# Dangling Pointers Avoid them Strictly!

This article takes off from an earlier one on constant pointers. Here, the focus is on dangling pointers—how they occur and how to prevent them. The article is a great guide for C newbies.

Dangling pointers are those in the C programming language that do not point to valid objects. Pointers are like sharp knives that have tremendous power in C programming. When used properly, they reduce the complexity of the programs to a great extent. But when not used properly, they result in adverse effects and, in the worst case scenario, may crash the system.

In this article, I mainly focus on dangling pointers and what causes them by looking at several different situations in which they occur. I also suggest simple methods to avoid them.

**Note:** Code snippets provided here are tested with the GCC compiler [gcc version 4.7.3] running under the Linux environment.

Let's now look at three different cases that give rise to dangling pointers.

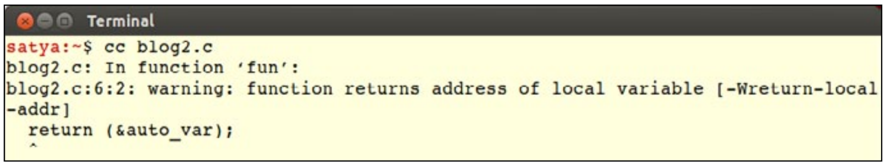
## Case 1: When a function returns the address of the auto variables

Consider the simple code snippet given below (Code 1):

```
#include <stdio.h>

int *fun(void);

int main()
{
        int *int_ptr = fun();
        printf("The value at address %p : %d\n", int_ptr,
  *int_ptr);
```



```
satya:~$ cc blog2.c
blog2.c: In function 'fun':
blog2.c:6:2: warning: function returns address of local variable [-Wreturn-local
-addr]
  return (&auto_var);
  ^
```

Figure 1: Compiler generating warning

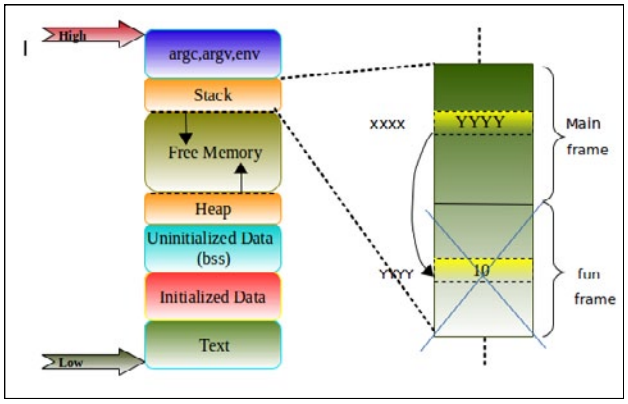


Figure 2: Program memory and stack frames

```
        return 0;
}

int *fun(void)
{
        int auto_var = 10;
        return (&auto_var);
}
```

When we run the program shown in Code 1 in the GCC compiler, we get the warning shown in Figure 1.

What is wrong when the address of the auto/local variables is returned? Let us examine this in detail:

1. We know that whenever there is a function call, a new stack frame will be created automatically, where *auto variable auto_var* is local in our example. Its scope and lifetime is within the function call.
2. When the control returns from the function call, all the memory allocated for that function will be freed automatically.
3. In our example program, we are returning the address of the auto variable and collecting this in the *int_ptr* pointer in the main function. So, *int_ptr* is still pointing to the memory, which is freed as mentioned in Point 2.
4. One can observe allocation and deallocation of the stack frame (let us called this *fun frame*) for the *fun()* in Figure 2 for better understanding.
5. Now, *int_ptr* becomes a dangling pointer. Dereferencing this pointer results in *unexpected output.*
6. So, always take extra care while playing with pointers and local variables.

Any attempt to dereference the pointer that is already dangling may still print the correct value after the control returning from a function call, but any functions called thereafter will overwrite the stack storage allocated for the *auto_var* variable with other values, and the pointer will no longer work correctly.

**How to prevent a pointer from becoming a dangling pointer in this case:** If a pointer to *auto_var* is to be returned, *auto_var* must have scope beyond the function so that it may be declared as static in order to avoid the pointer from dangling. This is because the memory allocated to the static variables is from the data segment, where the lifetime will be throughout the program.

## Case 2: When the variable goes out of scope

Consider the sample code (Code 2) given below for analysis:

```
 1 #include <stdio.h>
 2
 3 int main()
 4 {
 5         int *iptr;
 6         //Block started
 7         {
 8                 int var = 10;
 9                 iptr = &var;
10         } //After this block iptr is dangling
11         //Some code goes here
12         return 0;
13 }
```

Running the program shown in Code 2 in the GCC compiler with the *-Wall* option results in the warning shown in Figure 3.

Since the variable *var* is invisible for the outer block shown in Code 2, *iptr* is still pointing to the same object even when the control comes out of the inner block. Hence, the pointer *iptr* becomes a dangling pointer after Line 10 in the example shown in Code 2.

## Case 3: When, in dynamic memory allocation, the block of memory that is already freed is used

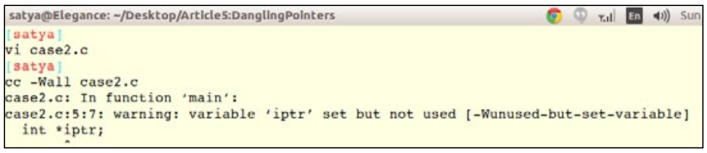
Consider the sample code given below:

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


int main()
{
int *block_ptr = (int *) malloc(sizeof(int));

//Do something with allocated memory

free(block_ptr);
```



```
satya@Elegance: ~/Desktop/Article5:DanglingPointers
[satya]
vi case2.c
[satya]
cc -Wall case2.c
case2.c: In function 'main':
case2.c:5:7: warning: variable 'iptr' set but not used [-Wunused-but-set-variable]
  int *iptr;
      ^
```

Figure 3: Compiler generating warning

```
12
13 //Some statements
14
15 *block_ptr = 20;
16 //Pointer becomes dangling, since the memory
17 //block to which it is pointing is already freed
18
19 return 0;
20 }
```

In the code snippet shown above,

Line 5: Memory allocation by *malloc()*.

Line 9: Memory allocated is freed by *free()* manually.

Line 13: Reusing the pointer, which is still pointing to the memory that is already freed. In our example, *block_ptr* is now the dangling pointer.

**Note:** In Case 1: Memory is freed automatically. In Case 3: Memory is freed manually. This is one of the key differences between stack and heap.

In the C programming language, deleting an object from memory explicitly or by destroying the stack frame on the return of the control does not alter associated pointers as seen in Case 1 and Case 3. The pointer still points to the same location in memory, even though the reference has since been deleted and may now be used for other purposes.

**Solution for the problem in Case 3:** In Case 3, the dangling pointer can be avoided by initialising it to NULL immediately after freeing it, if the OS is capable of detecting the runtime references as shown below:

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

int main()
{
int *block_ptr = (int *) malloc(sizeof(int));

//Do something with allocated memory
free(block_ptr);
//Initialising the block pointer to NULL
block_ptr = NULL;
//Now, block_ptr is no more dangling
//Some statements

*block_ptr = 20;
//Error: Dereferencing the NULL pointer, gives
segmentation fault
return 0;
}
```

Dangling pointers are very harmful and have adverse effects in embedded systems programming. So, they should be strictly avoided. END

**By: Satyanarayana Sampangi**

The author is a member of the embedded software team at Emertxe Information Technology (P) Ltd *(http://www.emertxe.com)*. His areas of interest are embedded C programming combined with data structures and microcontrollers. He can be reached at *satya@emertxe.com*