to other functions.

```
void hello() {
    std::cout << "Hello, world!" << std::endl;
}
```

```
void (*fp)() = hello;
fp();  // Calls hello
```

_Example submitted by senpai#4503

```
// Pointers and 1d arrays


a[i] = *(a + i) // this is a value
&a[i] = (a + i) // this is an address



          a[0]     a[1]     a[2]     a[3]
         _______  _______  _______  _______
        |       ||       ||       ||       |
a =     |   10  ||   20  ||   30  ||   40  |
        |       ||       ||       ||       |
        |_______||_______||_______||_______|
        |        |        |        |
        a       a+1      a+2      a+3


a[1] = 1[a] = *(a + 1)
a[2] = 2[a] = *(a + 2)


// Pointers and 2d arrays


int b[3][2] = {10,20,30,40,50,60}

b[i] = *(b + i)
&b[i] = (b + i)

b[i][j] = *(*(b+i) + j) // value of b[i][j]
&b[i][j] = *(b + i) + j // address of the value of b[i][j]
```

Integration (something fun after all this)

First off, here is the solution. As you know this calculates the area under a curve, if not, it doesn't matter, we don't care about the calculus here:

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

typedef double (*FUNC)(double);

// Returns the area under the curve between x=a and x=b.
// The function passed as the first parameter determines the shape of the curve.
// Use a Riemann sum to estimate the area by dividing it into narrow rectangles
(each rectangle should have a width of .0001)
// and adding up the areas of the individual rectangles.
// For a rectangle at horizontal position x, the height will be f(x).
// (when you call f(x) it'll call whatever function you passed as the first
parameter)
double integrate(FUNC f, double a, double b) {
    // Lowered the width of the rectangles to .000001 to get a more accurate result
    double width = .000001;
    int total_squares = (b-a)/width;
    double area = 0;
    for(int i = 0; i< total_squares; i++){
        area += f(a + i * width) * width;
    }
    return area;
}

double line(double x){
    return x;
}

double square(double x){
    return x*x;
}

double cube(double x){
    return x*x*x;
}

int main() {
    cout << "The integral of f(x)=x between 1 and 5 is: " << integrate(line, 1, 5)
<< endl;     // 12
    cout << "The integral of f(x)=x^2 between 1 and 5 is: " << integrate(square, 1,
5) << endl; // 41.3333
    cout << "The integral of f(x)=x^3 between 1 and 5 is: " << integrate(cube, 1,

5) << endl; // 156
```

```
5) << endl;   // 156
    return 0;
}
```

Let's begin the breakdown, wohooo (I will be one breaking down after I am done with this):

1. `typedef double (*FUNC)(double);`

This line is creating a new type alias using `typedef`. `FUNC` is now a type representing a pointer to a function that takes a `double` as an argument and returns a `double`. But what does this even do? Well first of all, it forces that the function provided to `integrate` is one that receives a double and returns a double after execution. And then why do we use `*FUNC`, goddamn pointers back at it again. Well, we do not want to pass the function itself (more like we can't even do it), but rather a pointer that tells us where the function is so we can use it, that way we can dynamically pass any of the functions we have to integrate.

But now you might be like wtf, how can we pass arguments to a memory address, but when we refer to the function address with parentheses, the function pointer, in this case `f`, calls the function and executes its code, and we can provide it arguments. This is magical polymorphism.

In short, a function pointer without parentheses will return the function's address, add parentheses and boom you got yourself an execution of the function - a function call.

Bubble Sort

Solution, thanks to senpai#4503

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

// parameterized bubble sort

typedef bool (*comparator)(int, int);

bool ascending(int k, int l){
    return k < l;
}

bool descending(int k, int l) {
    return k > l;
}

void sort(int num[], int size, comparator comp) {
```

```
    bool flag = true; // set flag to true to start first pass
    int temp;

    for (int i = 1; (i <= size) && flag; i++) {
        flag = false;

        for (int j = 0; j < (size - 1); j++) {
            if (comp(num[j+1], num[j])) {
                temp = num[j]; // swap elements
                num[j] = num[j + 1];
                num[j + 1] = temp;
                flag = true; // indicates a swap occurred
            }
        }
    }
    return;
}

int values[] = {40, 10, 100, 90, 20, 25, 33};

int main() {
    sort(values, 7, ascending);
    for (int n = 0; n < sizeof(values)/sizeof(values[0]); n++) {
        cout << values[n] << " ";
    }
    cout << endl;
    return EXIT_SUCCESS;
}
```

That looks pretty, but what's the point? Well, to sort in either ascending or descending order, you can pick your poison, but we are able to pass in either function which will be used to check (descending or ascending)

```
typedef bool (*comparator)(int, int);
```

by using typedef to pass a function pointer, this is explored very in-depth in the Integrals problem.

```
bool flag = true;
int temp;
```

