Dangling and Wild Pointers in C/C++

acarlstein.com/

Posted by Alejandro G. Carlstein Ramos Mejia on October 20, 2010 February 9, 2012 About Programming / ANSI/POSIX C / C++

When we define a variable, memory is allocated for that variable.

```
/* definition of a variable with a size memory allocated of one byte */
char character;
```

This declare that the name *character* is of type *char*.

As you may know, we use pointers in order to manipulate data in the memory. Pointers are designed to store a value which is a memory address. However, if a defined pointers is not instantiated then this pointer may have a memory address pointing to any place in memory. This is pointer is called a “dangling pointer”. For example,

```
/* Dangling pointer */
char* c_pointer;

char character;
char* c_char_pointer = &character;  /* No dangling pointer */
```

Lets assume we defined a dangling pointer and we attempt to print the content of the memory address that may store in the dangling pointer:

```
int* dangling_pointer;
printf('%d', *dangling_pointer);
```

This would give us *segmentation fault.*

Lets assume we define a pointer, *integer_pointer*, and we make sure that this pointer is not dangling by instantiated with NULL. Later in the our program, we define a variable inside brackets, *integer_variable*, and we assign the address of this variable to our pointer. The moment we get out of the brackets, the variable *integer_variable* will be out of scope (This means that this variable disappear in our program).

```
int main(int argc, char* argv[]){

  int* integer_pointer = NULL;
  ... /* Dots means that there are more code not included in the example */

  { /* integer_variable exist only between this brackets */
    int integer_variable = 100;
    integer_pointer = &integer_variable;
    ...
  } /* After this bracket, integer_variable is out of scope */
  ...

  *integer_pointer = 150;
  ...

  return 0;
}
```

This could produce unpredictable behaviour because the address store in the *integer_pointer* could be pointing to a sector in memory used for another program or another part of the program.

When using dynamic memory allocation such as *malloc* or *new*, the moment that we free the pointer, it becomes dangling. Therefore, it is a good policy to point the pointer to NULL. Why? Because the memory that was obtained by *malloc* or *new* will not exist anymore since it was freed by *free*.

```
#include <malloc.h>
...
int main(int argc, char* argv[]){
  ...

  char* character_pointer = malloc(sizeof(char));
  ...

  /* character_pointer is freed and become a dangling pointer */
  free(character_pointer);

  /* Point the pointer to NULL so it is not dangling anymore */
  character_pointer = NULL;
  ...

  return 0;
}
```

Sometimes is said that an dangling pointer is a wild pointer; however, there is a distinction. While a dangling pointer is a wild pointer, a wild pointer may not be a dangling pointer… What?!! (You may ask) Let me explain.

The different is the instantiation of the variable. Lets define to pointers:

```
char* character_pointer_1;
char* character_pointer_2;
```

Both pointers can be called dangling and/or wild pointers because they are not instantiated. Now lets assume we instantiate *character_pointer_1*.

```
char* character_pointer_1;
char* character_pointer_2;
...
character_pointer_1 = malloc(sizeof(char));
...
free(character_pointer_1);
...
```

In this case, things change because *character_pointer_1* was instantiated and later freed (with *free*). *character_pointer_1* is called dangling pointer for the fact that it was instantiated before, while *character_pointer_2* is called wild because it was never instantiated.