**CIS330** **Name:** _____
**Winter 2020**
**Midterm Exam**
**2/5/20, 02:00PM to 3:20PM**
**Time Limit: 80 Minutes**

---

This exam contains 13 pages (including this cover page and extra sheets) and 5 questions. Total number of points is 100, excluding extra credit. This exam is printed on both sides.

You are not allowed use any books, notes, calculators, or electronic devices of any kind. **Only exception is to post a question about the exam itself on Piazza.** Write your answers **carefully** and **legibly**. Remember, **partial answers are better than no answers**.

Feel free to skip around and go back to an earlier question later. You may find it helpful to **skim over the entire exam first** and start with the easier ones, then move on to the more difficult ones. Remember to distribute your time appropriately among the questions.

**There is an extra credit question at the end - do not miss it!**
Good luck!

Grade Table (for instructor use only)

| Question | Points | Score |
|---|---|---|
| 1 | 8 | |
| 2 | 6 | |
| 3 | 26 | |
| 4 | 20 | |
| 5 | 40 | |
| Total: | 100 | |

---

# 1 The C Language

1. (8 points) Answer the following short-answer questions. (8 minutes)

   (a) (4 points) Given the following macros defined using the `#define` directive, and its usage within the `main` function, write the **exact code that will be produced after textual replacement has been done by the compiler's pre-processor**.

   For example, given `#define PI 3.1`, then the statement `int x = PI * PI;` will become `int x = 3.1 * 3.1;` after pre-processing for textual replacement.

```
#define PI 3.1
#define calcCircleArea(r) (PI * (r) * (r))
#define calcCylinderArea(r,h) (calcCircleArea(r) * h)
int main()
{
  double i = calcCylinderArea(3.0,5.0 + 1);}
}
```

   `double i = ((3.1 * (3.0) * (3.0)) * 5.0 + 1);`

   (b) (2 points) What is stored in the variable `A` after the following has been executed (on a modern 64-bit architecture)? Please provide your answer in hexadecimal.

```
int* A = 0x8000;
A += 5;
```

   0x8014

   (c) (2 points) Given the following C code snippet, what would be printed (assuming it compiles)?

```
int main(int argc, char **argv)
{
    argc++;
    for(int i = 1; i < argc; i++) {
        printf("%s ", argv[i - 1]);
    }
    return 0;
}
```

   `./a.out Doctor Who?`

   ./a.out Doctor Who?

2. (6 points) For the following questions, choose the corret answer. (7 minutes)

   (a) (3 points) Given the following piece of code:

```
int i = 3; int j = 5; int k = 1;
k = (++i) + (j++);
```

   What would be displayed if you were to execute the appropriate instructions to

```
print (i++) (++j) (k++)
print i j k
```

   A. 3 5 8
      5 7 9

   B. 5 6 10
      5 7 10

   C. 4 7 9
      5 7 10

   D. 4 7 9
      5 7 9

   E. None of the above.

   (b) (3 points) Which of the following is **not** a correct method for printing the i$^{th}$ element (where i starts from 1, and **not** 0 - e.g., 1$^{st}$ element of A would be in A[0]) of an array `A`, defined as

```
int* A = (int*) malloc(sizeof(int) * 10)
```

   A. `int* B = A;` `printf("%d\n", B[i - 1]);`

   B. `int* B = &(A[i - 2]);` `printf("%d\n", B[1]);`

   C. `int* B = A + i - 1;` `printf("%d\n", *B);`

   D. `int* B = (&A)[0];` `printf("%d\n", B[i - 1]);`

   E. All of the above are correct.

# 2 Coding in C Part 1

3. (26 points) Given a 2-D array of integers (e.g., `int** arr`) with $m$ rows and $m$ columns, implement a function that swaps `arr[i][j]` with `arr[j][i]` **in-place**. (20 minutes)

   Use the function definition:
   `void swap_arr(int** mat, int m);`

   Normally, a new 2-D array is created first (e.g., `int** output_arr`) and elements from the input array is moved to the output array one element at a time (i.e., `output_arr[j][i] = arr[i][j]`).

