History & Language Paradigms

1. Explain the relationship between machine language, assembly language, and high level languages.
   Machine language is a set of instructions executed as-is by the computer's CPU. This is considered the lowest level representation of a computer program. While it is possible to program directly in machine code, it is highly tedious and error prone, making higher level languages favorable. Writing machine code is typically only done when troubleshooting a system or when implementing extreme optimization.

   Assembly language is the next step up from machine language and usually has a near 1:1 mapping from assembly code to the architecture's machine code instructions. *Assembly languages are specific to a computer's architecture* – this is because different architectures have different sets of hardware-supported commands. Programming in assembly language (and lower) is commonplace in embedded systems work.

   Finally, **high level languages** are generally designed to be portable across many different architectures. High level languages are designed to be human-readable and abstract away some of the low-level details of programming. Some examples of high level languages are C/C++, Python, and Java.

2. (a) List a difference between imperative/procedural programming and object-oriented programming.
   In procedural programming, the program is a flat collection of global functions and variables. In object-oriented programming, functions and variables are grouped together into class types, which integrate both data and behavior, and can be encapsulated from other classes.

   (b) List a difference that functional programming has from procedural or object-oriented programming.
   In functional programming, functions return values based solely on their input, and do not affect the state of any other data. Rather than programs being composed of statements executed in sequence, programs are composed of function applications which produce the desired output.

Basic C

3. Examine the following program:

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    int number = atoi(argv[1]);      // atoi parses int from str
    int found;
    do {
        int i;
        found = 1;
        --number;
        for (i = 2; i < number; ++i) {
            if (!(number % i)) {
                found = 0;
                break;
            }
        }
    } while (!found);
    printf("%d", number);
    return 0;
}
```

(a) What does this program do? What is the output of `./a.out 10`? Step through the program execution if it helps.

This program finds the largest prime number less than the first command line argument. `./a.out 10` sets `number` = 10.

For each iteration of the do-while loop, `number` is decremented, then the for-loop tests all integers between 2 and `number-1` inclusive for something that divides evenly into `number`. If such a value exists, it sets `found` = 0 and breaks out of the for-loop, but continues through the outer do-while loop. Otherwise `found` is still set to 1, which means `!found` is 0, so it breaks out of the do-while loop, prints the answer, and returns.

For input 10, the answer 7 is printed.

(b) Why can we write `if (!(number % i))` instead of `if (number % i == 0)`? (These statements are equivalent.)

If statements in C check whether their condition is not equal to 0, rather than if they are equal to a dedicated "true" value. If `number % i` is 0 (that is, `number` is a multiple of `i`), then `!(number % i)` will equal 1, which is not equal to 0, so the if condition will be satisfied. Comparison operators all produce 0 or 1, so here, `number % i == 0` would evaluate to `0 == 0`, which evaluates to 1.

(c) Rewrite the code using while loops in place of the do-while loop and the for loop.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    int number = atoi(argv[1]);      // atoi parses int from str
    int found = 0;
    while (!found) {
        int i;
        found = 1;
        --number;
        i = 1; // Start one below the desired initial value, as i will be immediate
        while (i < number) {
            ++i;
            if (!(number % i)) {
                found = 0;
                break;
            }
        }
    }
    printf("%d", number);
    return 0;
}
```

Note that we place the increment statement at the top of the loop body because if the loop body contained a `continue` statement, the increment statement will always be executed, unlike if the increment statement were placed at the bottom. Because this code does not include a `continue` statement, initializing `i` to 2 and placing the increment statement at the bottom of the loop body will also produce the correct behavior.

