Chapter 62: Common C programming idioms and developer practices

Section 62.1: Comparing literal and variable

Suppose you are comparing value with some variable

```
if ( i  == 2) //Bad-way
{
    doSomething;
}
```

Now suppose you have mistaken `==` with `=`. Then it will take your sweet time to figure it out.

```
if( 2 == i) //Good-way
{
    doSomething;
}
```

Then, if an equal sign is accidentally left out, the compiler will complain about an “attempted assignment to literal.” This won’t protect you when comparing two variables, but every little bit helps.

See here for more info.

Section 62.2: Do not leave the parameter list of a function blank — use void

Suppose you are creating a function that requires no arguments when it is called and you are faced with the dilemma of how you should define the parameter list in the function prototype and the function definition.

- You have the choice of keeping the parameter list empty for both prototype and definition. Thereby, they look just like the function call statement you will need.
- You read somewhere that one of the uses of keyword ***void*** (there are only a few of them), is to define the parameter list of functions that do not accept any arguments in their call. So, this is also a choice.

So, which is the correct choice?

ANSWER: using the keyword ***void***

GENERAL ADVICE: If a language provides certain feature to use for a special purpose, you are better off using that in your code. For example, using **`enum`**s instead of `#define` macros (that's for another example).

C11 section 6.7.6.3 "Function declarators", paragraph 10, states:

> The special case of an unnamed parameter of type void as the only item in the list specifies that the function has no parameters.

Paragraph 14 of that same section shows the only difference:

> … An empty list in a function declarator that is part of a definition of that function specifies that the

> function has no parameters. The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.

A simplified explanation provided by K&R (pgs- 72-73) for the above stuff:

> Furthermore, if a function declaration does not include arguments, as in `double atof();`, that too is taken to mean that nothing is to be assumed about the arguments of `atof`; all parameter checking is turned off. This special meaning of the empty argument list is intended to permit older C programs to compile with new compilers. But it's a bad idea to use it with new programs. If the function takes arguments, declare them; if it takes no arguments, use `void`.

So this is how your function prototype should look:

```
int foo(void);
```

And this is how the function definition should be:

```
int foo(void)
{
    ...
    <statements>
    ...
    return 1;
}
```

One advantage of using the above, over `int foo()` type of declaration (ie. without using the keyword ***void***), is that the compiler can detect the error if you call your function using an erroneous statement like `foo(42)`. This kind of a function call statement would not cause any errors if you leave the parameter list blank. The error would pass silently, undetected and the code would still execute.

This also means that you should define the `main()` function like this:

```
int main(void)
{
    ...
    <statements>
    ...
    return 0;
}
```

Note that even though a function defined with an empty parameter list takes no arguments, it does not provide a prototype for the function, so the compiler will not complain if the function is subsequently called with arguments. For example:

```
#include <stdio.h>

static void parameterless()
{
    printf("%s called\n", __func__);
}

int main(void)
{
    parameterless(3, "arguments", "provided");
    return 0;
}
```

If that code is saved in the file `proto79.c`, it can be compiled on Unix with GCC (version 7.1.0 on macOS Sierra 10.12.5 used for demonstration) like this:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o proto79
$
```

If you compile with more stringent options, you get errors:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-style-
definition -pedantic proto79.c -o proto79
proto79.c:3:13: error: function declaration isn't a prototype [-Werror=strict-prototypes]
 static void parameterless()
             ^~~~~~~~~~~~~
proto79.c: In function 'parameterless':
proto79.c:3:13: error: old-style function definition [-Werror=old-style-definition]
cc1: all warnings being treated as errors
$
```

If you give the function the formal prototype `static void parameterless(void)`, then the compilation gives errors:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-style-
definition -pedantic proto79.c -o proto79
proto79.c: In function 'main':
proto79.c:10:5: error: too many arguments to function 'parameterless'
     parameterless(3, "arguments", "provided");
     ^~~~~~~~~~~~~
proto79.c:3:13: note: declared here
 static void parameterless(void)
             ^~~~~~~~~~~~~
$
```

Moral — always make sure you have prototypes, and make sure your compiler tells you when you are not obeying the rules.