Department of Computer Science and Engineering,
PES University, Bangalore, India

Lecture Notes
Problem Solving With C
UE24CS151B

Lecture #9
Errors and Best practices associated with Dynamic Memory Management Functions

By,
Prof. Sindhu R Pai,
Theory Anchor, Feb-May, 2025
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Center, PES University)
Prof. Nitin V Poojari (Dean, Internal Quality Assurance Cell, PES University)

Unit #: 3

Unit Name: Text Processing and User-Defined Types

Topic: Errors and Best practices associated with Dynamic Memory Management Functions

Course objectives: The objective(s) of this course is to make students

- Acquire knowledge on how to solve relevant and logical problems using computing Machine.
- Map algorithmic solutions to relevant features of C programming language constructs.
- Gain knowledge about C constructs and its associated ecosystem.
- Appreciate and gain knowledge about the issues with C Standards and it's respective behaviours.

Course outcomes: At the end of the course, the student will be able to:

- Understand and Apply algorithmic solutions to counting problems using appropriate C Constructs.
- Understand, Analyze and Apply sorting and Searching techniques.
- Understand, Analyze and Apply text processing and string manipulation methods using Arrays, Pointers and functions.
- Understand user defined type creation and implement the same using C structures, unions and other ways by reading and storing the data in secondary systems which are portable.

Sindhu R Pai

Theory Anchor, Feb - May, 2025

Dept. of CSE,

PES University

Introduction

Dynamic Memory Allocation in C provides flexibility and efficient memory use, but also introduces potential pitfalls that can lead to serious runtime issues if not handled carefully. From **memory leaks to undefined behavior**, these errors are often subtle but critical. **Understanding these common mistakes/errors is essential for writing safe and reliable C programs.**

Improper handling of functions like malloc(), calloc(), realloc(), and free() can lead to serious bugs. Below are some typical errors described with code snippets.

1. **Dangling Pointer**
2. **NULL Pointer**
3. **Memory Leak**
4. **Double free error**
5. Uninitialized Pointer Use
6. Memory Overrun
7. Invalid free
8. Not checking return values of functions

Dangling Pointer

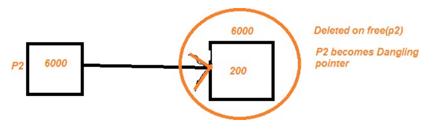
A pointer which points to a location that doesn't exist is known as dangling pointer. It can happen anywhere in the memory segment. Solution is to assign the pointer to NULL. Situations that results in dangling pointer are as below.

- Freeing the memory results in dangling pointer
- Using new pointer variable to store return address in realloc results in dangling pointer

Coding Example_1: Freeing the memory results in dangling pointer

```
#include<stdio.h>
#include<stdlib.h>
int main(){
    int *p2 = (int*)malloc(sizeof(int));        // conversion from void* to int*
    printf("%p\n",p2);
   *p2 = 200;
    printf("%p\n", p2);
```

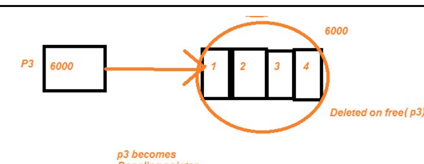
```
    printf("Before freeing %d\n", *p2);        // 200
    free(p2); // p2 becomes dangling after this
    printf("After freeing %d\n", *p2);         // undefined behavior
    return 0;
}
```




```
C:\Users\Dell>a
006C13B8
006C13B8
Before freeing 200
After freeing 7082984
```

Coding Example_2: Freeing the memory results in dangling pointer

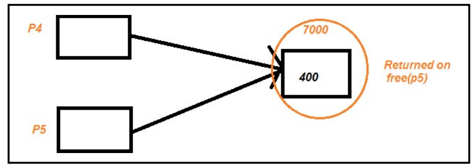
```
int main()
{    printf("Enter the number of integers you want to store\n");
     int n; int i;
     scanf("%d", &n);  // user entered 4
     int *p3 = (int*)malloc(n*sizeof(int));// initially all values undefined values
     printf("Enter %d elements\n", n);
     for(i = 0;i<n; i++)
           scanf("%d",&p3[i]);   // (p3+i)// user entered 1 2 3  4
     printf("Entered elements are\n");
     for(i = 0;i<n; i++)
           printf("%d\n",p3[i]); // *(p3+i)
     free(p3);
     printf("After freeing Entered elements are\n");
     for(i = 0;i<n; i++)
           printf("%d\n",p3[i]); // *(p3+i)
     return 0;
}
```



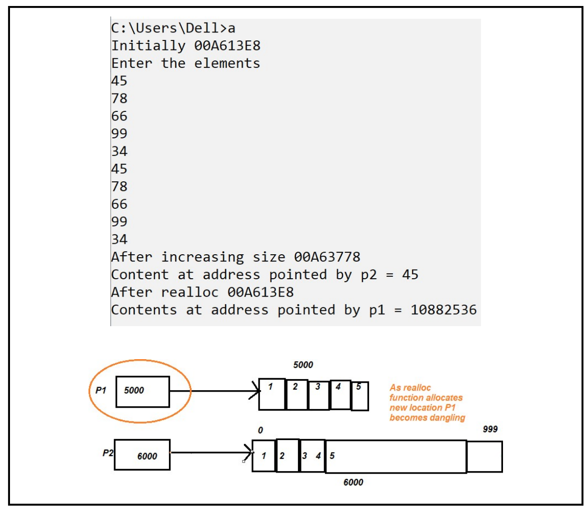

```
C:\Users\Dell>a
Enter the number of integers you want to store
4
12
676
33
22
Entered elements are
12
676
33
22
after freeing Entered elements are
10554856
10556344
33
22
```

Coding Example_3:

```
int *p4 = (int*)malloc(sizeof(int));
*p4 = 400;
printf(" %d\n", *p4);
int *p5 = p4;
free(p5);
// value of the pointer has the book keeping info available
//Both p4 and p5 becomes dangling pointer
```



Coding Example_4: Using new pointer variable to store return address in realloc results in dangling pointer.

