# FINAL EXAM

EMPLID

CSCI 135 NAME: FIRST LAST

**1** (18%)

(3%) **i** What is the difference between passing by value and passing by reference to a function?

Pass by value gives a function a copy of the variable while pass by reference gives the actual object.

(3%) **ii** How does a function return a dynamic array?

A function returns a pointer to the first element of a dynamic array.

(3%) **iii** Explain the difference between the current size and capacity of an array.

The size of a vector is the number of elements that are stored, while the capacity is the size of the memory (in terms of element) that is currently allocated to the vector.

(3%) **iv** Explain the purpose of the **::** operator?

The scope resolution operator is used to refer to and implement member functions outside of the class.

(3%) **v** What is a constructor function, and what is special about its name?

A constructor is a member function of a class that is automatically called when an object of that class is instantiated.
A constructor has the same name of the class.

(3%) **vi** Why is it important to use the **delete** operator on a pointer to a dynamic object, before setting its value to **nullptr**?

Because all dynamic memory must be deallocated after use, and because, once the pointer is set to nullptr, it no longer points to the object that has to be deleted, and the memory can neither be accessed nor reclaimed.

**2** (12%)

(6%) **i** What is the output of the following code snippet? Show your work and put answer in the box:

```
string message = "Hello! ";
string* p = &message;
*p = "Hi! ";
message = "Salut! " + *p;
cout << *p << endl;
```

message
"Hello! "
"Hi! "
"Salut! Hi! "

**ANSWER: Salut! Hi!**

(6%) **ii** Given the following code snippet:

```
int arr[10] = { -3, 54, 23, 69, 88, 666, 31, 25, -69, 12 };
int* ptr = arr;
```

Write the code snippet that uses pointer arithmetic (not array notation) to output the value of **arr[3]**. Write the value in the box.

```
ptr = ptr + 3;
cout << *ptr << endl;
```

**ANSWER: 69**

**3** (10%)

Write a recursive function that adds up all integers from **0** to the positive integer **n**.
Hint: if **n** is **0**, the function returns **0**; otherwise, it returns **n** plus the sum of integers from **0** to **(n - 1)**.

```
int add_up(int n) {


    if (n == 0) //base case
    {
        return 0;
    }
    else //recursive case
    {
        return n + add_up(n - 1);
    }



}
```

**4** (8%) Write a function that returns a dynamic array, of size `n`, composed of sequential odd integers starting from `1` : `1`, `3`, `5`, `7`, etc . . .

```
int* odd_array(int n) {

    int *p = new int[n];
    int odd = 1;
    for (int i = 0; i < n; i++) {
        *p[i] = odd;
        odd = odd + 2;
    }
    return p;


}
```

**5** (8%) Write a function that returns `false` if a string has no letters or digits, `true` if it does. You may use either recursion or iteration. Use `string` library functions: `bool isalpha()` and `bool isdigit()`.

```
bool not_alpha_num(string& s) {

  if (s == "") return false;
  if (isalpha(s[0]) || isdigit(s[0]))
     return true;
  else {
     return not_alpha_num(s.substr(1));
  }



}
```

**// OR**

```
if (input == "") return false;
for (int i = 0; i < s.substr) {
  if (isalpha(s[i]) || isdigit(s[i]))
     return true;
}
return false;
```

**6** (8%) Write a function that implements a standard algorithm for removing an element from an unordered vector of unique integers. Your function does not need to preserve the order of elements in the vector.

```
void delete(vector<int>& my_vector, int n) {
    int tmp = my_vector.[my_vector.size() - 1];
    for (int i = 0; i < my_vector.size(); i++) {
        if (my_vector[i] == n) {
            my_vector[my_vector.size()-1] = n;
            my_vector[i] = tmp;
            my_vector.pop_back();
        } }
```

**7** (10%) Write a function that finds the sums of the values in each row (one per row) of a two-dimensional array. Return a one-dimensional array with the sums. The returned array must persist beyond your function, without using global or static variables - use dynamic memory. Finish **`main()`** by cleaning up memory.

```
const int COLUMNS = 4;
int* find_sums(int values[][COLUMNS], int number_of_rows) {
    int * sums = new int[number_of_rows]{0, 0, 0};
    for (int i = 0; i < number_of_rows; i++) {
        for (int j = 0; j < COLUMNS; j++) {
            sums[i] = sums[i] + values[i][j];
        }
    }
    return sums;
}
```

```
int main() {
    int two_d_array[3][COLUMNS] = {{ 2, 1, 4, 9 },{ 1, 0, 2, 7 },{ 7, 3, 6,
1}};
    int* sums = find_sums(two_d_array, 3);//sum values in each of the three
rows
    cout << sums[0] << " " << sums[1] << " " << sums[2] << endl;
    delete[] sums; //deallocate dynamic memory
    sums = nullptr; //fix dangling pointer
}
```

**8** (16%)

(4%) **i** Define a simple data-only **`class SecretCode`** (with only public data members and no member functions – the book would call it a **`struct`**).

