ASSIGNMENT – C PROGRAMMING (RTOS)

1) What is the output of the following code? Explain the reason.

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

int main() {
    int *p = malloc(sizeof(int));
    *p = 10;
    free(p);
    *p = 20;
    printf("%d", *p);
    return 0;
}
```

\

→ **The code showcases undefined behavior. No specific output can be guaranteed .**

Initially through the malloc() function, a pointer p is defined that points to a block of memory (Here 4 bytes, the size of int in C).

*p = 10; This line writes the value 10 into the memory address pointed to by p.

free(p). This line now deallocates the block of memory (4 bytes) back to the heap.

A pointer that points to a block of memory that has been deallocated is called a dangling pointer.

Such pointers can cause the code to crash or corrupt memory.

After the deallocation, the memory address is not guaranteed to still hold the value 10.

*p = 20; Here, we are writing to the memory that has been deallocated, which can cause the code to crash.

But in some systems, the value 20 is written to the same memory address if the memory block has not been reused yet.

Hence, in certain compilers, we get the output as 20, but this is not always true.

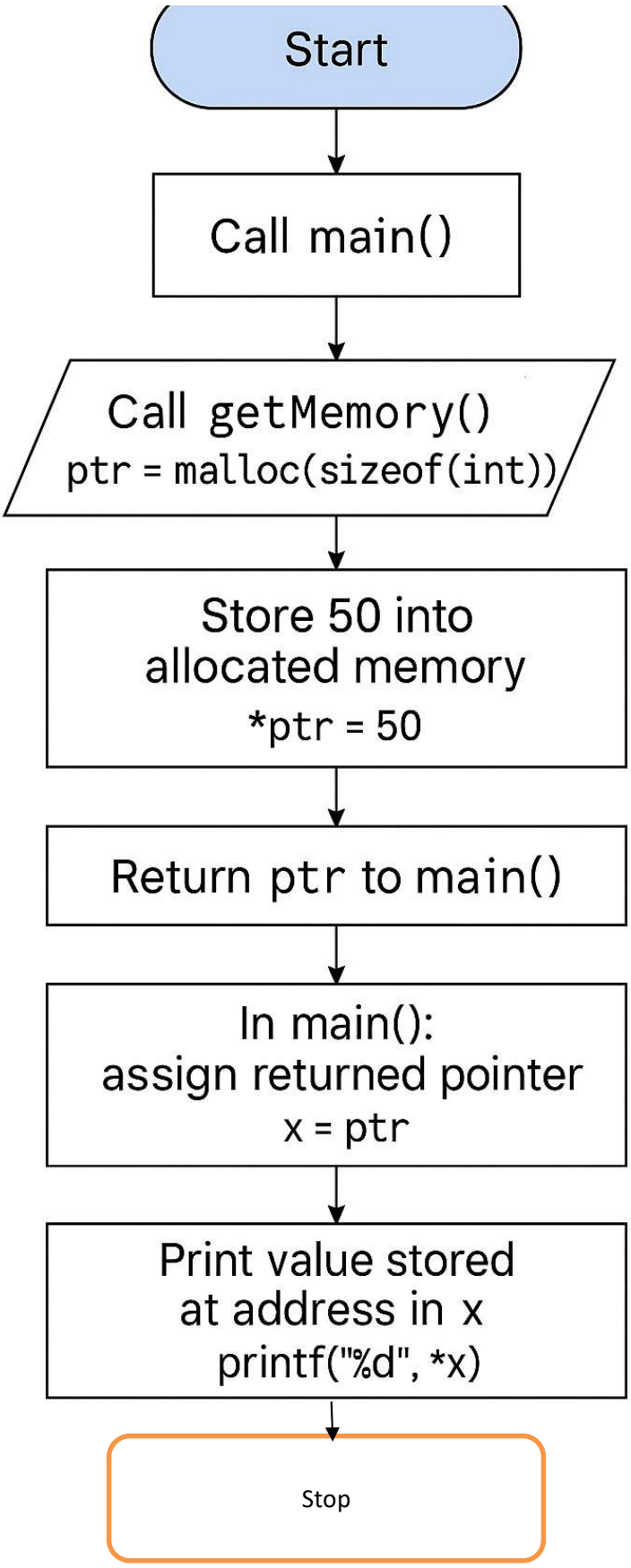
2) Determine the output of the following program and give a flow chart of the flow of operation.

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

int* getMemory() {
    int *ptr = malloc(sizeof(int));
    *ptr = 50;
    return ptr;
}

int main() {
    int *x = getMemory();
    printf("%d", *x);
    free(x);
    return 0;
}
```

➔ Output: **50**

Flow of operation:

1) Program execution starts from the main() method
2) A pointer variable x is initialized through the getMemory() function. Program control transferred to getMemory()
3) getMemory() is of pointer return type. By malloc(), a memory block of int size (4 bytes) is allocated and the memory address is stored in ptr.
4) By dereferencing ptr, the value 50 is written onto the memory address pointed to by ptr
5) This pointer variable ptr is returned back to the function call in main() and stored in x
6) x now points to the memory address that contains the value 50
7) 50 is printed as output. (data in the address pointed by x).
8) The memory block is deallocated using free()
9) The program is terminated


