MEMORY ERRORS

Memory errors can be broadly classified into **Heap Memory Errors** and **Stack Memory Errors**. Some of the challenging memory errors are:

- **Invalid Memory Access in heap and stack**
- **Memory leak**
- **Mismatched Allocation/Deallocation**
- **Missing Allocation**
- **Uninitialized Memory Access in heap and stack**
- **Cross Stack Access**

Invalid Memory Access

This error occurs when a read or write instruction references unallocated or deallocated memory.

```
char *pStr = (char*) malloc(25);
free(pStr);
strcpy(pStr, .parallel programming.); // Invalid write to deallocated memory in heap
```

Memory leaks

Memory leaks occur when memory is allocated but not released. If such leaks happen often enough and frequently enough, the leaks will eventually cause the application to run out of memory resulting in a premature termination (gracefully or as a crash).

```
char *pStr = (char*) malloc(512);
return;
```

Mismatched Allocation/Deallocation

This error occurs when a deallocation is attempted with a function that is not the logical counterpart of the allocation function used.

```
char *s = (char*) malloc(5);
delete s;
```

To avoid mismatched allocation/deallocation, ensure that the right deallocator is called. In C++, new[] is used for memory allocation and delete[] for freeing up. In C, malloc(), calloc() and realloc() functions are used for allocating memory while the free() function is used for freeing up allocated memory. Similarly, there are APIs in Windows programming to allocate and free memory.

Missing allocation

This error occurs when freeing memory which has already been freed. This is also called "repeated free" or "double free". Example:

```
char* pStr = (char*) malloc(20);
free(pStr);
free(pStr); // results in an invalid deallocation
```

Uninitialized Memory Access

This type of memory error will occur when an uninitialized variable is read in your application.

```
char *pStr = (char*) malloc(512);
char c = pStr[0]; // the contents of pStr were not initialized
void func()
{
    int a;
    int b = a * 4; // uninitialized read of variable a
}
```

To avoid this type of memory error, always initialize variables before using them.

Cross Stack Access

This occurs when a thread accesses stack memory of a different thread.

```
main()
{
    int *p;
    -------
    CreateThread(., thread #1, .); // Stack Owned
    CreateThread(., thread #2, .);
    -------
}
Thread #1
{
    int q[1024];
    p = q;
    q[0] = 1;
}
Thread #2
{
    *p = 2; // Stack Cross Accessed
}
```

One of the easiest ways to avoid this error is to avoid saving stack addresses to global variables.