A secret code has a**n `int key`** and a **`string message`**.

```
class SecretCode
{
public:
    string message;
    int key;
};
```

(8%) **ii** Write a function that returns an encrypted message by "shifting" each character by the **`key`**: adding the **`key`** to the ASCII code of that character. To do that you will have to cast **`char`** as **`int`** and back as **`char`**. See the review sheet for casting / type conversion.

```
string encrypt(SecretCode sc)
{


    string encrypt = "";
    for (int i = 0; i < sc.message.size(); i++) {
        encrypt = encrypt + (sc.message[i] + sc.key);
    }
    return encrypt;




}
```

(4%) **iii** Write a **`main()`** function that reads a message and a key from the user input, calls your function, and displays the encrypted message.

```
int main()
{


    SecretCode sc;
    cout << "key and message: ";
    cin >> sc.x >> sc.y;
    cout << encrypt(sc);



}
```

**9** (20%)

(8%) **i** Implement a class **`Circle`**. Provide a constructor to construct a circle with a given radius, member functions **`get_perimeter()`** and **`get_area()`** that compute the perimeter and area, and a member function void **`increase_radius(double delta)`** that resizes the circle by adding a value to the radius.

```
class Circle
{
public:
    Circle(double r);
    double get_perimeter();
    double get_area();
    void increase_radius(double delta);
private:
    double radius;
};

Circle::Circle(double r);
{
    length = l;
}
```

```
double Circle::get_perimeter()
{
    return 3.14 * r * r;
}

double Circle::get_area();
{
    return 2 * 3.14 * r;
}

void Circle::increase_radius(
                    double delta);
{
    radius = radius + factor;
}
```

(12%) **ii** Write individual one line commands to accomplish the following (2 points each):

1 Create a local circle object of radius 4;

```
Circle r  = (2, 4);
```

2 Increase its radius by 5;

```
r.increase_radius(5);
```

3 Print out its perimeter and its area;

```
cout << r.get_perimeter << " " << r.get_area << endl;
```

4 Create another circle object, but this time in the dynamic memory;

```
Circle * r2 = new Circle(2, 4);
```

5 Deallocate this object's memory;

```
delete r2;
```

6 Take care of the dangling pointer.

```
r2 = nullptr;
```

## Variable and Constant Definitions

```
int cans_per_pack = 6;
const double CAN_VOLUME = 0.335;
```

*Type*, *Name*, *Initial value*

## Type Conversion: Explicit Cast

```
char c ='A';  //'A' is ASCII code 65
int x = (int)c; //explicit cast to int; x is now 65
x = x + 5;  // x is 70
char d = (char)a; //explicit cast to char;
          //d now has the ASCII value 70, which is 'F'
```

## 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

*Condition*

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

*Executed while condition is true*

*Initialization*, *Condition*, *Update*

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

*Loop body executed at least once*

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

## Conditional Statement

*Condition*

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

*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

*Return type*, *Parameter type and name*

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

*Exits function and returns result.*

*Reference parameter*

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

*Modifies supplied argument*

## Arrays

*Element type*, *Length*

```
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]
}
```

## Vectors

```
#include<vector>
vector<int> values = { 0, 1, 4, 9, 16 };

vector<string> names;

names.push_back("Ann");
names.push_back("Cindy"); // names.size() is now 2

names.pop_back(); // Removes last element

names[0] = "Beth"; // Use [] for element access
```

*Element type*

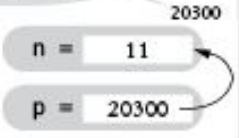
*Initial values (C++ 11)*

*Initially empty*

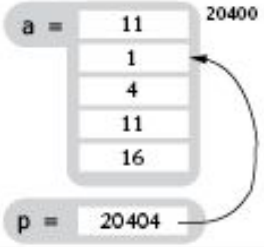
*Add elements to the end*

## 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);
```

## Static Variables

```
void increment_print() {
   static int s_value = 0; //static duration
   s_value++;
   cout << s_value << '\n';
} //s_value is not destroyed, but goes out of scope
int main() {
   increment_print(); //1
   increment_print(); //2
}
```

## Static Data Members

```
class Item {
private:
   int m_id;
   static int s_id_counter;
public:
   Item() {
      m_id = s_id_counter++;
   }
   int get_id() const {
      return m_id;
   }
};
int Item::s_id_counter = 1;
int main() { //
   Item first;
   Item second;
   cout << first.get_id();  //1
   cout << second.get_id();//2
}
```

## Range-based for Loop

```
for (int v : values)
{
   cout << v << endl;
}
```

*An array, vector, or other container (C++ 11)*

## 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) |

## 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;
}
```

## Class Definition

```
class BankAccount
{
public:
   BankAccount(double amount);
   void deposit(double amount);
   double get_balance() const;
   ...
private:
   double balance;
};

void BankAccount::deposit(double amount)
{
   balance = balance + amount;
}
```

*Constructor declaration*

*Member function declaration*

*Accessor member function*

*Data member*

*Member function definition*

## Inheritance

```
class CheckingAccount : public BankAccount
{
public:
   void deposit(double amount);
private:
   int transactions;
};

void CheckingAccount::deposit(double amount)
{
   BankAccount::deposit(amount);
   transactions++;
}
```

*Derived class*

*Base class*

*Member function overrides base class*

*Added data member in derived class*

*Calls base class member function*