COP3223 Final Review

UCF Study Union, Fall 2025
Jacob Melendez, Peer Tutor

Disclaimer

This resource was created by me (a student), and it was not checked/approved by a professor and was not checked for errors. These are questions that I feel will help prepare you for your upcoming final exam. These are not to act as a sole resource for your studying. Please do not only rely on this, expecting it to be enough for your exam.

For supplementary resources, check out your professor's materials and other Peer Tutors/SI Leaders' review packets. Also, be sure to review Arup Guha's Webpage, and SARC COP 3323 webcourses. Finally, feel free to reach out if you'd like pointers to anything else besides that!

Review Session

My review session is on Wednesday, April 29th from 6:00pm – 8:00pm in Garden Key in the Student Union. There is also an online access link. You'll be able to ask any questions you may have. If you can't attend, you'll be able to find the recording and solutions on the SARC Study Union web page after it ends. Hope to see you there!

Congratulations on being (almost) finished with Intro to C! Best of luck on finals!

List of Topics:

1. Printf/Scanf
2. If/Else Statements
3. Loops
4. Functions
5. Random Number Generation
6. File I/O
7. Arrays (1D/2D)
8. Strings
9. Pointers
10. Structs
11. DMA
12. Linked Lists
13. Practice Questions

Each topic will start with a bulleted summary (5-8 bullet points) of the most essential information you need to know for each topic. Then, there will be three practice questions (two multiple choice, one free response). Finally, there will be 5 free response miscellaneous practice questions which combine a mix of all of the topics seen above.

Printf/Scanf

- printf format specifiers: %d (int), %c (char), %s (string), %f (float).
- %.2f means “print float value with 2 digits after decimal”; %f prints default precision.
- scanf needs the address of variables to store input - scanf("%d", &i).
- arrays (char s[]) decay to pointer so use scanf("%s", s) without &.
- Use width limits in scanf to avoid buffer overflow (e.g., scanf("%19s", s) for a 20-char buffer).
- stdio.h provides I/O functions (printf, scanf, FILE operations); stdlib.h includes utilities (malloc, free, rand).
- printf and scanf format strings can include escape sequences (e.g., \n, \t) and formatting (width, padding, left/right alignment).

1. Which statement about scanf is correct?
A. scanf takes variable names, not addresses.
B. scanf automatically allocates memory for strings.
C. scanf("%d", &x) stores input in x because &x provides its address.
D. scanf cannot read floats.

Answer: C

2.

```
#include <stdio.h>

int main() {
   float x = 3.14159;
   printf("%.2f\n", x);
   return 0;
}
```

Output: 3.14

3. Write two lines of code:
Declare a double variable named d.
Use printf to print it with 4 decimal places.

```
double d;
printf("%.4f\n", d);
```

If/Else Statements

- Forms: if (condition) { ... }, if (condition) { ... } else { ... }, and chains with else if. else pairs with the nearest unmatched if.
- Conditions use relational operators: <, <=, >, >=, ==, !=. Remember = is assignment, == is comparison.
- Logical operators: && (and), || (or), ! (not). Short-circuiting: && stops if left is false; || stops if left is true.
- switch (expression) is useful for multi-way branching; case labels and default:. break prevents fall-through.
- switch works with integer types (including char), not floating-point or strings. Fall-through can be used intentionally but be explicit.
- Watch for truthiness: in C, 0 is false; any nonzero value is true.

1. What is the value of the expression (5 > 2 && 3 == 4)?
A. 1
B. 0
C. 3
D. Compiler error

Answer: B

2.
```
int x = 4;
if (x % 2 == 0)
   printf("even");
else
   printf("odd");
```

Output: even

3. Write a snippet that uses a switch statement to print "A" if grade == 90, otherwise printing "Other".

```
switch (grade) {
   case 90: printf("A"); break;
   default: printf("Other");
}
```

Loops

- for (initialization; condition; iteration) - (int i=0; i<n; i++). Initialization can declare loop variable.
- while (condition) repeats while condition is true; condition tested before body.
- do { ... } while (condition); tests condition after running body once (guarantees one iteration).
- break; exits the nearest loop immediately; continue; skips to the next loop iteration (runs update expression in for).
- Nested loops: outer and inner loops.
- Use loop invariants and clear termination conditions to avoid infinite loops.
- Prefer for to do index-based repetition, while or do-while when condition-driven.