   **In this function, you CANNOT create a new array, and everything has to be done using the given input array (you are allowed to use a temporary variable to do a swap).**

```
void swap_arr(int** mat, int m)
{
  for(int i = 0; i < m; i++) {
    for(int j = 0; j < m; j++) {
      if(j > i) {
        int tmp = mat[i][j];
        mat[i][j] = mat[j][i];
        mat[j][i] = tmp;
      }
    }
  }
}
```

   or

```
void swap_arr(int** mat, int m)
{
  for(int i = 0; i < m; i++) {
    for(int j = i; j < m; j++) {
      int tmp = mat[i][j];
      mat[i][j] = mat[j][i];
      mat[j][i] = tmp;
    }
  }
}
```

4. (20 points) Given the following pieces of code, implement `init_2d()` and `free_2d()` functions that allocates and frees a 2-D array of **int pointers** of size `first_dim` × `second_dim`. Make sure to include the function name and its arguments in your code. (15 minutes)

```
int main()
{
    const int first_dim = 10;
    const int second_dim = 20;
    // address_array is a 2-D array of int pointers (i.e., addresses)
    int ***address_array = NULL;

    init_2d(&address_array, first_dim, second_dim);
    // Do some stuff here with the 2-D array
    free_2d(address_array, first_dim);

    return 0;
}
```

(a) (10 points) Implement `init_2d_address_array()` (8 minutes)

```
void init_2d(int ****arr, const int fdim, const int sdim)
{
  int*** tmp = (int***) malloc(sizeof(int**) * fdim);
  for(int i = 0; i < fdim; i++) {
    tmp[i] = (int**) malloc(sizeof(int*) * sdim);
  }
  *arr = tmp;
}
```

or

```
void init_2d(int ****arr, const int fdim, const int sdim)
{
  *arr = (int***) malloc(sizeof(int**) * fdim);
  for(int i = 0; i < fdim; i++) {
    (*arr)[i] = (int**) malloc(sizeof(int*) * sdim);
  }
}
```

(b) (10 points) Implement `free_2d_address_array()` (7 minutes)

```
void free_2d(int ***arr, const int fdim)
{
  for(int i = 0; i < fdim; i++) {
    free(arr[i]);
  }
  free(arr);
}
```

# 3 Coding in C Part 2

5. (40 points) Given the following code that implements a 3-D Cartesian co-ordinate: (30 minutes)

```
typedef struct cart_coord {
  int x;        // x co-ordinate for this point
  int y;        // y co-ordinate for this point
  int z;        // z co-ordinate for this point
  double mass;  // mass for this point
  double force; // force exerted on this point
} coord_t;
```

and the following main and initialization functions:

```
void generate_points(int** x, int** y, int** z, double** m, int n)
{
  int* tmp_x = (int*) malloc(sizeof(int) * n);  assert(tmp_x);
  int* tmp_y = (int*) malloc(sizeof(int) * n);  assert(tmp_y);
  int* tmp_z = (int*) malloc(sizeof(int) * n);  assert(tmp_z);
  double* tmp_m = (double*) malloc(sizeof(double) * n);  assert(tmp_m);

  for(int i = 0; i < n; i++) {
    tmp_x[i] = rand() % 1000;
    tmp_y[i] = rand() % 1000;
    tmp_z[i] = rand() % 1000;
    tmp_m[i] = 1000000 * ((1.0 * rand()) / RAND_MAX);
  }

  *x = tmp_x;
  *y = tmp_y;
  *z = tmp_z;
  *m = tmp_m;
}
```

```
double G = 6.67e-11;

int main(int argc, char** argv)
{
  int num_points = atoi(argv[1]);
  int* x_coord;
  int* y_coord;
  int* z_coord;
  double* mass;

  generate_points(&x_coord, &y_coord, &z_coord, &mass, num_points);

  coord_t** space;
  init_space(&space, x_coord, y_coord, z_coord, mass, num_points);
  n_body(space, num_points);

  free_space(space, num_points);

  free(x_coord);
  free(y_coord);
  free(z_coord);
  free(mass);

  return 0;
}
```

where `x_coord[i]`, `y_coord[i]`, `z_coord[i]`, and `mass[i]` stores the x, y, z co-ordinates and mass for point $i$.

