A. Questions

1) Declare the following:

   a) A prototype for a function named `func1` that accepts two pointers to `int` as input parameters and does not return anything.
   b) A prototype for a function named `func2` that accepts two pointers to `int` as input parameters and returns a pointer to an `int`.
   c) A prototype for a function named `func3` that accepts a pointer to `int` as input parameter and returns a pointer to an `int`. The function is not allowed to modify the value (pointed to) of the input parameter.
   d) A static double-precision floating point number named `sdouble`.
   e) An `int` variable named `sreg` that has register storage class.

Answers:

```
a) void func1(int *, int *);
b) int *func2(int *, int *);
c) int *func3(const int *);
d) static double sdouble;
e) register int sreg;
```

2) Consider the following C snippet:

```
for(int j=0; j<10; j++) {
   int k;
   k = j-1;
}
int i = j;
```

a) What is the storage class of `j`?
b) What is the storage class of `k`?
c) What is the initial value of `k`?
d) Is the last statement valid? If so, what is the value assigned to `i`?

Answers:
a) Auto
b) Auto
c) Garbage
d) It is invalid because `j` does not exist anymore after the for-loop.

3) Consider the following C source file:

```
#include <stdio.h>

void init_x(void)
{
   x = 1;
}

int x;

int main (void)
{
   incr_x();
   printf("%d\n", x);
   return 0;
}

void incr_x(void)
{
   x++;
}
```

a) What is the storage class of `x`?
b) What is the initial value of `x`?
c) Can the function `init_x()` access `x` as it is? If not, rewrite `init_x()` so that it can access `x`.
c) What is the output of the program?

Answers:
a) Extern
b) 0
c) No, `init_x()` cannot access `x` as it is because the scope of `x` begins after its declaration. But since it is globall, we can declare a "link" to it inside `init_x()` using the extern keyword:

```
void init_x(void)
{
   extern int x; // "Link" to global variable x
   x = 1;
}
```

d) 1

4) Consider the following C snippet:

```
char *cp;
cp = (char *)malloc(10*sizeof(char));
```

a) Assuming that the allocation is successful, what is the size (in bytes) of the memory block pointed to by `cp`?

b) Is it necessary to typecast the return value of `malloc()`to `char *`?

c) Rewrite the second line to use `calloc()`.

Answers:
a) 10 bytes
b) It is not necessary because `void *` (the return type of `malloc()`) is automatically converted to the type of the left hand side of the assignment. The typecasting is done as a matter of good programming practice
c) `cp = (char *)calloc(10, sizeof(char));`

5) Consider the following C snippet:

```
1   int *ip;
2   ip = (int *)calloc(5, sizeof(int));
3   for(int i=0; i<5; i++) {
4       *ip = i;
5       ip++;
6   }
7   free(ip);
```

Describe 3 issues with the code.

Answers:
1) After line 2, there should be check on whether the call to calloc() was successful. This can be done by checking whether ip is NULL (unsuccessful) or NOT NULL (successful).
2) In line 5, the only pointer to the allocated memory is incremented. Should use another pointer to iterate over the array.
3) Because ip is not pointing to the start of the allocated memory, passing it to free() would result in undefined behaviour.