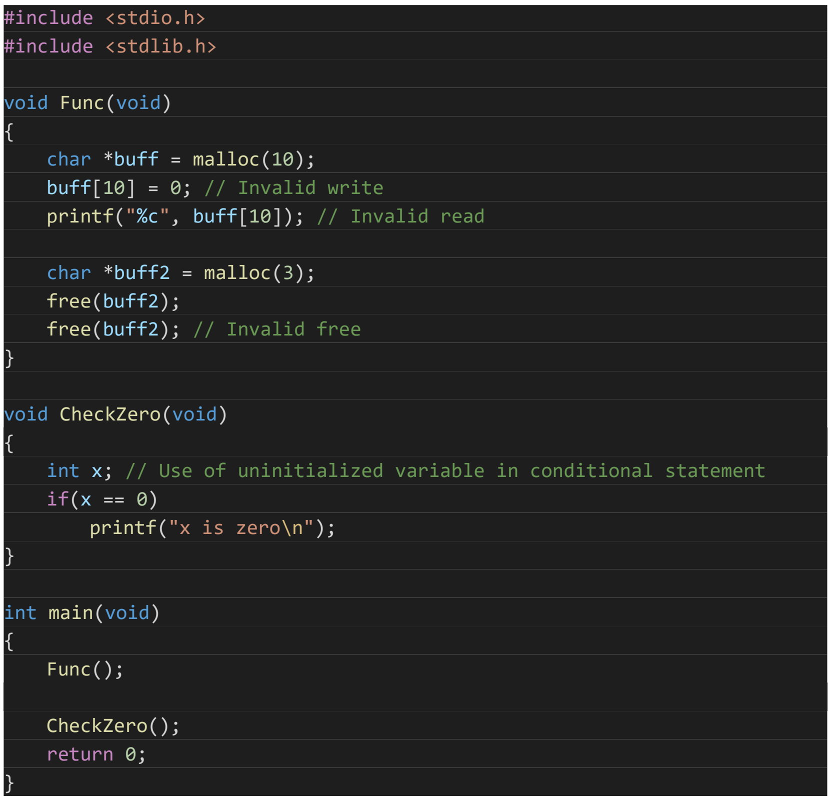
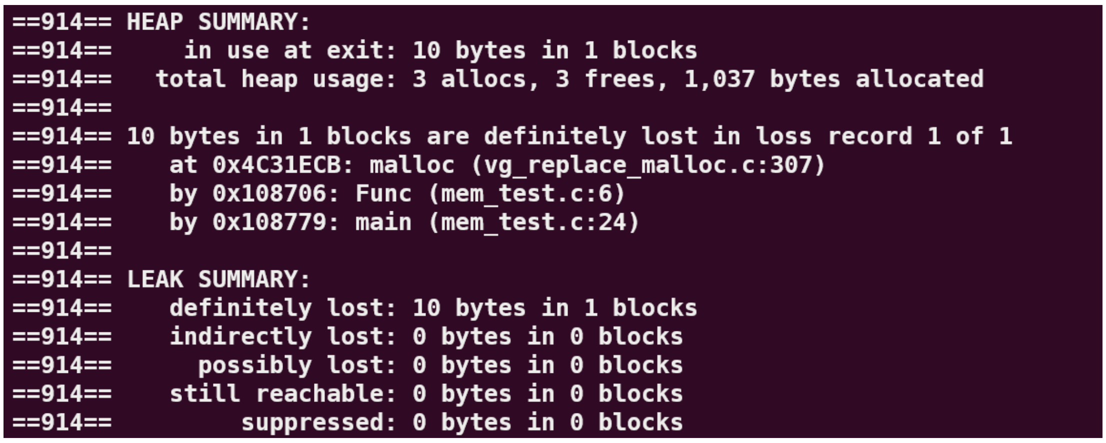
Using Valgrind

Sample Code:



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

void Func(void)
{
    char *buff = malloc(10);
    buff[10] = 0; // Invalid write
    printf("%c", buff[10]); // Invalid read

    char *buff2 = malloc(3);
    free(buff2);
    free(buff2); // Invalid free
}

void CheckZero(void)
{
    int x; // Use of uninitialized variable in conditional statement
    if(x == 0)
        printf("x is zero\n");
}

int main(void)
{
    Func();

