Allocation for Strings

String Function

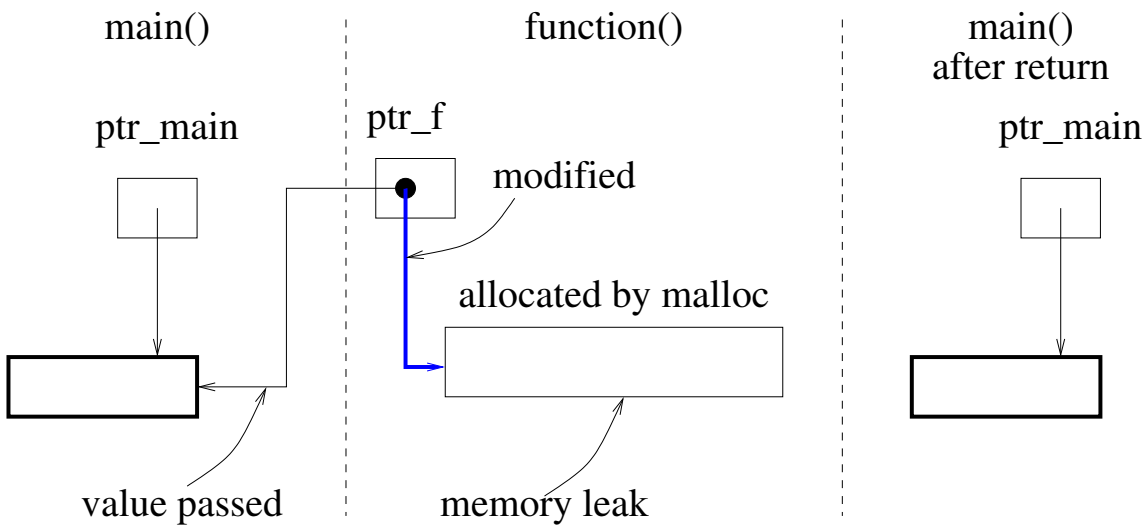
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char *concat(const char *s1, const char *s2) {
    char *result;

    result = malloc(strlen(s1) + strlen(s2) + 1);
    if (result != NULL) {
        strcpy(result, s1);
        strcat(result, s2);
    }
    return result;
}
int main() {
   char *str1 = "My String", *str2 = " Your string";
   printf("%s\n", concat(str1,str2));
}
```

Allocation for Strings

Using Pointers to Pointers

- When arguments to a function are pointers, we may want function to modify them.
- Passing single pointer would not work: results in memory leak.
- Consider a function for reading a string



Allocation for Strings

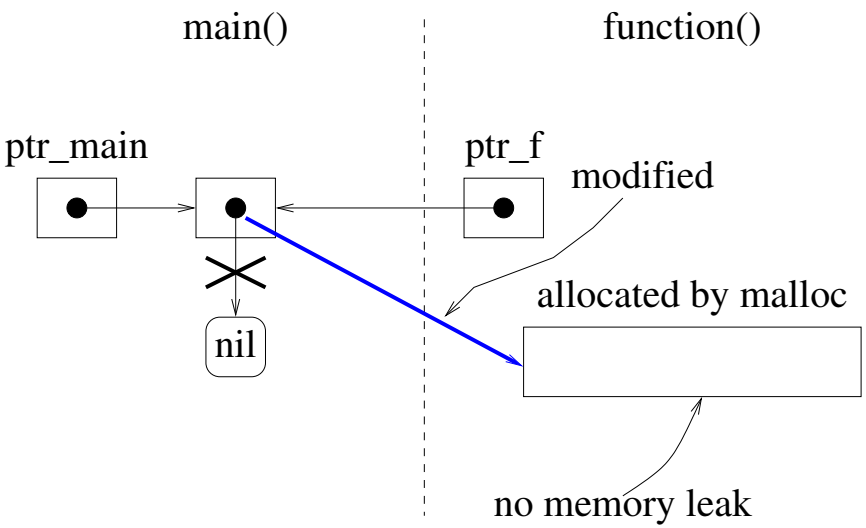
Using Pointers to Pointers

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void singlePtr( char * ptr) {
    ptr = malloc(50); // allocate some memory
    strcpy(ptr, "Hello World");
}
int main() {
    char *ptr = 0;
    singlePtr( ptr );
    printf("%p\n", ptr);
    free(ptr);
}
```

- Prints (nil).

Allocation for Strings

Using Pointers to Pointers



- Using `malloc/calloc`, a pointer to the memory block is obtained.
- The pointer returned back to the caller and holds a pointer to allocated space.

Allocation for Strings

Reading a Line of Text

- The function `readLine` appears below.

