MIDTERM EXAM 2

EMPLID

CSCI 135

NAME: FIRST LAST

1. Write code that creates `float` pointer variables to show each of the possibilities below. Include other variable definitions, when appropriate:

a) an uninitialized pointer

```
float * p;
```

b) a null pointer

```
float * p = nullptr;
```

c) a pointer to a single `float` variable with function scope

```
float x = 5;
float * p = & x;
```

d) a pointer to an array of `float` values with function scope

```
float x[3] = {3.5, 4.0, 5.1};
float * p = x; //or float * p = x[0];
```

e) a pointer to a single `float` object in dynamic memory

```
float * p = new float;
```

f) deallocate the memory in (e) and fix the dangling pointer

```
delete p;
p = nullptr;
```

g) a pointer to a dynamic array of `float` values

```
float * p = new float[5];
```

h) deallocate the memory in (g) and fix the dangling pointer

```
delete[] p;
p = nullptr;
```

2. Write a function that checks if two `float` arrays are of the same length and have the same values.

```
bool equals(float* a, int a_size, float* b, int b_size)
{
   if (a_size != b_size) {
      return false;
   }
   for (int i = 0; i < a_size; i++) {
      if (a[i] != b[i]) {
         return false;
      }
   }
   return true;



}
```

3. What does the following code print?

```
int x = 15;
int y = 25;
int* s = &x;
int* t = s;
t = &y;
y = *t + 5;
x = *t - *s;
cout << x << " " << y << endl;
```

```
15 30
```

First, use this table to show how values of variables change as instructions execute. Use the **address-of** operator to show values of pointer variables:

x	y	s	t
15	25	&x	&x
15	30		&y

4. Write a function that finds the first occurrence of a value in a two-dimensional array. Return an `int` array of length 2 with the indices of the row and column. For the returned array to persist beyond the scope of your function you must use dynamic memory. Do the clean up in `main()`.

```
const int COLUMNS = 5;
int* find_value(int values[][COLUMNS], int target, int rows) {
    bool found = false;
    int * results = new int[2]{-1, -1}; // initialize to not-found
    for (int i = 0; i < rows && !found; i++) {
        for (int j = 0; j < COLUMNS && !found; j++) {
            if (values[i][j] == target) {
                results[0] = i;
                results[1] = j;
                found = true;
            }
        }
    }
    return results;
}
int main() {
   int array[3][COLUMNS] = {{2, 1, 4, 9, 0}, {1, 0, 2, 7, 4}, {6, 7, 3, 8, 1}};
   int* results = find_value(array, 8, 3); // look for: 8
   cout << "8 found at: " << results[0] << " " << results[1];
   delete[] results; //deallocate dynamic memory
   results = nullptr; //fix dangling pointer
}
```

5. Write a code fragments that will use dynamic memory to initialize an "upside down" triangular array of characters with side 5, fill each element with character `'X'`, and print it out, so that it looks like this:

```
XXXXX
XXXX
XXX
XX
X
```

(This should remind you of the Galton Board example)

```
//Allocate rows in dynamic memory and set all to X
const int SIZE = 5;
char* rows[SIZE];
for (int i = 0; i < SIZE; i++) {
   rows[i] = new char[SIZE - i];
   for (int j = 0; j < SIZE - i; j++) {
      rows[i][j] = 'X';
   }
}
```

```
// Print out all rows of X's
for (int i = 0; i < SIZE; i++) {
   for (int j = 0; j < SIZE - i; j++) {
      cout << rows[i][j];
   }
   cout << endl;
}
```

```
// Deallocate the rows
for (int i = 0; i < SIZE; i++) {
   delete[] arrays[i];
}
```

6. Design a simple class **`Person`** that contains (or "has") the **`name`** of a person and two pointers: to the person's father and mother. In the **`main()`** function define objects for yourself and your parents, correctly establishing the pointer links. Use **`nullptr`** for your parents' parents.

```
class Person {
public:
    string name;
    Person * father;
    Person * mother;



};
```

```
int main() {
    Person mom;
    mom.name = "Carol";
    mom.father = nullptr;
    mom.mother = nullptr;
    Person dad;
    dad.name = "Bob";
    dad.father = nullptr;
    dad.mother = nullptr;
    Person me;
    me.name = "Alice";
    me.father = & dad;
    me.mother = & mom;
    return 0;


}
```

Variable and Constant Definitions

```
int cans_per_pack = 6;

const double CAN_VOLUME = 0.335;
```

Type *Name* *Initial value*

Mathematical Operations

```
#include <cmath>
```

`pow(x, y)`	Raising to a power x^y
`sqrt(x)`	Square root $\sqrt{x}$
`log10(x)`	Decimal log $\log_{10}(x)$
`abs(x)`	Absolute value $\lvert x \rvert$
`sin(x)`, `cos(x)`, `tan(x)`	Sine, cosine, tangent of x (x in radians)

Selected Operators and Their Precedence

(See Appendix B for the complete list.)