I/O

4. Write a program that searches a file for a provided number on `stdin`. Print out any errors on `stderr`. Example:

```
$ fileSearch file.txt
> 234
found: 234
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    if(argc != 2) {
        fprintf(stderr, "Usage: fileSearch <filename>");
        exit(EXIT_FAILURE);
    }
    FILE* handle = fopen(argv[1], "r");
    if(!handle) {
        perror("fopen failed:");
        exit(EXIT_FAILURE);
    }
    int input = 0;
    scanf("\%d", &input);
    int check = 0;
    while(fscanf(handle, "\%d", &check) != EOF) {
        if(check == input) {
            printf("found: \%d", input);
            exit(EXIT_SUCCESS);
        }
    }
    fclose(handle);
    printf("\%d not found", input);
    return EXIT_SUCCESS;
}
```

5. (a) The following program is intended to read a text file and outputs (as binary data) the number of characters (including new lines) in each line to a file. However, there is a problem with the code the way it is currently written. Find the problem and explain how to fix it. Note: The code is compiled and linked using GCC.

```
1  #define _GNU_SOURCE
2
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main(void) {
7      FILE* inputfile = fopen("input.txt", "r");
8      if(!inputfile) {
9          perror("fopen failed for inputfile:");
10         exit(EXIT_FAILURE);
11     }
12     int numchars = 0;
13     char* line = NULL;
14     FILE* outputfile = fopen("output.txt", "w");
15     if(!outputfile) {
16         perror("fopen failed for outputfile:");
17         fclose(inputfile);
18         exit(EXIT_FAILURE);
19     }
20     while((numchars = getline(line, 1024, inputfile)) != -1) {
21         fprintf(outputfile, "%d\n", numchars);
22     }
23     fclose(inputfile);
24     fclose(outputfile);
25     return EXIT_SUCCESS;
26 }
```

The signature of `getline` is `ssize_t getline(char** lineptr, size_t* n, FILE* stream)`. If `*lineptr` is `NULL` and `*n` is 0, then `getline` will dynamically allocate a C string containing the next line of `stream` (including the new line character), and set `*lineptr` to this C string, and `*n` to the size of this new stream. It returns the size of the string read. If the `*lineptr` given to the function is not `NULL`, then it expects `*n` to be the length of `*lineptr`, and it will attempt to use `*lineptr` to hold the next line if it can fit, otherwise it will use `realloc` to allocate enough space for it to fit. This program does not use `getline` correctly, and causes a type error when compiling. See the answer of the next question for how to fix this problem.

(b) Rewrite the code so that it outputs the data using binary streams instead.

```c
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE* inputfile = fopen("input.txt", "r");
    if(!inputfile) {
        perror("fopen failed for inputfile:");
        exit(EXIT_FAILURE);
    }
    ssize_t numchars = 0;
    char* line = NULL;
    FILE* outputfile = fopen("output.txt", "wb");
    if(!outputfile) {
        perror("fopen failed for outputfile:");
        fclose(inputfile);
        exit(EXIT_FAILURE);
    }
    size_t n = 0;
    while((numchars = getline(&line, &n, inputfile)) != -1) {
        int towrite[1] = { numchars };
        if(fwrite((void*)towrite, sizeof(int), 1, outputfile) != 1) {
            perror("fwrite into outputfile failed:");
            fclose(inputfile);
            fclose(outputfile);
            exit(EXIT_FAILURE);
        }
    }
    free(line);
    fflush(outputfile);
    fclose(inputfile);
    fclose(outputfile);
    return EXIT_SUCCESS;
}
```

C Preprocessor

6. Give an example of a header guard for a header file named `linkedlist.h`.

```c
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

<code>

#endif
```

7. Given the following, what is output of this program?

```
#define THING1 40
#define THING2 32
#define THING4 i

#if THING1 < THING2
#define THING3 5
#elif THING2 < THING1
#define THING3 6
#else
#define THING3 7
#endif

int main() {
    int THING4 = THING3;
    printf("%d", i);
}
```

6

Threads & Processes

8. Describe the differences between programs, processes, and threads.

Program - instructions on disk, static

Process - the executing program

Thread - instruction execution stream of a process

Makefiles

9. Consider the following makefile:

```
1  CFLAGS := -std=c99 -Wall -Wextra
2  me: me.o
3      $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
4  calc: calc.o real.o
5      $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)
6  calc.o: calc.c
7      $(CC) $(CFLAGS) -c $<
8  real.o: real.c
9      $(CC) $(CFLAGS) -c $<
10 .PHONY: clean
11 clean:
12     $(RM) me calc *.o
```

   (a) Ralph is annoying. One day, when Ralph makes a particularly unreasonable demand, his friend loses it and shouts, "Why don't you make me???" Always one to take things literally, Ralph pops open his favorite Bourne-compatible shell and types: `make me`. However, he is confronted with the message: `make: 'me' is up to date`.