Inside the `sort` function, `flag` is a boolean variable used to determine if the array is already sorted. `temp` is a temporary variable used to hold a value during the swapping of two elements.

```
for (int i = 1; (i <= size) && flag; i++) {
    flag = false;

    for (int j = 0; j < (size - 1); j++) {
        if (comp(num[j], num[j + 1])) {
            temp = num[j];
            num[j] = num[j + 1];
            num[j + 1] = temp;
            flag = true;
        }
    }
}
```

This is the heart of the bubble sort. It's two nested loops. The outer loop (`i`) continues until `i` has reached the `size` of the array or the `flag` has been set to `false`. Inside the outer loop, `flag` is set to `false`, indicating that no swaps have occurred yet. The inner loop (`j`) goes through the array, checking each pair of elements (`num[j]` and `num[j + 1]`). The `comp` function is used to decide whether to swap the pair. If a swap occurs, `flag` is set to `true`.

A bit of a deeper dive into the inner loop which is the meaty greedy of this bad boy,

This loop performs a pass through the array, comparing each pair of adjacent elements. It starts at the beginning of the array (`j = 0`) and goes up to one less than the size of the array (`j < size - 1`), because it's comparing pairs of elements: `num[j]` and `num[j + 1]`.

The `if` statement inside the inner loop is where the comparison function `comp` comes into play. It compares the current pair of adjacent elements, `num[j]` and `num[j + 1]`. If `comp(num[j], num[j + 1])` is `true`, it means these two elements are in the wrong order according to the sorting criteria defined by the comparison function, so they need to be swapped.

The swapping operation takes place right after the `if` statement: the current element `num[j]` is saved in a temporary variable `temp`, then `num[j]` is set to the next element `num[j + 1]`, and finally, `num[j + 1]` is set to the value saved in `temp`. Thus, `num[j]` and `num[j + 1]` have been swapped.

When a swap occurs, `flag` is set to `true`. This indicates that at least one swap operation took place during this pass, so the array might not be completely sorted yet, and further passes may be needed.

Time & Space Complexity

Best Case Scenario (Array is already sorted when inputted) = O(n) since it would iterate through all of the elements once to confirm it is sorted as the flag stops the sorting when no

swaps are made in a complete pass.

Average Case Scenario & Worst Case: O(n^2), in short trash and hot garbage, but essentially when elements are partially randomized or what not, just sucks as an algo, but it is the simplest sorting algorithm.

Space Complexity: O(1), you sort in-place meaning good for memory since you aren't creating a new array or something, making this very space-efficient.

K Bishops

Our cheap rip-off from N Queens, but we love it somewhat. But here is the code:

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

bool ok(int q[], int c, int n) {
    for (int i = 0; i < c; i++) {
        if (abs(q[c]/n - q[i]/n) == abs(q[c]%n - q[i]%n)) {
            return false;
        }
    }
    return true;
}

int nbishops(int n, int k) {
    int* q = new int[k];
    q[0] = 0;
    int c = 0, count = 0;
    while (c >= 0) {
        if (c == k) {
            count++;
            c--;
        } else {
            q[c] = c > 0 ? q[c - 1] : -1;
        }
        while (c >= 0 && c < k) {
            q[c]++;
            if (q[c] == n*n) {
                c--;
            } else if (ok(q, c, n)) {
                c++;
                break;            }
        }
    }
    delete[] q;
```

```
    return count;
}

int main() {
    int n, k;
    while (true) {
        cout << "Size of the board: ";
        cin >> n;
        if (n == -1){
            break;
        }
        cout << "# of bishops: ";
        cin >> k;
        cout << "Number of solutions: " << nbishops(n, k) << "\n\n";
    }
    return 0;
}
```

Now let's break down together, I mean break the problem down together. This is very similar to N Queens, so we will skim over a lot of stuff, or maybe not, lets see how my fingers do while typing.

So we have our beautiful `ok()` function.

```
bool ok(int q[], int c, int n) {
    for (int i = 0; i < c; i++) {
        if (abs(q[c]/n - q[i]/n) == abs(q[c]%n - q[i]%n)) {
            return false;
        }
    }
    return true;
}
```

So here we have `q[]`, the array that represents our board. Each index of the array represents a column and the value at that index represents the cell number where the bishop is placed in that column. Cell numbers are numbered row by row from 0 to n*(n-1), where here n is the sizeo fthe board. (pew pew)

The loop we have iterates from 0 to `c-1`. For each column `i`, it checks if the bishop in that column is in the same diagonal with the bishop in column `c`. In chess, two pieces are in the same diagonal if the difference of their row numbers is equal to the difference of their column numbers.

Here, `q[c]/n` and `q[i]/n` represent the row numbers of the bishops in column `c` and `i` (because we are dividing the cell number by the number of cells in a row). Similarly, `q[c]%n` and `q[i]%n` represent the column numbers (because we are taking the cell number modulo the number of cells in a row).

If the absolute differences of their row numbers and column numbers are equal, the two bishops are in the same diagonal. In that case, the function returns false. If no threatening bishop is found, the function returns true.

That probably sounded super confusing, so let's work with an example. Pretend we had the following 3x3 board:

```
0 1 2
3 4 5
6 7 8
```

If a bishop is placed in cell number 1, then the **row number** of the bishop would be 1/3 = 0 (using integer division where the fractional part is discarded). Meaning the bishop is in row 0 (index 0).

Then to determine its column, `1 % n` or 1 % 3 in this case which is equal to 1. This means the bishop is in column 1, which it is (index 0).

Now let's put a bishop in cell number 5, we do the math and row = 5/3 = 1 & column = 5 % 3 = 2. So the bishop is in row 1 (0 index) and column 2 (0 index), and this is correct. Now to determine if the second bishop is in the same diagonal as the first, we can check if the subtraction of rows is equal to their columns subtracted.

```
abs(B1_row - B2_row) = abs(0 - 1) = abs(-1) = 1
abs(B1_column - B2_column) = abs(1 - 2) = abs(-1) = 1

Since the values are both equal, they are both in the same diagonal
```

Let's cover a quick overview of the main function or I will let our GPT friend do it (I will curate whatever it says so do not fear)

```
int nbishops(int n, int k) {
    int* q = new int[k];
    q[0] = 0;
    int c = 0, count = 0;
    while (c >= 0) {
        if (c == k) {
            count++;
```

```
            c--;
        } else {
            q[c] = c > 0 ? q[c - 1] : -1;
        }
        while (c >= 0 && c < k) {
            q[c]++;
            if (q[c] == n*n) {
                c--;
            } else if (ok(q, c, n)) {
                c++;
                break;            }
        }
    }
    delete[] q;
    return count;
}
```

```
int nbishops(int n, int k) {
    int* q = new int[k];
    q[0] = 0;
    int c = 0, count = 0;
```

The `nbishops()` function takes the size of the board `n` and the number of bishops `k` as inputs and returns the number of ways to place the bishops on the board (duh). It initializes an array `q` of size `k` to represent the board, this is the array we use.

Each index of the array represents a column and the value at that index represents the cell number where the bishop is placed in that column. This means that if you select an item on the array, the index of that item is the column of the bishop and the value of that item is the cell number. It also initializes `c` as the current column and `count` as the number of valid ways to place the bishops.

```
    while (c >= 0) {
        if (c == k) {
            count++;
            c--;
        } else {
            q[c] = c > 0 ? q[c - 1] : -1;
        }
```

The function then enters a loop that continues as long as `c` is non-negative. If `c` equals `k`, it means that `k` bishops have been placed on the board in a valid way, so it increments `count` and moves to the previous column (backtracks to try and find more solutions). Otherwise, it

sets the cell number for the bishop in column `c` to be the same as the cell number for the bishop in the previous column if `c` is greater than 0, or -1 if `c` is 0.

```
        while (c >= 0 && c < k) {
            q[c]++;
```

This loop continues as long as `c` is non-negative and less than `k`. Inside the loop, the cell number for the bishop in column `c` is incremented. This represents moving the bishop to the next cell in the same column.

```
            if (q[c] == n*n) {
                c--;
            } else if (ok(q, c, n)) {
                c++;
                break;
            }
        }
    }
```

If the cell number equals `n*n`, it means that all cells in column `c` have been checked, so the function moves back to the previous column. Otherwise, it checks if the current placement of the bishop is safe using the `ok()` function. If the placement is safe, the function moves to the next column and breaks the inner loop.

```
    delete[] q;
    return count;
}
```

Finally, the function deallocates the memory allocated for the array `q` and returns the count of valid ways to place the bishops. Give it up for our GPT friend, I added some clarifications to make things a bit easier to understand.

N Queens

I used to love chess right before this class, I hear the word queen again and I will be like the last English queen. Anyways, let us dig into N queens.

My solution:

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

bool ok(int q[], int c) {
```

```
    for (int i = 0; i < c; i++) {
        if ((q[i] == q[c]) || (abs(q[c] - q[i]) == (c - i))) {
            return false;
        }
    }
    return true;
}

int nqueens(int n) {
    int* q = new int[n]; // Dynamically declare an array of size n
    q[0] = 0;
    int c = 0, count = 0;

    while (c >= 0) {
        c++;
        if (c == n) {
            count++;
            c--;
        } else {
            q[c] = -1;
        }

        while (c >= 0) {
            q[c]++;
            if (q[c] == n) {
                c--;
            } else if (ok(q, c)) {
                break;
            }
        }
    }
    delete[] q; // Delete the array
    return count;
}

int main() {
    int n;
    cout << "Enter a number: ";
    cin >> n;
    cout << endl;
    for (int i = 1; i <= n; ++i) {
        cout << i << ". There are " << nqueens(i) << " solutions to the " << i << "
queens problem.\n";
    }
    return 0;
}
```

We have our beautiful `ok()` function that we have used for the entire semester. Short rundown, The first condition `q[i] == q[c]` checks if there is a queen in the same row. The second condition `abs(q[c] - q[i]) == (c - i)` checks if there is a queen in the same diagonal.