`[]`	Array element access
`++ -- !`	Increment, decrement, Boolean *not*
`* / %`	Multiplication, division, remainder
`+ -`	Addition, subtraction
`< <= > >=`	Comparisons
`== !=`	Equal, not equal
`&&`	Boolean *and*
`||`	Boolean *or*
`=`	Assignment

Loop Statements

```
while (balance < TARGET)
{
   year++;
   balance = balance * (1 + rate / 100);
}
```

Condition

Executed while condition is true

```
for (int i = 0; i < 10; i++)
{
   cout << i << endl;
}
```

Initialization *Condition* *Update*

```
do
{
   cout << "Enter a positive integer: ";
   cin >> input;
}
while (input <= 0);
```

Loop body executed at least once

Conditional Statement

```
if (floor >= 13)
{
   actual_floor = floor - 1;
}
else if (floor >= 0)
{
   actual_floor = floor;
}
else
{
   cout << "Floor negative" << endl;
}
```

Condition

Executed when condition is true

Second condition (optional)

Executed when all conditions are false (optional)

String Operations

```
#include <string>
string s = "Hello";
int n = s.length(); // 5
string t = s.substr(1, 3); // "ell"
string c = s.substr(2, 1); // "l"
char ch = s[2]; // 'l'
for (int i = 0; i < s.length(); i++)
{
   string c = s.substr(i, 1);
   or char ch = s[i];
   Process c or ch
}
```

Function Definitions

```
double cube_volume(double side_length)
{
   double vol = side_length * side_length * side_length;
   return vol;
}
```

Return type *Parameter type and name*

Exits function and returns result.

Reference parameter

```
void deposit(double& balance, double amount)
{
   balance = balance + amount;
}
```

Modifies supplied argument

Arrays

```
int numbers[5];
int squares[] = { 0, 1, 4, 9, 16 };
int magic_square[4][4] =
{
   { 16, 3, 2, 13 },
   { 5, 10, 11, 8 },
   { 9, 6, 7, 12 },
   { 4, 15, 14, 1 }
};

for (int i = 0; i < size; i++)
{
   Process numbers[i]
}
```

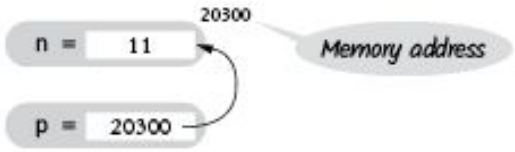
Element type *Length*

Enumerations, Switch Statement

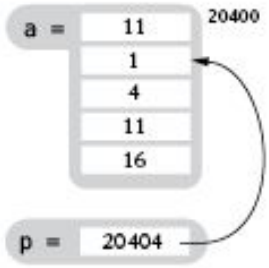
```
enum Color { RED, GREEN, BLUE };
Color my_color = RED;

switch (my_color) {
   case RED :
      cout << "red";   break;
   case GREEN:
      cout << "green"; break;
   case BLUE :
      cout << "blue";  break;
}
```

Pointers

```
int n = 10;
int* p = &n; // p set to address of n
*p = 11; // n is now 11
```




```
int a[5] = { 0, 1, 4, 9, 16 };
p = a; // p points to start of a
*p = 11; // a[0] is now 11
p++; // p points to a[1]
p[2] = 11; // a[3] is now 11
```



Input and Output

```
#include <iostream>
cin >> x; // x can be int, double, string
cout << x;

while (cin >> x) { Process x }
if (cin.fail()) // Previous input failed


#include <fstream>
string filename = ...;
ifstream in(filename);
ofstream out("output.txt");

string line; getline(in, line);
char ch; in.get(ch);
```

Range-based for Loop

```
for (int v : values)   // An array, vector, or other container (C++ 11)
{
   cout << v << endl;
}
```

Output Manipulators

```
#include <iomanip>
```

`endl`	Output new line
`fixed`	Fixed format for floating-point
`setprecision(n)`	Number of digits after decimal point for fixed format
`setw(n)`	Field width for the next item
`left`	Left alignment (use for strings)
`right`	Right alignment (default)
`setfill(ch)`	Fill character (default: space)

Class Definition

```
class BankAccount
{
public:
   BankAccount(double amount);          // Constructor declaration
   void deposit(double amount);         // Member function declaration
   double get_balance() const;          // Accessor member function
   ...
private:                                // Data member
   double balance;
};

void BankAccount::deposit(double amount)   // Member function definition
{
   balance = balance + amount;
}
```

Inheritance

```
class CheckingAccount : public BankAccount   // Derived class, Base class
{
public:
   void deposit(double amount);         // Member function overrides base class
private:
   int transactions;                    // Added data member in derived class
};

void CheckingAccount::deposit(double amount)
{
   BankAccount::deposit(amount);        // Calls base class member function
   transactions++;
}
```