   Explain the meaning of this message. Which (if any) of the relevant files are now located in Ralph's directory? What numeric values did `Make` compare before outputting this message?

   This output indicates that `Make` did not need to rebuild the `me` executable because:

   - the files `me.o` and `me` both existed already
   - and the modification timestamp of `me.o` was earlier than that of `me`.

   After the command completes, both files are still present in Ralph's directory.

   (b) Ralph is childishly proud of himself, but everyone just groans and tells him to "get real". Coincidentally, Ralph has a library providing functions for working with reals, as well as a calculator program to test the functionality. Ready to win another trivial victory, Ralph types `make clean` and then `make calc`.

   List the exact sequence of commands that are executed as a result of this new invocation.

   Because the `clean` target eliminated all the object files and executables, Make will decide it needs to rebuild everything:

```
$ cc -std=c99 -Wall -Wextra -c calc.c
$ cc -std=c99 -Wall -Wextra -c real.c
$ cc -o calc calc.o real.o
```

Arrays & Strings

10. (a) What is the issue with the code below? What is the result of the issue?

```
1  int i, x;
2  int arr[10];
3  for (i=0; i < 10; ++i)
4  {
5      scanf("%d", &x); // Reads a number from stdin and stores it in x
6      arr[i] = x;
7  }
8
9  for (i=10; i >= 0; --i)
10 {
11     printf("%d\n", arr[i]);
12 }
```

The second for loop starts at 10, not 9. `arr[10]` is out of bounds, but since C doesn't check for this the behavior is undefined. It might not actually crash at run time, it might just get whatever happens to be in memory at that location. *Scary.*

(b) What is the intended behavior of the program?

It reads in 10 numbers from stdin and prints them in reverse order.

11. (a) If you want to store the string ``Hello, world!'' in the `str` variable in the following code, what is the minimum acceptable value of `n`? Why?

```
int n;
...
char str[n];
```

`n` must be least 14. There are 13 characters in the string, and one extra place is needed for the null terminator.

(b) What is the character literal for the null terminator?

`'\0'`

Pointers, Structures, and Dynamic Allocation

12. Consider the following statement in a larger program:

```
int* x = (int*) malloc(20); //create an array of 20 ints
```

After testing the program, you notice that the values of $x[5], x[6], \cdots, x[19]$ keep changing unexpectedly.

(a) Why is this?

(b) What should the statement actually be?

The argument to `malloc` should be the number of bytes to allocate, not the number of elements. Since ints are usually 4 bytes long, the array only has enough room to hold 5 ints ($20/4 = 5$).

A proper (and portable) statement should be: `int* x = (int*) malloc(sizeof(int)*20);`

13. (a) Given the following:

```
struct Point {
    char label;
    double x;
    double y;
};
```

What (specifically) happens when the following command is executed?

```
struct Point *newPoint = (struct Point *) malloc( sizeof(struct Point) );
```

The `malloc` command allocates space in memory for the entire `Point` struct. Given the known sizes of `char` and `double` (x and y respectively), enough space to hold one `char` and two `doubles` are set aside in memory. Then the memory address of the newly-allocated space is returned and assigned to the `struct Point*`.

(b) Let's add some more information:

```
typedef struct {
    struct Point p1;
    struct Point p2;
    struct Point p3;
} Triangle;
```

What (specifically) happens when the following command is executed?

```
Triangle *tri = (Triangle *) malloc( sizeof(Triangle) );
```

The same thing happens, but now we need to know the size of `Triangle` from the last question. Based on the definitions provided, this means allocating exactly enough space for three `Point` structs. If the size of a `Point` struct is `X` bits, then exactly `3X` bits will be allocated in memory. Then the memory address is returned as before.