(a) (10 points) Implement the `init_space` function that does the following. (10 minutes)

- Allocate memory for an *array of* ***pointers*** *to* `coord_t`, where the number of elements in this array is `num_points`.
- For each element of this array, allocate memory for one point (i.e., one `coord_t`).
- Copy the $i^{th}$ `x_coord`, `y_coord`, `z_coord`, and `mass` to $i^{th}$ element in this array.
- Initialize the force on each point to 0 (i.e., member `force` in `struct cart_coord`).
- Store this array in variable `space` (defined in the `main` function).
- Make sure to include the function header/definition, including all arguments.

```
void init_space(coord_t*** space, int* x, int* y, int* z, double* v, int n)
{
  coord_t** s = (coord_t**) malloc(sizeof(coord_t*) * n);
  for(int i = 0; i < n; i++) {
    s[i] = (coord_t*) malloc(sizeof(coord_t));
    assert(s[i]);
    s[i]->x = x[i];
    s[i]->y = y[i];
    s[i]->z = z[i];
    s[i]->mass = v[i];
    s[i]->force = 0.0;
  }
  *space = s;
}
```

(b) (10 points) Implement the function that calculates and returns the distance between two points in this space. (7 minutes)

The function definition should be:

```
double distance(cart_coord_t* a, cart_coord_t* b);
```

Remember that the distance between two points with co-ordinates $(x_1, y_1, z_1)$ and $(x_2, y_2, z_2)$ is given by $distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$. Also, the square root function is defined as `double sqrt(double arg)` in C.

```
double distance(coord_t* a, coord_t* b)
{
  double r = (double) ((a->x - b->x) * (a->x - b->x) +
                       (a->y - b->y) * (a->y - b->y) +
                       (a->z - b->z) * (a->z - b->z));
  double d = sqrt(r);
  return d;
}
```

(c) (10 points) Implement the function that calculates and returns the force between two points in space. (7 minutes)

The function definition should be:

```
double calc_force(coord_t* a, coord_t* b);
```

The force between points $i$ and $j$ is calculated by
$F_{ij} = G\frac{m_i * m_j}{r_{ij}^2}$
where $m_i$ and $m_j$ are the masses of point $i$ and point $j$, respectively, $r_{ij}$ is the distance between the two points, and $G$ is the gravitational constant (declared globally as `6.67e-11` in the code above). Use the `distance` function from part (b) of this question above. Note that there is no shortcut for calculating the square of a number in C (i.e., you can't do x**2 to calculate $x^2$ in C, as you can do in Python).

```
double calc_force(coord_t* a, coord_t* b)
{
  double d = distance(a, b);
  double force = G * (a->mass * b->mass) / (d * d);
  return force;
}
```

(d) (10 points) In physics, the n-body problem is the problem of predicting the individual motions of a group of celestial objects interacting with each other gravitationally. Implement the `n_body` function shown in the `main` function above. (6 minutes)

The function does the following:

- For each point (i.e., body) $i$ in `space`, calculate the force (i.e., gravitational force) between $i$ and every *other* point $j$ (i.e., excluding $i$).
- For each force calculated between $i$ and $j$, *accumulate* it to $i$'s `force` member.
- Do this for every point in `space`.
- Use the `calc_force` function implemented above in part (c).

```
void n_body(cart_coord_t** space, int n)
{
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      if(j != i) {
        double f = calc_force(space[i], space[j]);
        space[i]->force += f;
      }
    }
  }
}
```

**Extra Credit 1** [5] For the given sparse matrix, list the required data structures (i.e., arrays) for the compressed sparse row (CSR) format, and fill it with the correct values. *You do not need to write code - just draw the required arrays and fill them out.* You may use either 0-indexing or 1-indexing (but be consistent)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 6 |
| 0 | 0 | 1 | 9 | 0 |