In chess, two pieces are in the same diagonal if the difference of their row numbers is equal to the difference of their column numbers. Similarly, two pieces are on the same row if their umm rows are the same... so yeah cool stuff. We don't check if they are in the same column here unlike K Bishops as we are placing the queens incrementally from column 0 to n.

.

```
bool ok(int q[], int c) {
    for (int i = 0; i < c; i++) {
        if ((q[i] == q[c]) || (abs(q[c] - q[i]) == (c - i))) {
            return false;
        }
    }
    return true;
}
```

Then we have our main function (which yes, it's not called main, but you get the point):

```
int nqueens(int n) {
    int* q = new int[n]; // Dynamically declare an array of size n to hold queen
positions
    q[0] = 0; // Initialize the first queen's position in the first column to 0
    int c = 0; // Initialize the current column to 0
    int count = 0; // Initialize the count of valid solutions to 0

    while (c >= 0) { // Continue until all columns are explored
        c++; // Move to the next column

        if (c == n) { // If all queens are successfully placed
            count++; // Increment the count of valid solutions
            c--; // Backtrack to the previous column to explore more possibilities
        } else {
            q[c] = -1; // Reset the position of the next queen to -1 (to be
incremented)
        }

        while (c >= 0) { // Explore positions in the current column
            q[c]++; // Move to the next position in the current column

            if (q[c] == n) { // If all positions are tried in the current column
                c--; // Backtrack to the previous column
            } else if (ok(q, c)) { // If the current position is safe for the queen
```

```
                break; // Exit the inner loop and move to the next column
            }
        }
    }

    delete[] q; // Free the dynamically allocated array
    return count; // Return the count of valid solutions
}
```

It starts by dynamically creating an array `q` of size `n` to hold the position of queens in the columns of the board. The `count` variable will keep track of the number of valid solutions.

```
int nqueens(int n) {
    int* q = new int[n];
    q[0] = 0;
    int c = 0, count = 0;
    // ...
}
```

This loop implements a form of backtracking. It tries to place a queen in every position of the current column `c`. If a queen can be placed (`c == n`), `count` is incremented, signifying a successful placement of `n` queens, and the loop goes back to the previous column (`c--`). If a queen can't be placed, the position is set to `-1`.

```
while (c >= 0) { // Continue until all columns are explored
        c++; // Move to the next column

        if (c == n) { // If all queens are successfully placed
                count++; // Increment the count of valid solutions
                c--; // Backtrack to the previous column to explore more
possibilities
        } else {
                q[c] = -1; // Reset the position of the next queen to -1 (to be
incremented)
                }

        ```

This is where the algorithm tries to find a safe position for the queen in the current column. It keeps incrementing the position in the current column (q[c]++). If it exceeds n, it signifies that it has tried all positions in the current column and failed to place a queen, hence it backtracks to the previous column (c--). If a safe position is found (ok(q, c) returns true), it breaks the loop and moves to the next column.
```
```

```cpp
while (c >= 0) { // Explore positions in the current column
        q[c]++; // Move to the next position in the current column

        if (q[c] == n) { // If all positions are tried in the current column
                c--; // Backtrack to the previous column
        } else if (ok(q, c)) { // If the current position is safe for the queen
                break; // Exit the inner loop and move to the next column
        }
}
```

Dynamic Memory

Now that we are refreshed with some sleep, let's begin with a simple example of dynamic memory. This is allocating a single object, in this case, an integer. Let's break it down to understand the syntax behind dynamic memory allocation.

1. Dynamic Allocation of a Single Object:

```cpp
int* dynamicInt = new int; // Allocates memory for a single integer
*dynamicInt = 10; // Assigns a value to the dynamically allocated integer
delete dynamicInt; // Deallocates the memory to prevent memory leaks
```

First we have `int*` which declares a pointer variable, then we provide the pointer the name `dynamicInt`. We set the pointer equal to `new int`, here the keyword `new` is used to allocate memory for objects or arrays at runintime from the heap (so their memory is allocated on the heap once the program is executed). After `new` allocates memory, it returns a pointer to the allocated memory for the data type, and then we can use that pointer as seen by `*dynamicInt = 10;` where we set the value of the integer pointer. Then once we no longer need the data, we deallocate the memory using `delete dynamicInt;` and the pointer is deleted with the information inside the memory. This prevents memory leaks and decreases the likelihood of performance issues.

Here are other examples of dynamic memory allocation:

1. Dynamic Allocation of an Array of Floating-Point Numbers:

```cpp
int size = 5;
float* dynamicFloatArray = new float[size]; // Allocates memory for an array of
floating-point numbers
for (int i = 0; i < size; i++) {
    dynamicFloatArray[i] = 1.5 * i; // Assigns values to the dynamically allocated
array
}
```

