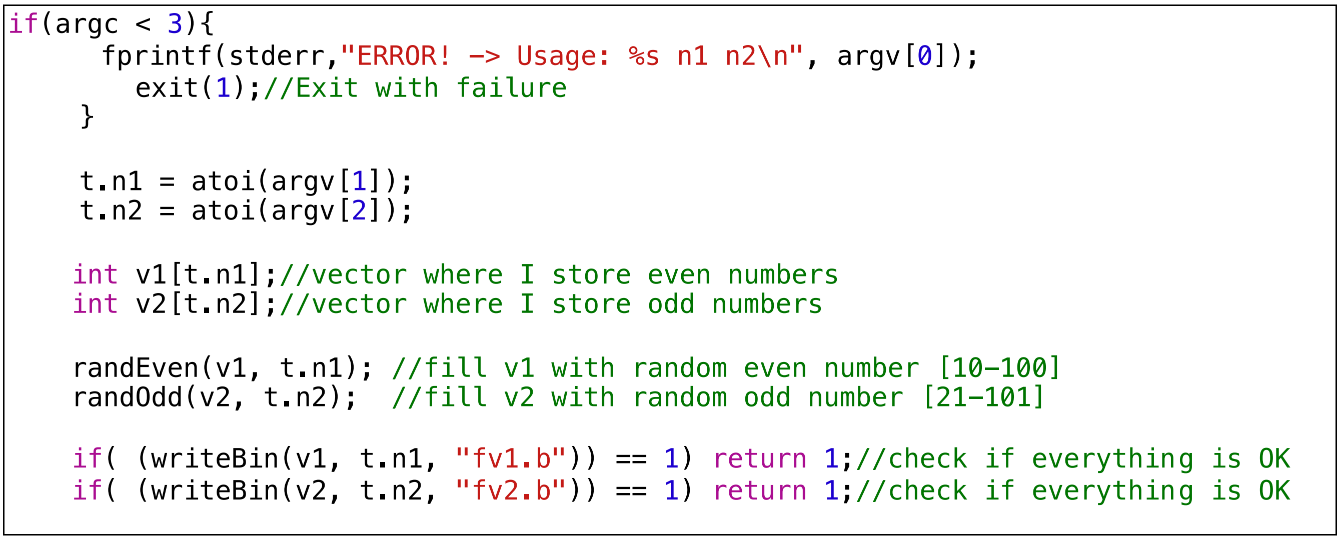
REPORT

- ## Exercise3

  In the first part, the program takes from the command line two variables **n1** and **n2**(dimension of the vectors), allocates two vectors and fills them one with random even number **[10-100]** and the other with random odd numbers **[21-101]**. Finally, it saves the two vectors into two binary files, respectively.