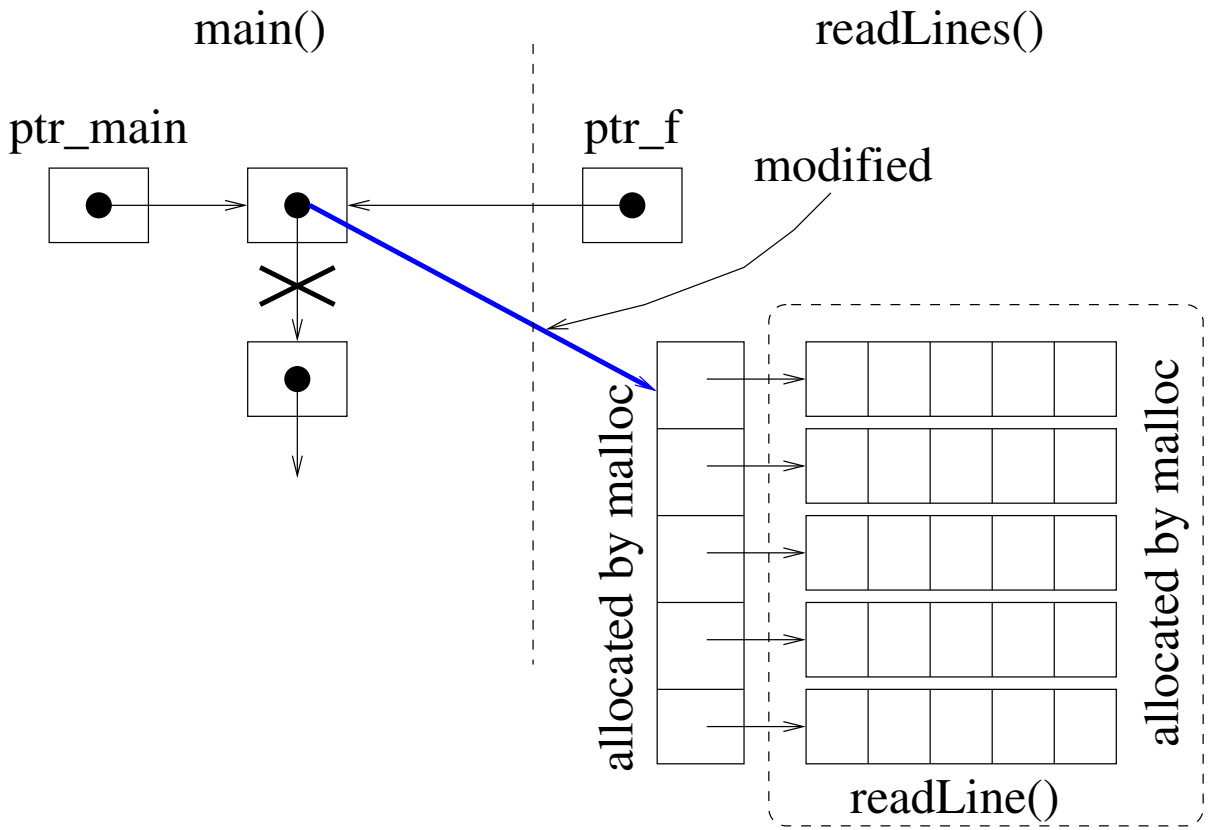
```
int readLine(char **str) {
    char buf[256];
    int len; // Length of string to read

    gets(buf); // Read the string into buf
    len = strlen(buf); // Count the null character
    if (buf[len-1] == '\n') // Remove '\n' and pad with '\0'
        buf[len - 1] = '\0'; // length less than 256

    *str = (char *) calloc(len, sizeof(char)); // Allocate space
    strcpy(*str, buf); // Copy input string into space allocated
    return len; // Return length of string read
}
```

Allocation for Strings

Reading Several Lines of Text



Allocation for Strings

Reading Several Lines of Text

- `malloc` returns a pointer to object.
- So, in this case triple level indirection is needed.

```
int readLines(char ***pstrs) {
    int i, n;

    printf("Enter number of strings: ");
    scanf("%d", &n);
    getchar(); // skip new line

    *pstrs = (char **) calloc(n, sizeof(char *));
    for (i = 0; i < n; i++) {
        printf("Enter string %d: ", i+1);
        readLine(&(*pstrs)[i]); //
    }
    return n;
}
```

Allocation for Arrays

Allocation Using Malloc

- Used in the same way as was done for strings. The difference being: the size of the elements of arrays could be different for different arrays.
- E.g., in case of an array of integers: `a = malloc(n * sizeof(int));`
- After allocation is done, `a` is used as the name of an array, ignoring the fact that it is a pointer.

Allocation Using Calloc

- It clears memory, so sometimes we use `calloc` instead of `malloc`.
- Use: `a = calloc(n, sizeof(int));`

Allocation for Arrays

Allocation Using Realloc

- Dynamically, we may find allocated memory is bigger/smaller than the requirement.
- Requires pointer to point to the memory block returned by a previous `malloc/calloc/realloc`.
- Requires size parameter (defining new size).
- Does not initialize bytes that are added to the block
- If block can not be enlarged returns null, old block unchanged.
- If called with null pointer, behaves like `malloc`
- If size parameter is 0 then behaves like `free`