

Symbolic Analysis from Source Code

```
void test(int x, int y){  
  
    for(int i=1;i<y;i++){  
        if(x%y == 0){  
            ERROR;  
        }  
    }  
}
```

Dynamic memory management errors in C can lead to various issues such as memory leaks, buffer overflows, dangling pointers, and segmentation faults. Here are some common errors and how to avoid them:

- **Memory Leaks:** Memory leaks occur when dynamically allocated memory is not properly deallocated, leading to a loss of available memory over time. To avoid memory leaks, always make sure to free dynamically allocated memory using the `free()` function when it is no longer needed.

```
int *ptr = (int *)malloc(sizeof(int));  
  
if (ptr == NULL) { // handle allocation failure }  
  
free(ptr); // Free memory when done using it
```

- **Dangling Pointers:** Dangling pointers occur when a pointer continues to point to a memory location that has been deallocated. To avoid dangling pointers, always set pointers to `NULL` after freeing them.

Ex-1:

```
int *ptr = malloc(sizeof(int));  
*ptr = 10;  
free(ptr);  
*ptr = 20; // ptr is now a dangling pointer
```

Corrected:

```
int *ptr = malloc(sizeof(int));  
*ptr = 10;  
free(ptr);  
  
ptr = NULL; // to avoid dangling pointers, set pointer to NULL after freeing
```

Ex-2:

```
void foo() {  
    int *ptr = malloc(sizeof(int));  
    *ptr = 10;  
    return; // ptr is now a dangling pointer  
}  
int main() {  
    foo();  
    *ptr = 20; // accessing the memory location pointed to by ptr can lead to  
    undefined behavior
```

```
    return 0;
}
```

Ex-3:

```
char *ptr;
char *get_string() {
    char buffer[100];
    scanf("%s", buffer);
    ptr = buffer; // ptr is now a dangling pointer
    return ptr;
}
int main() {
    char *str = get_string();
    printf("%s\n", str); // accessing the memory location pointed to by str can lead
    to undefined behavior
    return 0;
}
```

Solve:

```
char *get_string(char *ptr) {
    scanf("%s", ptr);
    return ptr;
}
int main() {
    char *str = malloc(100 * sizeof(char));
    str = get_string(str);
    printf("%s\n", str); // str is not a dangling pointer
    free(str);
    return 0;
}
```

- **Buffer Overflows:**

Buffer overflows occur when data is written beyond the bounds of an allocated memory buffer, potentially overwriting adjacent memory locations and leading to undefined behavior. To avoid buffer overflows, always ensure that you allocate sufficient memory for your data and use functions that perform bounds checking, such as fgets() instead of gets() for reading input from the user.

Ex-1:

```
char buffer[10]; // Allocate sufficient memory
fgets(buffer, sizeof(buffer), stdin); // Use fgets instead of gets
```

- **Use After Free:** Use after free errors occur when a program attempts to access memory that has already been deallocated. Avoid such errors by ensuring that memory is not accessed after it has been freed.

```
int *ptr = (int *)malloc(sizeof(int));
free(ptr);

// Accessing ptr here would result in undefined behavior
```

- **Uninitialized Memory:** Accessing uninitialized memory can lead to unpredictable behavior. Always initialize dynamically allocated memory before using it.

C

Copy code

```
int *ptr = (int *)malloc(sizeof(int)); *ptr = 10; // Initialize memory before using it
```

- **Memory Fragmentation:** Memory fragmentation occurs when free memory becomes divided into small, unusable chunks. To avoid memory fragmentation, use memory allocation and deallocation consistently and try to allocate memory in contiguous blocks whenever possible.

C

Copy code

```
int *ptr = (int *)malloc(sizeof(int) * n); // Allocate memory in contiguous blocks
```

By being aware of these common dynamic memory management errors and following best practices, you can write safer and more reliable C programs. Additionally, tools like Valgrind can help detect memory errors by analyzing memory usage and detecting issues such as memory leaks and invalid memory accesses.