14. (a) What does the following function do?

```
1  int foo(int n, int *arr, int **bestp) {
2      int *start;
3      int *end;
4      int best = 0;
5      *bestp = arr;
6
7      for (start = arr; start < arr + n; ++start) {
8          for (end = start; end < arr + n && *end == *start; ++end);
9          if ((end - start) > best) {
10             best = (end - start);
11             *bestp = start;
12         }
13         start = end - 1;
14     }
15     return best;
16 }
```

Finds the longest sequence of identical integers in the given array. Returns the length of the sequence and stores the pointer to the start of the sequence in `bestp`.

(b) Make a memory map of `foo`. Use the first value set to each variable in the map.

Stack: n, arr, bestp, start, end, best

Heap: *arr (size n), *bestp (size int)

start → arr

end → start

bestp → arr

(c) Given the following code in `main`, write code that calls our `foo` function from above and prints the result.

```
int main(int argc, char **argv)
{
    int n, i;
    int arr[] = {1, 1, 1, 2, 2, 2, 5, 5, 5, 5};

    // Write your code here.

     int *res = NULL;
     n = sizeof(arr) / sizeof(arr[0]);

     n = foo(n, arr, &res);

     for (i=0;i<n;++i)
     {
         printf("%d, ", res[i]);
     }

     puts("");


    return 0;
}
```

15. The following program compiles.

(a) Will the program crash at run time? If so, on which line will it crash?

```
1  #include <stdlib.h>
2  int main(int argc, char **argv)
3  {
4      int *x = NULL;
5      int *y = NULL;
6      int *z = NULL;
7
8      x = (int *) malloc(sizeof(int) * 10);
9      y = (int *) malloc(20);
10     x = (int *) malloc(sizeof(char) * 50);
11
12     free(x);
13     free(y);
14     free(z);
15
16     return 0;
17 }
```

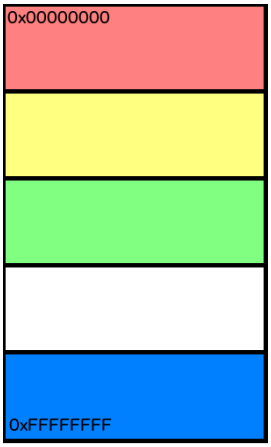
No, it runs and terminates normally. `free` will not do anything if the pointer passed to it is `NULL`.

(b) What tool can you use to find memory leaks? What options would you use?
`valgrind --leak-check=full <program name>`

(c) What output would you get from part b? Is there a memory leak? (If so, where?)
Something similar to "40 bytes lost". Yes, the first call to `malloc` is never freed.

Memory Layout

16. Label the sections of the program in memory and describe what is stored in each.
    Your options are: `stack, data, heap, unallocated space, text`.



`text` – machine language version of the source code

`data` – global variables

`heap` – dynamically allocated storage during program execution

(extra space for the heap and stack to use as needed)

`stack` – local variables and the function-calling sequence

Modular Design

17. For each of the following snippets of code, state whether it should be placed in the header file (`.h`) or the source file (`.c`).

(a)
```
struct Point
{
    double x;
    double y;
};
```
Either. If the fields of the type need to be accessed from other source files, it must be defined in the header; otherwise, it only needs to be defined in the source file and possibly forward-declared or `typedef`'d in the header.

(b)
```
int main(int argc, char **argv)
{
    int x = run_some_function();
    printf("%d\n", x);
    return 0;
}
```
Source

(c)
```
int do_something_and_return(int x, int y)
{
    if (y != 0)
        return x*y + (x/y);
    else
        return x*y;
}
```
Source

(d)
```
int do_something_and_return(int x, int y);
void do_something(int x, int y);
```
Header

Abstract Data Types

18. Generally speaking, ADTs are easier to define and work with in procedural languages like C, as opposed to object-oriented languages like Java or C#. (**True** or **False**). Explain your answer.

