History and Evolution of Programming Languages

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.

Differences in Language Paradigms

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.

Programming In C

3. Explain the purpose of each of the following GCC options:

- `-o` Place output in a specific file.
- `-c` Compile/assemble source files, but do not link them.
- `-std` Specify a specific C standard to use during compilation.
- `-Wall` Turn on most optional warnings. You might still need `-Wextra` and others.

Modular Design and Development

4. 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

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

```
1 #ifndef LINKEDLIST_H
2 #define LINKEDLIST_H
3
4 <code>
5
6 #endif
```

Variables and Basic Data Types

6. List at least 7 of the basic data types in C (not including the unsigned variants).

   char, short, int, long, long long, float, double, long double, _Bool

Initialization and Coercion

7. There are at least two issues in the following code. What are they?

```
#include <stdio.h>

int main(void) {
    int numbers[] = { 3, 6, 4, 5, 14, 9 };
    int sum, i = 0;
    double average;
    for (; i < 6; ++i) {
        sum += numbers[i];
    }
    average = sum / 6;
    printf("%f", average);
    return 0;
}
```

(a) `sum` is not initialized, so its value is unspecified at the beginning of the loop. If it doesn't happen to start at 0, the total will be incorrect at the end of the program.

(b) Because `sum` and `6` are both of type `int`, integer division is performed, and `average` does not contain the expected average, but the floor of the average. Change that line of code to one of the following:

```
average = sum / 6.0;

average = (double)sum / 6;

average = sum / (double)6;
```

Strings

8. (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'`

Arrays

9. (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.

I/O

10. There is a problem with the following code as written that will manifest itself for certain user input. Explain what this issue is and how to solve it.

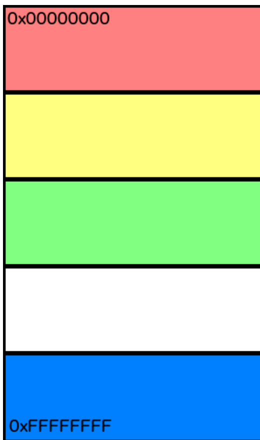
```
#include <stdio.h>

int main(void) {
        const int BUF_SIZE = 256;
        char buf[BUF_SIZE];
        while (fgets(buf, BUF_SIZE, stdin)) {
                printf(buf);
        }
}
```

If the user enters a string with a `%` character, `printf` will try to interpret this string as a format string, which will not work as intended and will cause undefined behavior, as there are no other arguments to `printf` for it to format. To solve this issue, replace `printf(buf);` with `printf("%s", buf);`.

Memory and Program Layout

11. 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

C Pre-processor

12. 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

C Pointers

13. (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;
}
```

Structs, Dynamic Storage

14. 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.