1. Which loop always executes at least once?
A. for
B. while
C. do-while
D. All of them

Answer: C

2.
```
for (int i = 0; i < 3; i++) {
   printf("%d ", i);
}
```

Output: 0 1 2

3. Write a nested for loop that prints the following (3 rows):

```
*
**
***
```

```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < = i; j++) {
     printf("*");
   }
   printf("\n");
}
```

Random Number Generation

- rand() (from stdlib.h) returns a pseudo-random integer in [0, RAND_MAX]. Use srand(seed) to seed the generator - srand(time(NULL)).
- RAND_MAX is implementation-defined (commonly 32767 or larger); normalize with division if you need 0..1 floating values: rand() / (double)RAND_MAX.
- To get integers in a range [a..b] inclusive: a + rand() % (b - a + 1).
- For half-open ranges [0..n) use rand() % n.
- Use good seeding (e.g., time(NULL)) to get different sequences each run; without srand, rand() repeats same sequence each program start.

1. rand() % 10 produces values in which range?
A. 1–10
B. 0–10
C. 0–9
D. 1–9

Answer: C

2.
srand(5);
printf("%d\n", rand() % 5);

Output: Some value in 0-4

3. Write a line of code that generates an integer between 10 and 20 (inclusive).

int r = rand() % 11 + 10;

Functions

- Functions encapsulate reusable logic: declaration/prototype, definition, and call. Prototype example: int add(int a, int b); definition includes body.
- Return type indicates what the function yields; void means no returned value. Parameters are local variables (copies) inside function.
- C uses pass-by-value: primitives passed into a function are copied (modifying parameter does not change caller's variable).
- To modify caller variables, pass pointers (address) and dereference inside function (void inc(int *p)), i.e., pass-by-reference via pointers.
- Function scope: local variables live on the stack and vanish after return; static variables persist across calls.
- Header files declare prototypes so the compiler can check usage before definition.

1. What does C use when passing primitive types to functions?
A. Pass-by-reference
B. Pass-by-value
C. Pass-by-copy-reference
D. Pass-by-pointer by default

Answer: B

2.
```
int add(int x, int y) {
   x = x + 1;
   return x + y;
}

int main() {
   int a = 3, b = 4;
   printf("%d", add(a, b));
}
```

Output: 8

3. Write a function prototype and definition for int square(int x) that returns x * x.

```
int square(int x);

int square(int x) {
   return x * x;
}
```

File I/O

- FILE *fopen(const char *pathname, const char *mode) — open file: modes "r","w","a","r+","w+","a+". Returns NULL on error.
- int fclose(FILE *stream) — close file and flush buffers; returns 0 on success, EOF on error.
- int fprintf(FILE *stream, const char *format, ...) — formatted output to file (like printf but takes a FILE*).
- int fscanf(FILE *stream, const char *format, ...) — formatted input from file (like scanf but reads from FILE*); check return value for number of items read.
- char *fgets(char *s, int size, FILE *stream) — read up to size-1 chars or newline; stores null terminator; returns NULL at EOF or error.
- int fputs(const char *s, FILE *stream) / int puts(const char *s) — write a string (no formatting) to file / stdout (puts appends newline).
- Always check return values (NULL, 0, EOF) and close files to avoid resource leaks.