False. C lacks object-oriented features that streamline the creation and use of ADTs in the language. C doesn't have access modifiers (e.g., `private` or `protected`), so hiding the underlying implementations is tougher, and will often involve the use of pointers.

19. What is a *primitive data type*? Provide 3 examples of primitive data types in C.

Primitive data types are supported by the language itself, as opposed to ones that are built from a combination of other objects (*structured*), and ones that are extensions to the language (*user-defined*). Examples of primitive data types include: `int`, `float`, `char`, `double`, `void`.

20. Write a program that defines the structure of a queue:

    - The queue ADT is described by its size (an `unsigned int`) and an array representing the queue's contents.
    - The queue is built on an array of a specified size (think `#define`), the elements of which are each described by their value (any data type).
    - The queue's size describes the number of occupied indices in the array; in other words, the number of elements present within the queue.
    - The queue will support the following operations:
        - A function which returns an `int` representing whether or not the queue is empty, `int isEmpty(QueueADT q)`.
        - A function which adds a new element (new data) onto the end of the queue, `void enqueue(QueueADT q, void* data)`.
        - A function which removes the first element in the queue and returns it, `void* dequeue(QueueADT q)`.
        - A function which returns an `unsigned int` representing the size of the queue, `unsigned int size(QueueADT q)`.

```c
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#define QUEUE_SZ 10

typedef struct queue {
    unsigned int size;
    void *contents[ QUEUE_SZ ];
} *QueueADT;

/* create, malloc and return a new queue */
QueueADT createQueue(void) {
    QueueADT q = (QueueADT)malloc(sizeof(struct queue));
    q->size = 0; /* initialize the queue's size to 0 */
    return q;
}

/* return whether or not the queue is empty based on its size */
int isEmpty(QueueADT q) {
    assert(q != NULL);
    return ( q->size == 0 );
}

/* push element onto end of the queue */
void enqueue(QueueADT q, void *data) {
    assert( q != NULL ); assert( q->size < QUEUE_SZ );
    q->contents[q->size++] = data;
    return;
}

/* remove and return the data of the first element of the queue */
void* dequeue(QueueADT q) {
    assert(q != NULL); assert(q->size > 0);
    void* ret = q->contents[0];
    q->size--;
    short int i = 0;
    while ( i < q->size ) {
        q->contents[i] = q->contents[i+1];
        ++i;
    }
    return ret;
}

/* return the size of the queue */
unsigned int size(QueueADT q) {
    assert(q != NULL);
    return q->size;
}
```

Advanced C Features

21. (a) Define a union, `typedef`ed to the name `printableData`, that can hold any of: an seven-character long string, a `double`, or an `int`.

```
typedef union {
    char stringData[8]; // 8 = 7 + NUL character
    double doubleData;
    int intData;
} printableData;
```

(b) An idiom found in some C programs is the *tagged union*.

```
typedef unsigned char tag_t;

// could also use an enum
const tag_t STRING_TAG = 0;
const tag_t DOUBLE_TAG = 1;
const tag_t INT_TAG = 2;

typedef struct {
    printableData data;
    tag_t tag;
} taggedData;
```

In addition to the actual data, the tagged union stores a tag whose value corresponds to the type of data being stored. Because of this, we can tell which of the three types of values are being stored in the `data` field.
Write a function that takes an array of `taggedData` and prints out the data each element holds, each on its own line. Also take a `count` parameter. Assume all necessary headers have been included. (*Hint:* Use a series of `if`/`else if` statements to properly print each type of data that can be stored.)

```
void printAll(taggedData* mixedArray, int count) {
    int i;
    for(i = 0; i < count; ++i) {
        if(mixedArray[i].tag == STRING_TAG) {
            printf("%s\n", mixedArray[i].data.stringData);
        } else if(mixedArray[i].tag == DOUBLE_TAG) {
            printf("%f\n", mixedArray[i].data.doubleData);
        } else if(mixedArray[i].tag == INT_TAG) {
            printf("%d\n", mixedArray[i].data.intData);
        }
    }
}
```