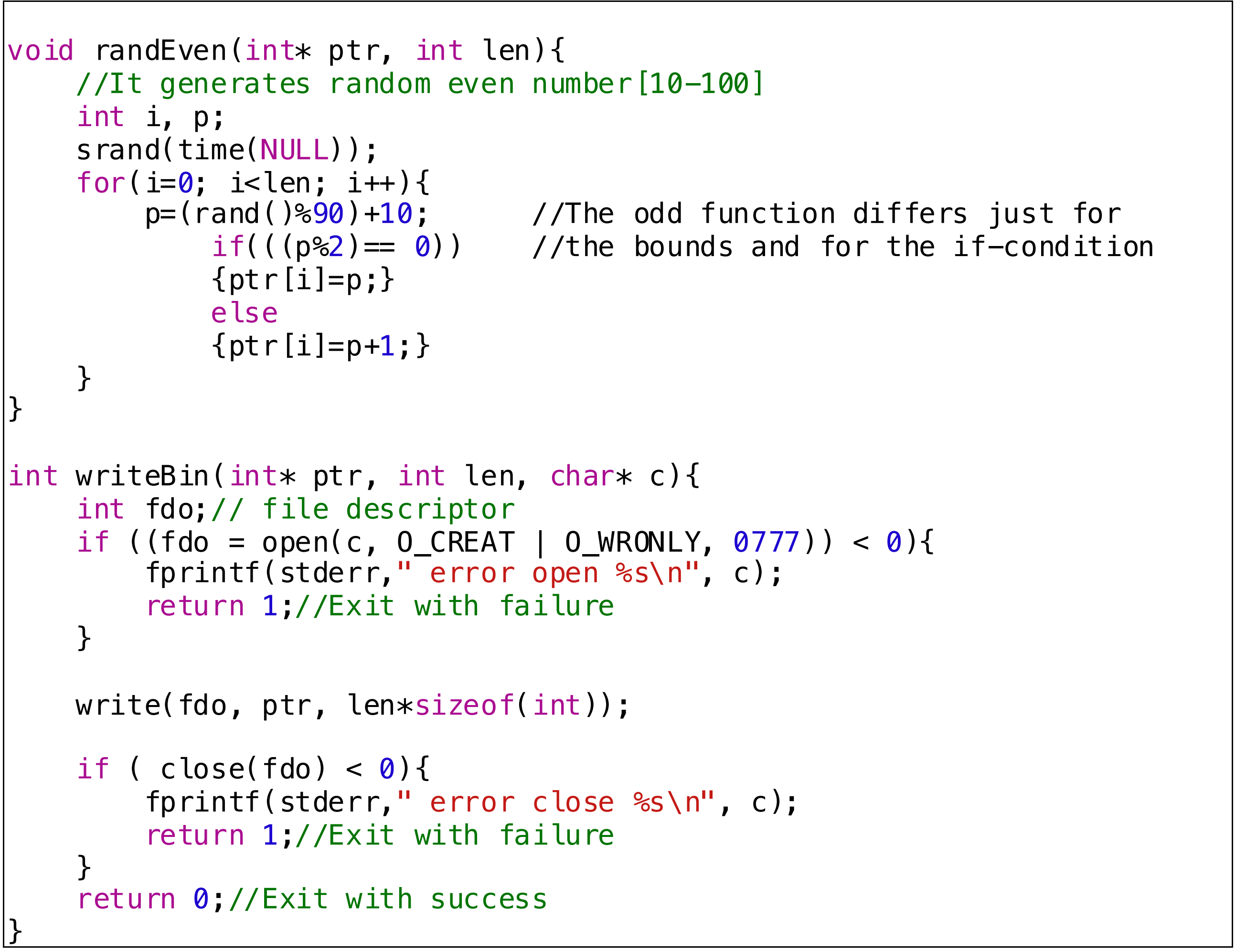
```
if(argc < 3){
      fprintf(stderr,"ERROR! -> Usage: %s n1 n2\n", argv[0]);
        exit(1);//Exit with failure
    }

    t.n1 = atoi(argv[1]);
    t.n2 = atoi(argv[2]);

    int v1[t.n1];//vector where I store even numbers
    int v2[t.n2];//vector where I store odd numbers

    randEven(v1, t.n1); //fill v1 with random even number [10-100]
    randOdd(v2, t.n2);  //fill v2 with random odd number [21-101]

    if( (writeBin(v1, t.n1, "fv1.b")) == 1) return 1;//check if everything is OK
    if( (writeBin(v2, t.n2, "fv2.b")) == 1) return 1;//check if everything is OK
```



```
void randEven(int* ptr, int len){
    //It generates random even number[10-100]
    int i, p;
    srand(time(NULL));
    for(i=0; i<len; i++){
        p=(rand()%90)+10;      //The odd function differs just for
            if(((p%2)== 0))    //the bounds and for the if-condition
            {ptr[i]=p;}
            else
            {ptr[i]=p+1;}
    }
}

int writeBin(int* ptr, int len, char* c){
    int fdo;// file descriptor
    if ((fdo = open(c, O_CREAT | O_WRONLY, 0777)) < 0){
        fprintf(stderr," error open %s\n", c);
        return 1;//Exit with failure
    }

    write(fdo, ptr, len*sizeof(int));

    if ( close(fdo) < 0){
        fprintf(stderr," error close %s\n", c);
        return 1;//Exit with failure
    }
    return 0;//Exit with success
}
```

In the second part, the program creates two client threads and then it acts like a server. The client thread loops reading the numbers from the binary file written before, at each iteration stores the number into a global variable **g**, then it signals on a semaphore to indicate to the server that the variable is ready to be processed (simply multiplied by 3) and waits on a semaphore that the server has done its task. Finally, the client prints the result and its identifier.

```
...
if ((fdo = open("fv1.b", O_RDONLY, 0777)) < 0){
        fprintf(stderr," error open %s\n", "fv1.b");
        exit(1);//Exit with failure
    }
    //read the sequence of bits and store them into an array
    v=(int*)malloc((th->n1)*sizeof(int));
    read(fdo, v, (th->n1)*sizeof(int));
    //colse the file
    if ( close(fdo) < 0){
        fprintf(stderr," error close %s\n", "fv1.b");
        exit(1);//Exit with failure
    }
    //loop the call to the server
    for(i=0; i<(th->n1); i++){
     sem_wait(mutex);
    //store into g the read value
        (th->g) = v[i];
    //call the server and wait before to print
     sem_post(sem0);
     sem_wait(ready);
     printf("[C1]: tID = %ld --> g = %d\n", pthread_self(), th->g);
     sem_post(mutex);
    }
```

The server loops waiting the signal of the clients, then does the multiplication and stores the result into the same global variable **g**, finally signals back to the clients that the result is ready. At the end, after it waits the end of both the clients, prints how many requests has served.

```
...
sem_init(mutex, 0, 1);//initialize the semaphores
sem_init(sem0, 0, 0);
sem_init(ready, 0, 0);
t.g = 0;
counter = 0;
pthread_create(&th_b, NULL, C1, (void*) &t);//create threads
sleep(1);
pthread_create(&th_c, NULL, C2, (void*) &t);
while(counter<(t.n1+t.n2)){
  sem_wait(sem0);
  counter++;
  t.g = 3 * t.g;
  sem_post(ready);
}
pthread_join(th_b, NULL);
pthread_join(th_c, NULL);
printf("The server \"main\" has served %d request from the clients;\n",
counter);
```