1. What does fopen("data.txt", "r") return on failure?
A. EOF
B. -1
C. NULL
D. Empty file

Answer: C

2. Which line contains the error?
FILE *fp;
fp = fopen("nums.txt", "r");
int x;
fscanf(fp, "%d", x);

A. Line 1
B. Line 2
C. Line 3
D. Line 4

Answer: D

3. Write two lines:
Open "out.txt" for writing.
Write "Hello" to it using fprintf.

FILE *fp = fopen("out.txt", "w");
fprintf(fp, "Hello");

Arrays

- C arrays are zero-indexed: arr[0] is first element, arr[n-1] last. Access out-of-bounds is undefined behavior.
- 1D declaration: int a[10]; — contiguous memory of 10 ints. You can iterate with a for loop: for (int i=0; i<10; i++).
- Passing arrays to functions: array name decays to pointer to first element (int *), so a function receiving an array sees a pointer; size must be passed separately if needed.
- sizeof trick: for a local array sizeof(a)/sizeof(a[0]) gives the number of elements; this does not work if a is a pointer parameter.
- 2D arrays: int m[3][4]; accessed by m[i][j]. C stores 2D arrays row-major (rows stored consecutively).
- When passing 2D arrays to functions, you either specify the second dimension (int fn (int m[][4], int rows)) or use pointer-to-pointer / dynamic allocation.
- Multidimensional arrays are just nested arrays; nested loops are the usual way to iterate. Watch index order for cache efficiency (row-major -> iterate rows then columns).

1. Which is the correct max index for the 2D array declaration int a[3][4];?
A. a[3][4]
B. a[2][3]
C. a[4][3]
D. a[1]

Answer: B

2.
```
int a[5] = {1, 2, 3, 4, 5};
printf("%d", a[2]);
```

Output: 3

3. Write a loop that sums all elements in a 1D array int arr[5].

```
int sum = 0;
for (int i = 0; i < 5; i++) sum += arr[i];
```

Strings

- C strings are arrays of char terminated by the null character '\0'. Example declarations: char s1[] = "hi"; or char *s2 = "hello"; (s2 points to string literal).
- printf("%s", s) prints until '\0'. scanf("%s", s) reads a whitespace-delimited word (no & for s since s decays to char *). Use width specifier to avoid overflow.
- string.h essentials: size_t strlen(const char *s) (length excluding '\0'), char *strcpy(char *dest, const char *src), int strcmp(const char *a, const char *b) (0 if equal, - if a < b, + if a > b).
- Iterating: treat string like char array: for (int i=0; s[i] != '\0'; i++) or pointer iteration for (char *p = s; *p; p++).
- String literals are typically stored in read-only memory — modifying char *lit = "x" is undefined; use an array char lit[] = "x".

1. Which function returns the length of a string excluding the null terminator?
A. strcmp
B. strcpy
C. strlen
D. size

Answer: C

2.
char s[] = "hi";
printf("%s", s);

Output: hi

3. Write a line of code that copies "hello" into char name[10]; using strcpy.

strcpy(name, "hello");

Pointers

- Pointer holds the memory address of a value: declaration int *p; (p points to an int). Use & to get address (&x) and * to dereference (*p to read/write).
- Pointers enable functions to modify caller values (pass-by-reference), build dynamic data structures, and handle arrays and strings efficiently.
- Pointer arithmetic: for int *p, p+1 moves by sizeof(int) bytes to the next element; pointer arithmetic respects pointed-to type size.
- Use NULL to indicate a pointer that points nowhere; always initialize pointers before use.
- void * is a generic pointer type used by memory functions (malloc) — cast to the appropriate type when using.
- In structs, if you have struct S *sp; use sp->field (shorthand for (*sp).field) to access members through a pointer.
- Dangling pointers: don't use pointers to freed memory; after free(p), set p = NULL to avoid accidental reuse.