(c) Suppose that we want to be able to process this data in other ways besides just printing to standard out. We can use function pointers to write a more generic `processAll` function. Suppose the following struct is defined:

```
typedef struct {
    void (*processString)(char*);
    void (*processDouble)(double);
    void (*processInt)(int);
} processingFunctions;
```

Define a `processAll` function that takes an array of `taggedData`, a `count` parameter, and a `processingFunctions` value, and uses the functions pointed to by the struct to process each element of the array, rather than using `printf` as was used in the `printAll` function.

```
void processAll(taggedData* mixedArray, int count, processingFunctions procs) {
    int i;
    for(i = 0; i < count; ++i) {
        if(mixedArray[i].tag == STRING_TAG) {
            (*procs.processString)(mixedArray[i].data.stringData);
        } else if(mixedArray[i].tag == DOUBLE_TAG) {
            (*procs.processDouble)(mixedArray[i].data.doubleData);
        } else if(mixedArray[i].tag == INT_TAG) {
            (*procs.processInt)(mixedArray[i].data.intData);
        }
    }
}
```

22. Suppose we have the following bit field representing a player in a ripoff of the Halo video game series.

```
struct player {
    unsigned int is_alive:1;
    unsigned int team_color:1;
    unsigned int weapon_type:2;
    unsigned int ammo_remaining:4;
};
```

However, you need to store these players as 8 bit `unsigned char` values. You decide to store `ammo_remaining` in the lowest 4 bits of the `unsigned char`, `weapon_type` in the next 2 bits, `team_color` in the next bit, and `is_alive` in the highest bit. Write a function that takes a `player` struct and returns its `unsigned char` representation under the aforementioned rules.

```
unsigned char serializePlayer(struct player player1) {


    // can also do a single return and `or` the values in one expression
    unsigned char data = 0;
    data |= player1.ammo_remaining;
    data |= player1.weapon_type << 4;
    data |= player1.team_color << 6; // 4 + 2
    data |= player1.is_alive << 7; // 4 + 2 + 1
    return data;


}
```

OS-Level I/O

23. Explain the relationship between the FD table, the open file table, and the I-node table. Which tables are in user space (a.k.a. private kernel space) and which are in system space (a.k.a. shared kernel space).

    Each process has its own FD table, which are indexed by the file descriptors returned by functions like `open`. Among other things, they hold a file pointer which refers to an entry in the open file table. The open file table is shared by all processes and holds data about all currently open files. A new entry is created in the open file table each time `open` is called by a C program, even if that same file is currently open in another process. The data it holds includes the current offset, as well as a pointer to an entry in the I-node table. The I-node table is also shared by all processes, but it has exactly one entry for each file in the file system, whether that file is open or not; the data stored in the table includes things like file permissions. The FD table is in user space, while the open file table and the I-node table are in system space.

24. You create a file named `sliding.txt` with the following contents (there are newlines between each line, but not at the end of the file):

    ```
    >>>>v
    >>>>v
    >>>>v
    >>>>v
    >>>>v
    ```

    However, your no-good friend sneakily changes one of the lines to

    ```
    ^<<<<
    ```

    and you don't know which line it is. Write a program that fixes the file, but only writes 5 characters (it can read the whole file, however). Do not worry about error handling. (*Hint:* You can use the `strstr` function in `<string.h>`, which takes two strings, searches for the second string inside the first, and returns the `char*` location within the first string where the second string was found. You may also need pointer arithmetic.)

```c
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main(void) {
    char buffer[30]; // number of characters in file + 1
    int fd = open("sliding.txt", O_RDWR);
    read(fd, buffer, 29);
    buffer[29] = '\0'; // for use in string functions
    char* modifiedLocation = strstr(buffer, "^<<<<");
    int offset = modifiedLocation - buffer;
    lseek(fd, offset, SEEK_SET);
    write(fd, ">>>>v", 5);
    close(fd);
    return 0;
}
```