    CheckZero();
    return 0;
}
```

Finding Memory Leaks



```
==914== HEAP SUMMARY:
==914==     in use at exit: 10 bytes in 1 blocks
==914==   total heap usage: 3 allocs, 3 frees, 1,037 bytes allocated
==914==
==914== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==914==    at 0x4C31ECB: malloc (vg_replace_malloc.c:307)
==914==    by 0x108706: Func (mem_test.c:6)
==914==    by 0x108779: main (mem_test.c:24)
==914==
==914== LEAK SUMMARY:
==914==    definitely lost: 10 bytes in 1 blocks
==914==    indirectly lost: 0 bytes in 0 blocks
==914==      possibly lost: 0 bytes in 0 blocks
==914==    still reachable: 0 bytes in 0 blocks
==914==         suppressed: 0 bytes in 0 blocks
```

The allocated memory buff is definitely lost with 10 bytes size.

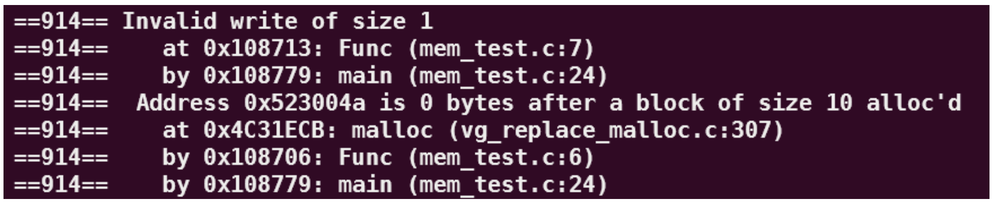
On the source code line 6 inside the Func function and called by the main function on line 24.

- **Definitely lost**: All the pointer to an allocated memory is lost and the memory is not freed.
- **Indirectly lost**: The pointer is in a class or a struct and the pointer or class itself is lost. Usually comes with Definitely lost.
- **Possibly lost**: There is a pointer point to the allocated memory but not point to the start of it. Therefore, calculation is needed to find the start of the memory and free it.

- **Still reachable**: There is live pointer point to the allocated memory when the program exit.

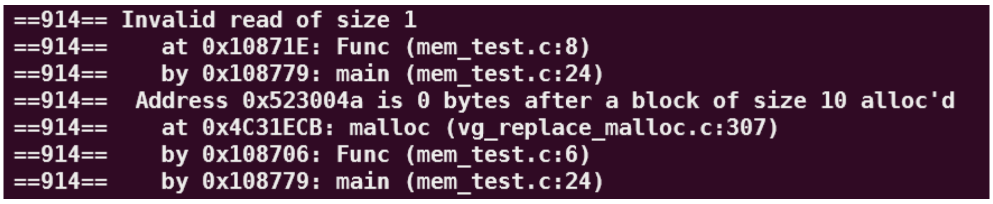
Invalid Memory Access

- Invalid Write:



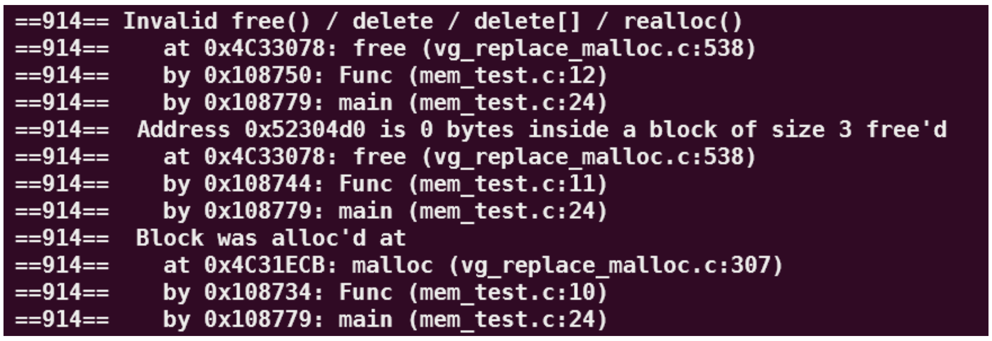
```
==914== Invalid write of size 1
==914==    at 0x108713: Func (mem_test.c:7)
==914==    by 0x108779: main (mem_test.c:24)
==914==  Address 0x523004a is 0 bytes after a block of size 10 alloc'd
==914==    at 0x4C31ECB: malloc (vg_replace_malloc.c:307)
==914==    by 0x108706: Func (mem_test.c:6)
==914==    by 0x108779: main (mem_test.c:24)
```

- Invalid Read:



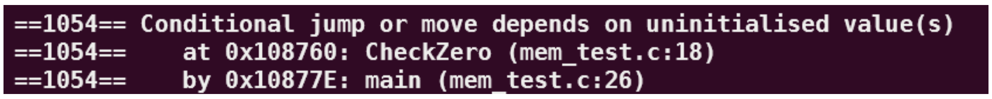
```
==914== Invalid read of size 1
==914==    at 0x10871E: Func (mem_test.c:8)
==914==    by 0x108779: main (mem_test.c:24)
==914==  Address 0x523004a is 0 bytes after a block of size 10 alloc'd
==914==    at 0x4C31ECB: malloc (vg_replace_malloc.c:307)
==914==    by 0x108706: Func (mem_test.c:6)
==914==    by 0x108779: main (mem_test.c:24)
```

- Invalid Free:



```
==914== Invalid free() / delete / delete[] / realloc()
==914==    at 0x4C33078: free (vg_replace_malloc.c:538)
==914==    by 0x108750: Func (mem_test.c:12)
==914==    by 0x108779: main (mem_test.c:24)
==914==  Address 0x52304d0 is 0 bytes inside a block of size 3 free'd
==914==    at 0x4C33078: free (vg_replace_malloc.c:538)
==914==    by 0x108744: Func (mem_test.c:11)
==914==    by 0x108779: main (mem_test.c:24)
==914==  Block was alloc'd at
==914==    at 0x4C31ECB: malloc (vg_replace_malloc.c:307)
==914==    by 0x108734: Func (mem_test.c:10)
==914==    by 0x108779: main (mem_test.c:24)
```

Finding Conditional Statement with Uninitialized Variables

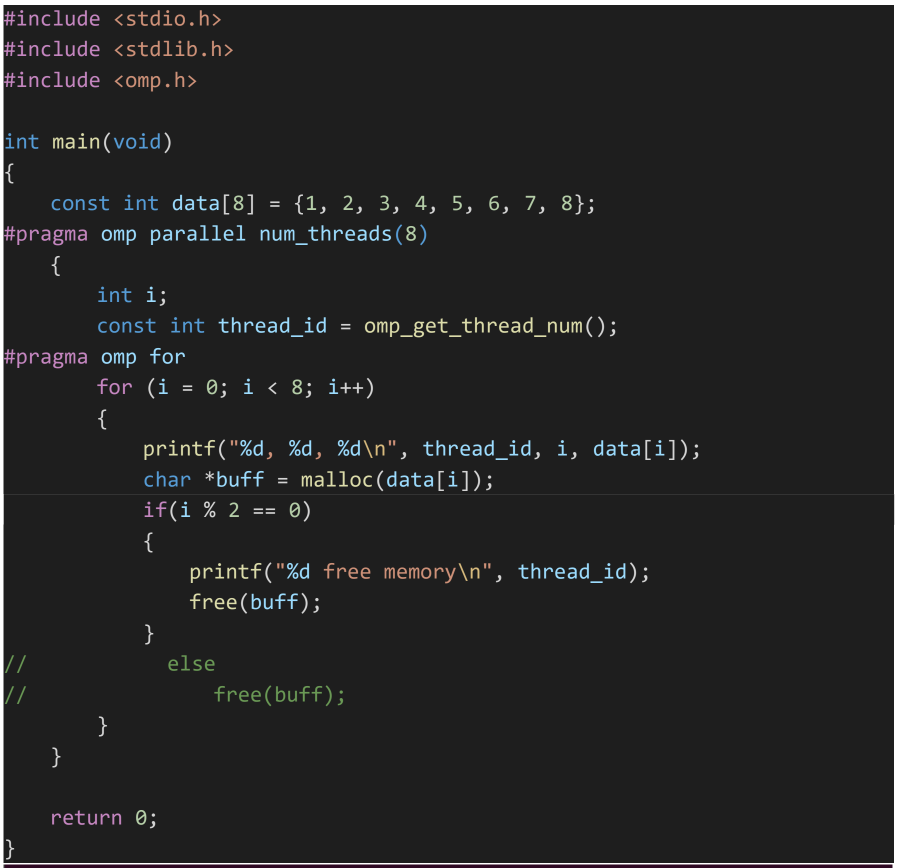