1. What does the expression *p mean for a pointer p?
A. Assign a pointer
B. Access the value stored at the address
C. Create a new pointer
D. Multiply by p

Answer: B

2. Which line is the issue?
int *p;
int x = 5;
*p = x;

A. Line 1
B. Line 2
C. Line 3
D. No issue

Answer: C

3. Write two lines:
Declare int x = 10;
Declare a pointer p pointing to x.

int x = 10;
int *p = &x;

Structs

- struct groups related fields: struct Point { int x; int y; }; — lets you create compound data types.
- Declare variables: struct Point p; or with typedef struct Point Point; you can write Point p; (typedef creates an alias).
- Access fields with dot p.x for struct variables, and pptr->x for pointers to struct (pptr->x is the same as (*pptr).x).
- Useful for modeling records, objects (data only), and building linked data structures (nodes with next pointers).
- You can nest structs and include arrays inside a struct. Be mindful of padding and sizeof.
- Pass structs to functions by value (copy) or pass pointer to struct to avoid copying and to modify members in-place.

1. Which operator is used to access a struct member through a pointer?
A. .
B. ->
C. *.
D. &.

Answer: B

2.
```
struct P { int x; int y; };
struct P p = {3, 4};
printf("%d", p.y);
```

Output: 4

3. Write a struct Student with two fields: an int id and a float gpa.

```
struct Student {
   int id;
   float gpa;
};
```

Dynamic Memory Allocation (DMA)

- void *malloc(size_t size) allocates size bytes and returns pointer to the block (uninitialized contents). Returns NULL on failure.
- void *calloc(size_t nmemb, size_t size) allocates memory for an array of nmemb elements, each size bytes, and zero-initializes the block.
- void *realloc(void *ptr, size_t new_size) resizes an allocated block (may move it); returns new pointer (or NULL on failure, leaving original unchanged).
- free(void *ptr) releases previously allocated memory; after free, pointer becomes invalid (dangling). Set pointer to NULL if keeping it around.
- sizeof(type) and sizeof expression evaluate number of bytes needed — use malloc(n * sizeof *p) or malloc(n * sizeof *p) to avoid type mistakes.
- Memory leaks occur when allocated memory is never freed; dangling pointers occur when you free memory that is still referenced. Use matching malloc/free.
- Always check returned pointer from allocation before use. For arrays of structs, allocate n * sizeof(struct Type).

1. What does malloc return?
A. A block of memory filled with zeros
B. A pointer to uninitialized memory
C. A pointer to a file
D. A reference variable

Answer: B

2. Where is the error?

```
int *p = malloc(5 * sizeof(int));
for (int i = 0; i < 5; i++)
   p[i] = i;
free(p);
free(p);
```

A. Line 1
B. Line 2
C. Line 4
D. Line 5

Answer: D

3. Write one line allocating space for 10 floats using calloc.

```
float *p = calloc(10, sizeof(float));
```

Linked Lists

- A singly-linked list node typically: struct Node { int val; struct Node *next; }; Head pointer points to first node; next == NULL indicates end.
- Doubly-linked list adds prev pointer: struct DNode { int val; struct DNode *prev, *next; }; makes reverse traversal and O(1) deletion easier.
- Tail pointer optimization: keep tail pointer to last node to allow O(1) append; without it appending is O(n). Maintain both head and tail.
- Basic operations: traversal (follow next pointers), insertion at head (fast), insertion at tail, insertion after a node, deletion (need previous pointer unless deleting head).
- For singly list deletion of a node other than head, you either track prev during traversal or use a double pointer technique (pointer-to-pointer).
- Freeing: each node must be freed when removed to avoid memory leaks. For full list destruction, iterate and free every node.
- Pros/Cons: linked lists allow fast insert/delete in middle but have overhead per node (pointer storage) and poor locality compared to arrays.

1. In a singly linked list, what does the next pointer typically hold?
A. The previous node
B. The index of the node
C. A pointer to the next node
D. Nothing; it's unused

Answer: C

2. What is the error?
struct Node { int x; struct Node *next; };
struct Node *head = NULL;
printf("%d", head->x);

A. struct definition
B. head should be int
C. Dereferencing NULL pointer
D. Missing semicolon

Answer: C

3. Write 3 lines:
Declare a struct Node with an int data and a next pointer. Create a node using malloc. Set its data to 5.

struct Node { int data; struct Node *next; };
struct Node *node = malloc(sizeof(struct Node));
node->data = 5;