```
// ...
delete[] dynamicFloatArray; // Deallocates the memory for the array
```

2. Dynamic Allocation of a String:

```
const string source = "Hello, World!";

string* dynamicString = new string(source);

cout << "Source: " << source << endl;
cout << "Dynamic String: " << *dynamicString << endl;

delete dynamicString;
```

3. Dynamic Allocation of a 2D Array of Booleans:

```
int rows = 3;
int cols = 4;

// Dynamically allocate memory for the 2D array
int** dynamic2DArray = new int*[rows]; // Allocate memory for the outer array
(rows)
for (int i = 0; i < rows; i++) {
    dynamic2DArray[i] = new int[cols]; // Allocate memory for each inner array
(columns)
}

// Accessing and modifying elements of the dynamically allocated 2D array
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        dynamic2DArray[i][j] = i * cols + j; // Assign values to the 2D array
    }
}

// Deallocate the memory for the 2D array
for (int i = 0; i < rows; i++) {
    delete[] dynamic2DArray[i]; // Deallocate the memory for each inner array
(columns)
}
delete[] dynamic2DArray; // Deallocate the memory for the outer array (rows)
```

Short explanation, we have `int**` as the aterisks serve two purposes.

- First Asterisk: The first asterisk indicates that `dynamic2DArray` is a pointer variable. It is used to store the memory address of another pointer.
- Second Asterisk: The second asterisk indicates that the pointer stored in `dynamic2DArray` points to another pointer. It represents the memory address of the first element of an array of pointers.

We then have `new int*[rows];` and there is an aterisk after int to denote that the value stores inside of int will be pointers to integers. Finally we have `new int[cols]`; which allocates memory to store `cols` number of integers in a row.

4. Dynamic Allocation of Objects with Constructor Arguments:

```
class MyClass {
public:
    int data;
    MyClass(int d) : data(d) {}
};

MyClass* dynamicObject = new MyClass(42); // Allocates memory for an object with
constructor argument
// ...
delete dynamicObject; // Deallocates the memory for the object
```

8 Queens

So a lot of this was covered in N Queens, but I will provide the solutions for all the others.

Dumb 8 Queens 1D:

```
bool ok(int q[]) {

    // Current Queen
    for (int i = 0; i < 8; i++) {
        // Runs through all the other queen positions in the array
        for (int j = 1 + i; j < 8; j++) {
            // Checks to see if the rows are equal first, then it checks if when
the columns and rows are subtracted, both equal the same, therefore being in a
diagonal
            if (q[i] == q[j] || abs(i - j) == abs(q[i] - q[j])) {
                return false;
            }
        }
    }
```

```
    return true;
}
```

2D Eight Queens Cells Check

```
    nextRow:
    r++;

    if (r == 8) goto backtrack;

    // row test
    for (int i = 0; i < c; i++){
        if (q[r][i] == 1) goto nextRow;
    }

    // up diagonal test

    for (int i = 1; ((r-i)>=0 && (c-i)>=0); i++){
        if (q[r-i][c-i] == 1) goto nextRow;
    }

    // down diagonal test

    for (int i = 1; ((r+i)<8 && (c-i)>=0); i++){
        if (q[r+i][c-i] == 1) goto nextRow;
    }

    q[r][c] = 1;
    goto nextCol;
```

1D Eight Queens with Backtracking

```
bool ok(int q[], int c) {

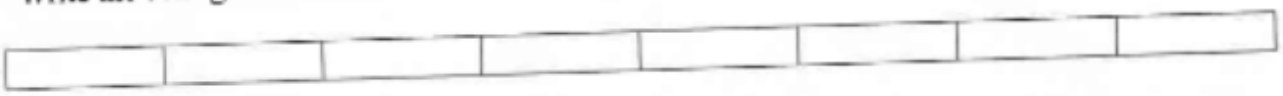
    for (int i = 0; i < c; i++) {
    /*
    First, check if any rows to the left are equal
    Second, check if any diagonals have been taken.
    abs(q[c] - q[i]) gives the difference between betwewen the rows
    c - i gives the difference between the rows
    If they are both equal, then that means they are on the same diagonal
    */
        if ((q[i] == q[c]) || (abs(q[c] - q[i]) == (c - i))) {
            return false;
        }
```

```
    }
    return true;
}
```

Other 8 Queens Problems

a. What will be the configuration printed when the value of "**count**" printed is 289? (5 points)

Write the configuration here:



Explain fully how you got the answer here:

```
Convert the number to base 8 as there are  8 variables in the loop:


289 = 4 * 8^2 + 4 * 8^1 + 1 * 8^0
```

Classes-Rat

Will add explanation from people in the chat or when I watch the videos, I get the basic gist tho.

Finessed the code from Github:

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

// Class representing rational numberrs
class Rat {
private:
    int n;
    int d;
public:
    Rat(){  // default constructor
        n = 0;
        d = 1;
    }

    Rat(int i){  // conversion constructor
        n = i;
        d = 1;
    }
```

```
    Rat(int i, int j){ // Two parameter constructor
        n = i;
        d = j;
    }

    // Accessor functions (get() and set())
    int getN(){ return n;}
    int getD(){return d;}

    void setN(int i){n = i;}
    void setD(int i){d = i;}


    // Overload arithmetic operators
    Rat operator+(const Rat r){
        return Rat(n*r.d+d*r.n,d*r.d);
    }

    Rat operator-(const Rat r){
        return Rat(n*r.d-d*r.n,d*r.d);
    }

    Rat operator*(const Rat r){
        return Rat(n*r.n,d*r.d);
    }

    Rat operator/(const Rat r){
        return Rat(n*r.d,d*r.n);
    }


    friend ostream& operator<<(ostream& os, Rat r);
    friend istream& operator>>(istream& is, Rat r);
};

// Overload of ostream operator
// Returns a reference to an ostream object in order to chain output.
ostream& operator<<(ostream& os, Rat r){
    if (r.n == 0)
        os << r.n;
    else if(r.n == r.d)
        os << 1;
    else if (r.n > r.d)
        os << (r.n % r.d > 0 ? r.n/r.d << r.n%r.d: r.n/r.d);
    else        os << r.n << "/" << r.d;
```

```
    return os;
}

istream& operator>>(istream& is, Rat r){
    char slash = '/';
    is >> r.n >> slash >> r.d;
    return is;
}

int main() {
    Rat r1(1, 2), r2(3, 2);
    Rat r3(4/2);

    cout << "r1: " << r1 << endl;
    cout << "r2: " << r2 << endl;
    cout << "r1 + r2: " << r1 + r2 << endl;
    cout << "r1 - r2: " << r1 - r2 << endl;
    cout << "r1 * r2: " << r1 * r2 << endl;
    cout << "r1 / r2: " << r1 / r2 << endl;
    cout << "test: " << r3 << endl;

    return 0;
}```

### Class Provided Notes

**Credits to snepai#4503**
```

How does public have access to private? It’s because they are member functions part of the same class.

If you want to add ½ to ⅔ you need to define a member function to have access to the int in private.
If x and y are Rats then x+y is really syntactic sugar for x.operator+(y)

Rat x,y;
X = y + 5
We can do this because the compiler converts 5 into 5/1. The function knows how to add rats.
x.operator+(y)
The left operator (x) has to be a Rat.

Rat x,y;
X = 5 + y;
This cant work because the method must be an object of the class. The left operator is not a Rat.

A solution to this is to overload operator+. You can have different functions with the same name.
operator+(int, Rat)
operator(Rat, Rat)
Define the function as not as a member function.

Friend ostream& operator<<(ostream& os, Rat r);
Friend istream& operator<<(istream& is, Rat r);

Why do we pass by reference?
When you want to make a change
When you dont want a copy of ostream
Why do we return os?
For something called CHAINING
Cout << i << 2 << “Hello”
(((x+y) + z) + m)

````
## Memoization

So what is this? Essentially storing or caching results of function calls and reusing them when the same inputs are necessary again, so for example, factorials make this easy to understand.

`5! = 5 * 4 * 3 * 2 * 1`

If we wanted to get `6!` we could have to multiply all the numbers again, but why do that when we know what` 5!` is, less computation, so we store `5!`.

To get `6!`, I don't have to do `6 * 5 * 4 * 3 * 2 * 1`, but rather `6 * 5!`, and I already have the value of `5!`, so a lot less computation is needed.

This is the code for factorials:
```cpp
#include <iostream>
#include <vector>

std::vector<long long> factorialCache;

void initializeCache(int n) {
    factorialCache.resize(n + 1, -1);
````
````

```
}

long long factorial(int n) {
    if (factorialCache[n] != -1)
        return factorialCache[n];

    if (n <= 1)
        return factorialCache[n] = 1;

    return factorialCache[n] = n * factorial(n - 1);
}

int main() {
    int n = 10;
    initializeCache(n);

    std::cout << "Factorial of " << n << ": " << factorial(n) << std::endl;

    return 0;
}
```

Here is an array-based implementation:

```
#include <iostream>

long long factorialCache[1000];

void initializeCache(int n) {
    for (int i = 0; i <= n; ++i)
        factorialCache[i] = -1;
}

long long factorial(int n) {
    if (factorialCache[n] != -1)
        return factorialCache[n];

    if (n <= 1)
        return factorialCache[n] = 1;

    return factorialCache[n] = n * factorial(n - 1);
}

int main() {
```

```
    int n = 10;
    initializeCache(n);

    std::cout << "Factorial of " << n << ": " << factorial(n) << std::endl;

    return 0;
}
```

Here is how it is done for the fibonacci sequence:

Vector:

```
#include <iostream>
#include <vector>

std::vector<long long> fibCache;

void initializeCache(int n) {
    fibCache.resize(n + 1, -1);
}

long long fibonacciMemoization(int n) {
    if (n <= 1)
        return n;

    if (fibCache[n] != -1)
        return fibCache[n];

    return fibCache[n] = fibonacciMemoization(n - 1) + fibonacciMemoization(n - 2);
}

int main() {
    int n = 10;
    initializeCache(n);

    std::cout << "Fibonacci number at position " << n << " (using memoization): "
<< fibonacciMemoization(n) << std::endl;

    return 0;
}
```

Array:

```
#include <iostream>
```

```cpp
long long fib(int n) {
    // Static array for memoization, initialized to zero
    static int memo[1000] = {0};

    // Base cases
    if(n == 0) {
        return 0;
    } else if(n == 1) {
        return 1;
    }

    // If the value has already been computed, return it
    if(memo[n] != 0) {
        return memo[n];
    }

    // Otherwise, compute the value
    memo[n] = fib(n - 1) + fib(n - 2);

    // Return the result
    return memo[n];
}

int main() {
    for(int i = 0; i <= 20; i++) {
        std::cout << "Fibonacci of " << i << " = " << fib(i) << std::endl;
    }

    return 0;
}
```

Hexadecimals

Hexadecimal is a number system that uses base 16 instead of base 10, which we use in our everyday decimal system. Here is a fun fact that is not so fun because who cares, but the name "hexadecimal" is derived from Greek "hexa" which means six and "deca" means ten, combine themand you get 16.

In a hexadecimal system, numbers are represented by 16 different digits:

- 0-9 to represent zero to nine
- A, B, C, D, E, F to represent ten to fifteen

Hexadecimal to Decimal Conversion:

Let's consider a hexadecimal number 3B9.

The decimal equivalent is calculated as follows:

(3 *(16^2))* + *(11* (16^1)) + (9 * (16^0)) = 953

Note that B in hexadecimal is 11 in decimal.

Memory Addresses:

In computing, hexadecimal is commonly used to represent computer memory addresses. Each byte of memory is uniquely identified by a hexadecimal address. For example, a memory address might look like 3F5BCE.

Hexadecimal Arithmetic:

Arithmetic with hexadecimal numbers works similarly to arithmetic in any other number system. The only thing you need to remember is the set of hexadecimal digits (0-9 and A-F).

Addition and subtraction follow the usual rules, with the caveat that you carry or borrow from the 16th place rather than the 10th.

For example, if you're adding 9 (in hexadecimal) and 8 (in hexadecimal), the result would be 17 (in decimal). But in hexadecimal, 17 is represented as 11. So, you would write down 1 and carry 1 to the next column.

Our Holy Trinity of Hexadecimals

1. Know your hexadecimal digits. 0-9 are the same as in decimal, but A-F represent 10-15.
2. When converting from hexadecimal to decimal, remember that each digit in the hexadecimal number refers to a power of 16. Start from the rightmost digit (16^0), move to the left (16^1), and so on. Multiply each digit by the corresponding power of 16 and add the results.
3. When converting from decimal to hexadecimal, divide the decimal number by 16 and keep track of the remainder. The quotient is then divided by 16 again, and this process is repeated until the quotient is 0. The hexadecimal number is then the remainders, read in reverse order.
4. For arithmetic, remember to carry or borrow from the 16th place rather than the 10th.
5. For memory addresses, remember that each pair of hexadecimal digits represents a byte. The rightmost pair is the least significant byte and the leftmost pair is the most significant byte.

Example with a Memory Address

If you have two memory addresses, say 3F5BCE and 4A3D2F, and you want to add them together, you would do so just like adding two hexadecimal numbers, because that's what they are.

Here's what addition looks like:

```
  3F5BCE
+ 4A3D2F
--------
  8998FD
```

Here's what subtraction looks like:

```
  4A3D2F
- 3F5BCE
--------
   AE161
```

Structs

Structs are a way to define new data types that can group together related data items of different name types. It is very similar to a class with a few differences such as the access specified being private for a class and public for a struct. Here is some sample syntax:

```cpp
#include <iostream>

// Define a struct called "Person"
struct Person {
    std::string name;
    int age;
};

int main() {
    // Create an instance of the struct
    Person person1;

    // Assign values to the struct members
    person1.name = "John Doe";
    person1.age = 25;

    // Access and print the struct members
    std::cout << "Name: " << person1.name << std::endl;
    std::cout << "Age: " << person1.age << std::endl;
```

```
    return 0;
}
```

In the example , we define a struct called "Person" with two members: a string member called "name" and an integer member called "age". Inside the `main` function, we create an instance of the `Person` struct named `person1`. We can access and modify the struct members using the dot operator (`.`). Very similar to a class right?

Structures and Bytes:

```
struct MyStruct {
    int a;
    double b;
    char c;
};
```

1. `int a;` : The size of an `int` is typically 4 bytes (32 bits) in many modern systems, but it can vary. C++ standard doesn't specify the exact size for `int` but rather provides a minimum size.
2. `double b;` : The size of a `double` is typically 8 bytes (64 bits) in most modern systems.
3. `char c;` : The size of a `char` is 1 byte (8 bits) by definition in the C++ standard.

