1. C program to illustrate effect of setjmp and longjmp functions on register, volatile and automatic variables.

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

static void f1(int, int, int, int);
static void f2(void);

static jmp_buf jmpbuffer;
static int globval;

int main(void)
{
        int autoval;
        register int regival;
        volatile int volaval;
        static int statval;

        globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;
        if (setjmp(jmpbuffer) != 0)
        {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d, volaval = %d, statval = %d\n", globval,
autoval, regival, volaval, statval);
        exit(0);
        }
/*
* Change variables after setjmp, but before longjmp.
*/
        globval = 95; autoval = 96; regival = 97; volaval = 98;
        statval = 99;
        f1(autoval, regival, volaval, statval); /* never returns */
                exit(0);
}

static void f1(int i, int j, int k, int l)
{
        printf("in f1():\n");
        printf("globval = %d, autoval = %d, regival = %d, volaval = %d, statval = %d\n", globval, i, j,
k, l);
        globval=10000;
        j=10000;
        f2();
}

static void f2(void)
{
        longjmp(jmpbuffer, 1);
}
```

Explanation

This program demonstrates the behavior of setjmp and longjmp on different variable storage classes:

1. Initialization:

- Global variables:
  - globval is initialized to 1.
- Automatic variable:
  - autoval is initialized to 2.
- Register variable:
  - regival is initialized to 3 (compiler might store it in a register).
- Volatile variable:
  - volaval is initialized to 4.
- Static variable:
  - statval is initialized to 5.

2. setjmp(jmpbuffer)**:**

- The call to setjmp(jmpbuffer) saves the current execution state (including register values) in the jmpbuffer buffer.
- It returns 0 on the first call, indicating the initial jump context.

3. Variable Modification:

- **After** setjmp **but before** longjmp**:**
  - The program modifies the values of all variables:
    - globval is changed to 95.
    - autoval is changed to 96.
    - regival is changed to 97.
    - volaval is changed to 98.
    - statval is changed to 99.

4. f1(autoval, regival, volaval, statval)**:**

- The program calls f1 with the current values of the arguments (96, 97, 98, 99). However, due to longjmp inside f1, this function never returns control to main.

5. Inside f1**:**

- f1 prints the values of the arguments (which are the initial values from main after setjmp).
- f1 modifies globval to 10000.
- f1 modifies the second argument (j, which is a copy of regival) to 10000 (this change won't affect the actual regival in main).

6. f2()**:**

- f1 calls f2().
- f2 calls longjmp(jmpbuffer, 1). This does the following:

- Restores the execution state saved in jmpbuffer during the first setjmp call.
  - Jumps back to main immediately after the setjmp call, as if f1 and f2 never happened.
  - The value of the argument (1) passed to longjmp is ignored in this program.

7. Back in main **(after** longjmp**):**

- The program execution resumes in main as if the function calls to f1 and f2 never occurred.
- The modified values of the variables before longjmp are no longer relevant.
- main prints the values of the variables after setjmp:
  - globval will be 95 (modified before longjmp).
  - autoval will be undefined (stack memory overwritten during function call/return).
  - regival will have an unpredictable value (compiler might have used a register).
  - volaval might be 98 (compiler's optimization behavior), but could also be the original value (4) depending on the compiler.
  - statval will be 99 (static variable retains its value).

8. Program Termination:

- main exits with exit(0).

2. Write a C program To create a child process and child should perform addition of two numbers and parent process to perform product of two numbers.

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>


int main() {

  int num1 = 5, num2 = 3;


  int pid = fork();


  if (pid < 0) {

    perror("fork");

    exit(1);

  }

else if (pid == 0) { // Child process

    int sum = num1 + num2;

    printf("Child process: Addition of %d and %d is %d\n", num1, num2, sum);
```

```c
        exit(0);
    }
else { // Parent process
        int product = num1 * num2;
        printf("Parent process: Multiplication of %d and %d is %d\n", num1, num2, product);
        wait(NULL); // Wait for child process to finish
    }

    return 0;
}
```

Explanation:

Initial State:

- Parent process starts execution.
- Variables:
    - num1 = 5
    - num2 = 3

fork() **System Call:**

1. The parent process calls fork().
2. The kernel creates a child process that is a nearly identical copy of the parent process.
3. Both processes (parent and child) now exist and continue execution from the same point after fork().

Variable Copies:

- Each process (parent and child) gets its own copy of the variables.
    - num1 (value 5) and num2 (value 3) are copied into both processes' memory spaces.

Process Divergence:

1. The parent process checks the return value of fork():
    - If the return value is less than 0 (pid < 0), an error occurred during fork(). The program exits with an error message.
    - If the return value is 0 (pid == 0), it's the child process.
    - If the return value is greater than 0 (pid > 0), it's the parent process with the child's process ID (pid) stored in the variable pid.
2. **Child Process (pid == 0):**
    - The child process continues execution.
    - It calculates the sum of num1 and num2: sum = num1 + num2 = 5 + 3 = 8.
    - The child process prints the calculated sum:
    - Child process: Addition of 5 and 3 is 8

  - The child process exits successfully (exit(0))
3. **Parent Process (pid > 0):**
  - The parent process continues execution after the child process exits.
  - It calculates the product of num1 and num2: product = num1 * num2 = 5 * 3 = 15.
  - The parent process prints the calculated product:
  - Parent process: Multiplication of 5 and 3 is 15

  - The parent process waits for the child process to finish using wait(NULL).
  - The parent process exits successfully (return 0).