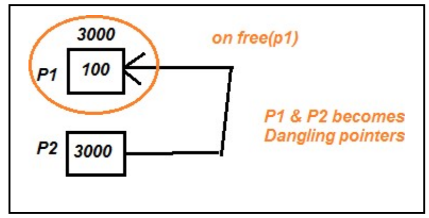



```
int main()
{
        int *p1 = (int *) calloc(5, sizeof(int));
        printf("Initially %p\n", p1);
        printf("Enter the elements\n");
        for(int i = 0; i < 5; i++)
                scanf("%d", &p1[i]);
        for(int i = 0; i < 5; i++)
                printf("%d\n", p1[i]);
        int *p2 = (int*) realloc(p1, 1000*sizeof(int));  // p1 becomes dangling if new locations allotted
        printf("After increasing size %p\n", p2); // same address as p1 if same location can be extended
        //else Different address
        printf("Content at address pointed by p2 = %d\n", *p2);
        printf("After realloc %p\n", p1);
        printf("Contents at address pointed by p1 = %d\n", *p1); return 0;
}
```

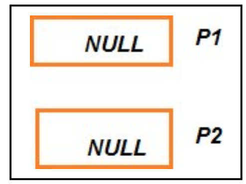
NULL Pointer

It is always a good practice to assign the pointer to NULL once after the usage of free on the pointer to avoid dangling pointers. This results in NULL Pointer. **Dereferencing the NULL pointer results in guaranteed crash.**

Coding Example_5:




```
int *p1 = (int*)malloc(sizeof(int));
*p1 = 100;      printf(" %d\n", *p1); // 100
int *p2 = p1;
printf(" %d\n", *p2); // 100
free(p1); // p1 and p2 both becomes dangling pointer.
p1 = NULL; p2 = NULL; // safe programming
printf(" %d\n", *p1);// Guaranteed crash
printf(" %d\n", *p2);
```

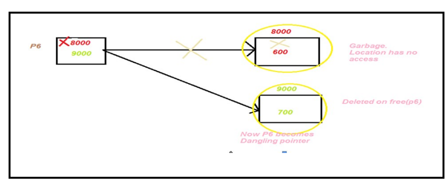


Garbage/Memory Leak

Garbage is a **location which has no name and hence no access**. If the same pointer is allocated memory more than once using the DMA functions, initially allocated spaces becomes garbage. **Garbage in turn results in memory leak. Memory leak can happen only in Heap region**.

Coding Example_6:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{     int *p6 = (int*)malloc(sizeof(int));
      *p6 = 600;
      printf(" %d\n", *p6); // 600
      p6  =  (int*)malloc(sizeof(int));
// changing p6 loses the pointer to the location allocated by the previous malloc
// So memory allocated by previous malloc has no access. Hence becomes garbage
      *p6 = 700;
      printf(" %d\n", *p6);
      free(p6);
      return 0;
}
```



Double free error: DO NOT TRY THIS

- If free() is used on a memory that is already freed before.
- Leads to undefined behavior.
- Might corrupt the state of the memory manager that can cause existing blocks of memory to get corrupted or future allocations to fail.
- Can cause the program to crash or alter the execution flow.

Coding Example_7:

```
int* p = (int*)malloc(sizeof(int));
*p = 10;
printf("p = %d", *p);
free(p);
free(p);
```

The above code might lead to undefined behavior. So, it is a good practice to make the pointer NULL after the usage of free. By chance, if the pointer is freed again, it doesn't do anything with NULL.

```
free(p);
p = NULL; //Function free does nothing with a NULL pointer.
free(p);
```

NOTE:

- **Dereferencing the dangling pointer results in undefined behavior.**
- **Dereferencing the NULL Pointer results in guaranteed crash.**

Coding Example_8: Uninitialized Pointer use - Using a pointer without assigning valid memory.

```
int *ptr;        *ptr = 5; // Undefined behavior
```

Coding Example_9: Memory overrun - Writing outside the bounds of allocated memory.

```
int *ptr = malloc(3 * sizeof(int));      ptr[3] = 10; // Index out of bounds!
```

Coding Example_10: Invalid free - Trying to free memory that was not dynamically allocated.

```
int a = 5;        int *ptr = &a;  free(ptr); // Invalid: memory not allocated via malloc
```

Best Practices/Guidelines to be followed while using the Dynamic Memory Management functions.

- **Always check for NULL:** After calling malloc, calloc, or realloc, verify if the returned pointer is **not NULL** before using it.
```
int *arr = malloc(n * sizeof(int));
if (arr == NULL) {
   // Handle memory allocation failure
}
```
- **Initialize memory when needed:** Use calloc if you need zero-initialized memory instead of malloc + manual initialization.
- **Avoid memory leaks:** Always free any dynamically allocated memory once it is no longer needed.
- **Avoid dangling pointers**: After freeing memory, **set the pointer to NULL** to avoid accidental access.
- **Be cautious with realloc:** Store the result of realloc in a **temporary pointer**, check for NULL, and then assign.
- **Use correct size calculations:** Always multiply the number of elements with sizeof(datatype) during allocation. To allocate 10 locations to store integers,
```
int *p = (int*) malloc(10) ; //  Wrong
int *p = (int*) malloc(10*sizeof(int));  // Perfect
```
- **Match allocation and deallocation:** Every malloc/calloc/realloc must be matched with one free. Use consistent naming to track ownership.

Happy managing memory smartly with DMM!!