I was wondering if anyone would like to augment the information I will receive from a series of hopeful google searches with their own answer to this question:

When I declare a function which returns a value, but then I call that function in my code without assigning the returned value to anything... what happens to it? I don't get a compiler warning even with the extra, and so forth. Where in the green digital void does this value end up and why does the codeGod, usually so viciously fickle and cruel suddenly deem such an oversight perfectly acceptable?!

In example:

```c
savageBurn *makeBurnMoreSavage(savageBurn *mildBurn, int desiredBurnSeverity);
```

But then I call the function without assigning the augmented burn, thusly:

```c
savageBurn *lukeWarmInsult = newSavageBurn(char* iDoNotLikeYourInsults);
makeBurnMoreSavage(savageBurn *lukeWarmInsult, *9001);
displayBurn(savageBurn *lukeWarmInsult);
```

The original value is altered, and all is well. However, I wonder what becomes of the pointer that was supposed to be returned logically, but is ignored syntactically. Poor little guy... all alone out there... probably cold... scared.

---

There are compiler options that detect thrown-away return values. The conscientious programmer indicates that intent with a cast to void, as in:

```c
savageBurn *lukeWarmInsult = newSavageBurn(char* iDoNotLikeYourInsults);
(void) makeBurnMoreSavage(savageBurn *lukeWarmInsult, *9001);
displayBurn(savageBurn *lukeWarmInsult);
```

---

Thank you for your response!
I have been using this method since you mentioned it in response to a fellow student's question in class.

If you have a moment to elaborate just a bit:

I wonder, in the case of no return set, and no void notation, does the value being 'returned' just get swept away with the other temporarily instantiated variables from the function when control returns to the call point?

I have found this a very tedious question to google with most explanations of what happens to the 'contents' of a called function after control returns not touching directly on this case.

Subject: Re: Always acknowledge your return values! Or don’t. But do.
Posted by padietl on Thu, 16 Mar 2017 19:33:40 GMT
The explanation really requires a look at assembly.

Here's an example. Say you have this C program:

```c
#include <stdio.h>

int add1(int x)
{
    return 1 + x;
}

int main()
{
    int a = 1;
    int result = add1(a);
    printf("%d + 1 = %d\n", a, result);
    return 0;
}
```

It produces the result:

```
1 + 1 = 2
```

Now let's have a look at the gcc x86_64 assembly. I have added some comments to attempt to explain what is going on.

In case you have never encountered assembly before, note that

```assembly
mov a, b
```
means move b into a.
NOT a into b.
It’s a little backwards.

.intel_syntax noprefix
.text
.add1:
    mov eax, edi    # move the argument int eax
    add eax, 1      # add one to eax
    # return value always put into eax
    ret

.section .rodata
.formatString:
    .string "%d + 1 = %d\n"

.globl main
main:
    push rbp
    mov rbp, rsp
    sub rsp, 16
    mov DWORD PTR [rbp-8], 1        # (rbp - 8) is the address of a. [] means dereference
    # We are basically doing *(&a) = 1
    mov edi, DWORD PTR [rbp-8]      # add1’s first argument needs to be in the edi 32-bit register
    call add1                      # Call the function

    # Result of a function is always stored in the eax register
    mov DWORD PTR [rbp-4], eax      # Move the result out of the register and into the 'result'
variable

    # argumets to printf need to be stored in order from left
    # to right in registers edi esi edx
    mov edi, OFFSET FLAT:.formatString      # copy format string
    mov esi, DWORD PTR [rbp-8]          # copy variable 'a' into register eax
    mov edx, DWORD PTR [rbp-4]          # copy variable 'result' into register edx
    call printf

    mov eax, 0      # main returns 0
    leave
    ret
The gist of what this is saying is that the return value is always placed the eax register, thus the calling function is free to completely ignore whatever is placed in that register after having called a function. It's perfectly safe and legal to do so. Gcc just wants to let you know that you declared a function which is supposed to return something, but you are ignoring it, which is possibly an error.

Subject: Re: Always acknowledge your return values! Or don't. But do. 
Posted by bkaaron on Thu, 16 Mar 2017 23:11:09 GMT

Damn fine response, good sir. Thank you kindly!

Subject: Re: Always acknowledge your return values! Or don't. But do. 
Posted by padietl on Fri, 17 Mar 2017 20:54:48 GMT

np, yo.