Start
Call main()
Call getMemory()
ptr = malloc(sizeof(int))
Store 50 into
allocated memory
*ptr = 50
Return ptr to main()
In main():
assign returned pointer
x = ptr
Print value stored
at address in x
printf("%d", *x)
Stop

3) Determine the output of this calloc() example.

```
int *arr = calloc(3, sizeof(int));
printf("%d", arr[1]);
free(arr);
```

➔ Output **: 0**

calloc() is a function that allocates memory similar to malloc() but initializes the memory block to zero.

calloc(3, sizeof(int));

This line allocates 3 blocks of 4 bytes each(integer size) to the pointer arr and all 3 memory locations are initialized with 0.

Below is the representation of the allocated memory, let's assume the address of 1st block is 'k'

0	0	0
K	k+4	k+8
arr[0]	arr[1]	arr[2]

Printing arr[1] gives output as 0, since it is the value stored in the 2nd memory block that is pointed to by arr.

4) Determine the output of the following code snippet.

```
#include <iostream>
using namespace std;
int main()
{
int a = 32, *ptr = &a;
char ch = 'A', &cho = ch;
cho += a;
*ptr += ch;
cout << a << ", " << ch << endl;
return 0;
}
```

- ➔ Initially an integer variable a is initialized with the value 32, and a pointer ptr is created which stores the address of a.
- ➔ A char variable ch is initialized with 'A', and cho is a reference to ch ; changing cho changes ch and vice versa
- ➔ cho +=a; This line performs addition on the value stored in variable ch (cho is a reference to ch) with a.
- ➔ 'A' is converted to its ASCII value 65. 65+32=97 which is the ASCII value for 'a'.
- ➔ *ptr +=ch; This line performs addition of the value stored in the pointed memory address of ptr with ch.
- ➔ 32+ASCII of 'a' which is 32+97=129.
- ➔ Therefore; a=129 ch ='a'

- ➔ OUTPUT: **129, a**

5) Why does the below code give a "compile error"? Identify the lines which cause this error and what changes have to be made for the code to run smoothly?

```
#include <iostream>
using namespace std;
int main()
{
const int i = 20;
const int* const ptr = &i;
(*ptr)++;
int j = 15;
ptr = &j;
cout << i;
return 0;
}
```

- ➔ const int i = 20;
- ➔ const int* const ptr = &i;
- ➔ These 2 lines ensure that the value stored in i should not change. The address to which ptr is pointing should not change, and the value stored in that memory address should not change .
- ➔ Errors are caused by (*ptr)++ and ptr= &j. As we are trying to change the values in const variables.

Ways to fix:

i) Remove const so that the pointer and value can be modified

```
#include <iostream>
using namespace std;

int main()
{
   int i = 20;
   int* ptr = &i;
   (*ptr)++;      // i becomes 21
   int j = 15;
```

```
    ptr = &j;      // now ptr points to j
    cout << i;     // prints 21
    return 0;
}
```

ii) Remove the lines which try to modify the const variables

```
#include <iostream>
using namespace std;

int main()
{
    const int i = 20;
    const int* const ptr = &i;
    int j = 15;
    cout << i;
    return 0;
}
```

iii) Data can be modified but pointer must be fixed

```
#include <iostream>
using namespace std;

int main()
{
    int i = 20;          // removed const on i
    int* const ptr = &i;  // const pointer to non-const int
    (*ptr)++;            // i becomes 21
    int j = 15;
    cout << i;           // prints 21
    return 0;
}
```

iv) Data should be fixed, but the pointer can be movable

```
#include <iostream>
using namespace std;
int main()
{
   const int i = 20;
   const int* ptr = &i;   // pointer (non-const) to const int
   const int j = 15;      // j must also be const if ptr is const int*
   ptr = &j;              //  pointer can move

   cout << i;             // still 20
   return 0;
}
```

6) Determine the output of the following code and explain how it works.

```
#include <stdio.h>
char *fun()
{
   static char arr[1024];
   return arr;
}

int main()
{
   char *str = "rtos";
   strcpy(fun(), str);
   str = fun();
   strcpy(str, "cprog");
   printf("%s", fun());
   return 0;
}
```

- ➔ arr is a static array created of size 1024. Since it is static, every time fun() is called, the same array is returned.
- ➔ char *str = “rtos”; Here str is a pointer to the string literal “rtos”.
- ➔ strcpy(fun(), str); Here fun() returns the static array arr.
- ➔ Therefore, it becomes strcpy(arr, "rtos");
- ➔ “rtos” is now stored in the array arr
- ➔ str = fun(); Now str is assigned the same pointer returned by fun(). That is, str now points to arr
- ➔ strcpy(str, "cprog"); Now cprog is copied into arr (because str points to arr)
- ➔ Now, arr[] = “cprog”
- ➔ The printf statement prints out the array arr

Therefore : OUTPUT = **cprog**