So, considering padding, the struct would likely be:

- 4 bytes for `int a`
- 4 bytes of padding to align `double b` on an 8-byte boundary
- 8 bytes for `double b`
- 1 byte for `char c`
- 7 bytes of padding to reach a total size that is a multiple of 8.

This totals to 24 bytes in size for the struct.

Vectors

Vectors occupy contiguous memory locations, which allow iteration over its elements usingpointers.

```
std::vector<int> vec;
```

Operations

1. **push_back()**: This function is used to input values into the vector from the end.

```
std::vector<int> vec;
for(int i=0; i<5; i++){
    vec.push_back(i);
}
```

2. **pop_back()**: This function is used to remove values from the vector from the end.

```
vec.pop_back();
```

3. **size()**: This function returns the number of elements present in the vector.

```
std::cout << "Vector size: " << vec.size() << std::endl;
```

4. **begin() and end()**: These functions return an iterator pointing to the beginning and the end of the vector respectively.

```
for (auto i = vec.begin(); i != vec.end(); ++i)
    std::cout << *i << " ";
```

5. **clear()**: This function is used to remove all elements of the vector.

```
vec.clear();
```

6. **insert()**: This function inserts new elements before the element at the specified position.

```
vec.insert(vec.begin() + 2, 10);
```

7. erase(): This function removes from the vector either a single element or a range of elements `[first,last)`.

```
vec.erase(vec.begin());
vec.erase(vec.begin(), vec.begin()+3);
```

Here is an example of a program:

```
#include<iostream>
#include<vector>

int main(){
    std::vector<int> vec;
```

```
    // Inserting elements
    for(int i=0; i<5; i++){
        vec.push_back(i);
    }

    // Displaying elements of vector
    std::cout << "Elements in the vector are: ";
    for(int i=0; i<vec.size(); i++){
        std::cout << vec[i] << " ";
    }

    // Removing an element from end
    vec.pop_back();
    std::cout << "\nElements after pop_back operation: ";
    for(int i=0; i<vec.size(); i++){
        std::cout << vec[i] << " ";
    }

    // Inserting elements at specific position
    vec.insert(vec.begin() + 2, 100);
    std::cout << "\nElements after insert operation: ";
    for(int i=0; i<vec.size(); i++){
        std::cout << vec[i] << " ";
    }

    // Removing element at specific position
    vec.erase(vec.begin() + 2);
    std::cout << "\nElements after erase operation: ";
    for(int i=0; i<vec.size(); i++){
        std::cout << vec[i] << " ";
    }

    // Clearing the vector
    vec.clear();
    std::cout << "\nElements in vector after clear operation: " << vec.size();

    return 0;
}
```

Output:

```
Elements in the vector are: 0 1 2 3 4
```

Operator Overloading

Operator overloading is a very powerful feature in C++ which lets us make the standard operators (like `+, -, *, /`, and so on) work with our own classes. We add stuff on to them or modify what they would return based on things such as our parameters.

Here is a class with the `+` operator overloaded:

```
#include <iostream>

class Complex {
public:
    int real, imag;

    Complex(int r = 0, int i =0) {  // constructor
      real = r;
      imag = i;
    }

    // Overload operator '+'
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void print() {
       std::cout << real << " + i" << imag << std::endl;
    }
};

int main() {
    Complex c1(10, 5), c2(2, 4);

    // An example call to operator '+'
    Complex c3 = c1 + c2;

    c3.print();
}
```

The operator overloading part is:

```
Complex operator + (Complex const &obj) {
    Complex res;
    res.real = real + obj.real;
    res.imag = imag + obj.imag;
```

```
    return res;
}
```

Here, `Complex operator + (Complex const &obj)` is a function that overloads the '+' operator. When we write `c1 + c2`, it's equivalent to `c1.operator+(c2)`. The function takes a `Complex` object as parameter, which represents the second operand of the '+'.

Inside the function, a new `Complex` object `res` is created, and its `real` and `imag` fields are set to the sum of the `real` and `imag` fields of the operands. The function then returns `res`.

WARNING: READ
Overloading the '+' operator does not affect the operation of '+' with built-in types like int or float. So 2 + 2 = 4 still. It only changes its operation when its operands (the things being operated on) are of type `Complex` or whatever name your class is named after.

A complex operator can also be declared as as global function:

```
#include <iostream>

class Complex {
public:
    int real, imag;

    Complex(int r = 0, int i =0) {  // constructor
      real = r;
      imag = i;
    }

    void print() {
       std::cout << real << " + i" << imag << std::endl;
    }
};

// Overload operator '+' as a non-member function aka global function
Complex operator + (Complex const &c1, Complex const &c2) {
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

int main() {
    Complex c1(10, 5), c2(2, 4);

    // An example call
```