Program Termination:

- Both child and parent processes terminate, and the program execution ends.

Output:

Child process: Addition of 5 and 3 is 8
Parent process: Multiplication of 5 and 3 is 15

3. Write a C program To create a child process and both parent and child should share a text file to perform read and write operations on a file. Show how the offset value is shared between parent and the child processes.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>

#define FILE_NAME "shared_file.txt"
#define SHARED_MEM_SIZE sizeof(int)

int main() {
    int fd;
    int *offset_ptr;
    char *write_buffer = "This is written from the parent process.\n";
    char read_buffer[100];

    // Create/open the file with read/write permissions
    fd = open(FILE_NAME, O_RDWR | O_CREAT, 0666);
```

```
    if (fd == -1) {
        perror("open");
        exit(1);
    }

    // Create a shared memory segment
    int shm_fd = shm_open("offset_shm", O_RDWR | O_CREAT, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(1);
    }

    // Set the size of the shared memory segment
    ftruncate(shm_fd, SHARED_MEM_SIZE);

    // Map the shared memory segment to the address space of this process
    offset_ptr = mmap(NULL, SHARED_MEM_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, shm_fd, 0);
    if (offset_ptr == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    pid_t pid = fork();

    if (pid < 0) {
        perror("fork");
        exit(1);
    } else if (pid == 0) { // Child process
        printf("Child process started.\n");

        // Seek to the current offset in the file
```

```
    lseek(fd, *offset_ptr, SEEK_SET);

    // Read from the file
    int bytes_read = read(fd, read_buffer, sizeof(read_buffer));
    if (bytes_read > 0) {
        read_buffer[bytes_read] = '\0'; // Null-terminate the string
        printf("Child process: Read %d bytes: %s\n", bytes_read, read_buffer);
    } else {
        perror("read");
    }

    // Update the offset with the current position in the file
    *offset_ptr = lseek(fd, 0, SEEK_CUR);

    printf("Child process finished.\n");
} else { // Parent process
    printf("Parent process started.\n");

    // Write to the file
    write(fd, write_buffer, strlen(write_buffer));

    // Update the offset with the current position in the file
    *offset_ptr = lseek(fd, 0, SEEK_CUR);

    printf("Parent process finished.\n");
}

// Unmap the shared memory segment
munmap(offset_ptr, SHARED_MEM_SIZE);

// Close the shared memory segment descriptor
```

```
    close(shm_fd);


    // Close the file descriptor
    close(fd);


    return 0;
}
```

Explanation

Initial State:

- Parent process starts execution.
- Shared memory segment "offset_shm" is created with size sizeof(int).
- File shared_file.txt is opened/created with read/write permissions.
- offset_ptr points to the mapped shared memory segment containing the offset value (initially 0).

fork() **System Call:**

1. The parent process calls fork().
2. The kernel creates a child process, which is a nearly identical copy of the parent.
   - Both processes (parent and child) now have their own copy of variables but share the same file descriptor (fd) and mapped shared memory segment (offset_ptr).

Process Divergence:

1. **Parent Process (pid > 0):**
   - Prints "Parent process started."
   - **Write Operation:**
     - Writes the content "This is written from the parent process.\n" to the file using write.
     - Updates the offset value in the shared memory segment (*offset_ptr) with the current position in the file using lseek(fd, 0, SEEK_CUR). This points to the end of the written data.
   - Prints "Parent process finished."
2. **Child Process (pid == 0):**
   - Prints "Child process started."
   - **Read Operation:**
     - Seeks to the current offset value in the file using lseek(fd, *offset_ptr, SEEK_SET).
     - **Scenario 1 (Child Reads After Parent Writes):**
       - If the parent has already written, the child will read the written content (This is written from the parent process.\n).
     - **Scenario 2 (Child Reads Before Parent Writes):**
       - If the child reads before the parent writes, the read operation will return 0 bytes (read_buffer will be empty).
   - Prints the read data (if any) along with the number of bytes read.

- Updates the offset value in the shared memory segment (*offset_ptr) with the current position in the file (which might be the same as before or at the end of the read data).
- Prints "Child process finished."

Synchronization with Shared Memory:

- The offset value in the shared memory segment acts as a coordination point between the parent and child processes.
- The parent writes, updates the offset, and then the child reads from that updated position.
- This ensures that the child doesn't overwrite the parent's written data and can potentially read the written content (depending on the timing).

Cleanup:

- Both processes unmap the shared memory segment, close the shared memory segment descriptor, and close the file descriptor before exiting.

Output:

The output will vary depending on the timing of the read operation in the child process, as explained in the previous explanation