```
==1054== Conditional jump or move depends on uninitialised value(s)
==1054==    at 0x108760: CheckZero (mem_test.c:18)
==1054==    by 0x10877E: main (mem_test.c:26)
```

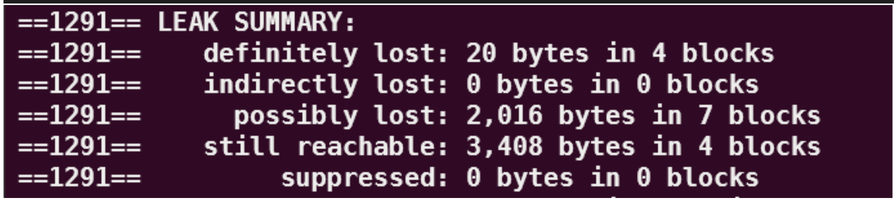
The x isn't initialized when used in the conditional statement.

On Multithread Programs

Use OpenMP for multithread programming.



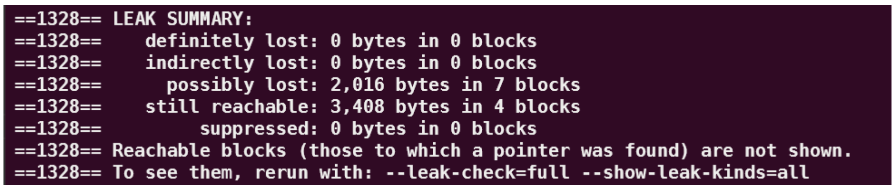
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(void)
{
    const int data[8] = {1, 2, 3, 4, 5, 6, 7, 8};
#pragma omp parallel num_threads(8)
    {
        int i;
        const int thread_id = omp_get_thread_num();
#pragma omp for
        for (i = 0; i < 8; i++)
        {
            printf("%d, %d, %d\n", thread_id, i, data[i]);
            char *buff = malloc(data[i]);
            if(i % 2 == 0)
            {
                printf("%d free memory\n", thread_id);
                free(buff);
            }
//            else
//                free(buff);
        }
    }

    return 0;
}
```



```
==1291== LEAK SUMMARY:
==1291==    definitely lost: 20 bytes in 4 blocks
==1291==    indirectly lost: 0 bytes in 0 blocks
==1291==      possibly lost: 2,016 bytes in 7 blocks
==1291==    still reachable: 3,408 bytes in 4 blocks
==1291==         suppressed: 0 bytes in 0 blocks
```

The definitely lost section shows the correct amount of lost memory.

As for possibly lost and still reachable memory blocks, it seems to be the multithread programming. The following is the result after fixing the memory leak (commented lines) and the possibly lost and still reachable section are the same.



```
==1328== LEAK SUMMARY:
==1328==    definitely lost: 0 bytes in 0 blocks
==1328==    indirectly lost: 0 bytes in 0 blocks
==1328==      possibly lost: 2,016 bytes in 7 blocks
==1328==    still reachable: 3,408 bytes in 4 blocks
==1328==         suppressed: 0 bytes in 0 blocks
==1328== Reachable blocks (those to which a pointer was found) are not shown.
==1328== To see them, rerun with: --leak-check=full --show-leak-kinds=all
```