Practice Questions

1. Use nested loops to find the row and column with the highest total.

Write a function that:
Takes a 2D array of rows × 10 integers.
Computes the sum of each row and the sum of each column.
Determines which row has the largest total and which column has the largest total (use if to compare as you loop).
Prints both using formatted output (e.g., "Row %d has the highest total: %d\n").

void findMaxTotals(int arr[][10], int rows);

Answer:

```
void findMaxTotals(int arr[][10], int rows) {
    int maxRow = 0, maxRowSum = -1;
    int maxCol = 0, maxColSum = -1;

    // Row totals
    for (int r = 0; r < rows; r++) {
        int sum = 0;
        for (int c = 0; c < 10; c++) sum += arr[r][c];
        if (sum > maxRowSum) {
            maxRowSum = sum;
            maxRow = r;
        }
    }

    // Column totals
    for (int c = 0; c < 10; c++) {
        int sum = 0;
        for (int r = 0; r < rows; r++) sum += arr[r][c];
        if (sum > maxColSum) {
            maxColSum = sum;
            maxCol = c;
        }
    }

    printf("Row %d has max total %d\n", maxRow, maxRowSum);
    printf("Col %d has max total %d\n", maxCol, maxColSum);
}
```

2. Add two money structs (dollars + cents) while properly handling overflow.

Write a function that:
Takes two Money structs by pointer (a, b).
Adds their fields.
If the total cents exceed 100, convert to dollars (using an if statement).
Stores the result in the result struct (also passed by pointer).

Struct Definition to Use

```
typedef struct {
    int dollars;
    int cents;
 } Money;

void addMoney(const Money *a, const Money *b, Money *result);
```

Answer:

```
void addMoney(const Money *a, const Money *b, Money *result) {
    result->dollars = a->dollars + b->dollars;
    result->cents = a->cents + b->cents;

    if (result->cents >= 100) {
        result->dollars += result->cents / 100;
        result->cents %= 100;
    }
}
```

3. Read a number from the first line, then read that many integers into an array.

Write a function that:
Opens a file for reading.
Reads the first line as an integer N.
Reads the next N integers, storing them into arr.
Stop reading if N > maxSize.
Return the number of integers actually stored.

```
int loadValues(const char *filename, int arr[], int maxSize);
```

Answer:

```
int loadValues(const char *filename, int arr[], int maxSize) {
    FILE *fp = fopen(filename, "r");
    if (!fp) return 0;

    int n;
    fscanf(fp, "%d", &n);

    int count = (n > maxSize) ? maxSize : n;
    for (int i = 0; i < count; i++) {
        fscanf(fp, "%d", &arr[i]);
    }

    fclose(fp);
    return count;
}
```

4. Use different functions on an array of strings.

Write a function that:
Takes a 1D array of strings (words) and its size.
Takes a target string.
Returns how many strings in the array match exactly (strcmp == 0).
Use a for loop, strcmp, and %s conventions.
List the library that must be imported for these functions.

```
int countMatches(char words[][50], int size, const char *target);
```

Answer:

```
#include <string.h>

int countMatches(char words[][50], int size, const char *target) {
    int count = 0;
    for (int i = 0; i < size; i++)
        if (strcmp(words[i], target) == 0)
            count++;
    return count;
}
```

5. Allocate memory, copy a string, and uppercase it (A - Z).

Write a function that:
Dynamically allocates a new string using malloc.
Allocates exactly enough space using strlen(input) + 1.
Copies the input string into the new memory using strcpy.
Converts lowercase letters to uppercase using pointer iteration.
Returns the newly allocated pointer.
List the library that must be imported for these functions.

```
char* allocateAndUpper(const char *input);
```

Answer:

```
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

char* allocateAndUpper(const char *input) {
    char *out = malloc(strlen(input) + 1);
    strcpy(out, input);

    for (char *p = out; *p; p++)
        if (*p >= 'a' && *p <= 'z')
            *p = *p - 32;

    return out;
}
```