Understanding “extern” keyword in C

- Difficulty Level : Medium
- Last Updated : 16 Nov, 2020

I’m sure this post will be as interesting and informative to C virgins (i.e. beginners) as it will be to those who are well-versed in C. So let me start by saying that the extern keyword applies to C variables (data objects) and C functions. Basically, the extern keyword extends the visibility of the C variables and C functions. That’s probably the reason why it was named extern.
Though most people probably understand the difference between the “declaration” and the “definition” of a variable or function, for the sake of completeness, I would like to clarify them.

- **Declaration** of a variable or function simply declares that the variable or function exists somewhere in the program, but the memory is not allocated for them. The declaration of a variable or function serves an important role–it tells the program what its type is going to be. In case of *function* declarations, it also tells the program the arguments, their data types, the order of those arguments, and the return type of the function. So that’s all about the declaration.
- Coming to the **definition**, when we *define* a variable or function, in addition to everything that a declaration does, it also allocates memory for that variable or function. Therefore, we can think of definition as a superset of the declaration (or declaration as a subset of definition).

A variable or function can be *declared* any number of times, but it can be *defined* only once. (Remember the basic principle that you can’t have two locations of the same variable or function).
Now back to the extern keyword. First, Let’s consider the use of extern in functions. It turns out that when a function is declared or defined, the extern keyword is implicitly assumed. When we write.

```
int foo(int arg1, char arg2);
```

The compiler treats it as:

```
extern int foo(int arg1, char arg2);
```

Since the extern keyword extends the function’s visibility to the whole program, the function can be used (called) anywhere in any of the files of the whole program, provided those files contain a declaration of the function. (With the declaration of the function in place, the compiler knows the definition of the function exists somewhere else and it goes ahead and compiles the file). So that’s all about extern and functions.
Now let’s consider the use of extern with variables. To begin with, how would you *declare* a variable without *defining* it? You would do something like this:

```
extern int var;
```

Here, an integer type variable called var has been declared (it hasn't been defined yet, so no memory allocation for var so far). And we can do this declaration as many

times as we want. So far so good.
Now, how would you *define* var? You would do this:

```
int var;
```

In this line, an integer type variable called var has been both declared **and** defined (remember that *definition* is the superset of *declaration*). Since this is a *definition*, the memory for var is also allocated. Now here comes the surprise. When we declared/defined a function, we saw that the extern keyword was present implicitly. But this isn't the case with variables. If it were, memory would never be allocated for them. Therefore, we need to include the extern keyword explicitly when we want to declare variables without defining them. Also, as the extern keyword extends the visibility to the whole program, by using the extern keyword with a variable, we can use the variable anywhere in the program provided we include its declaration the variable is defined somewhere.
Now let us try to understand extern with examples.
Example 1:

- C

```
int var;

int main(void)

{

    var = 10;

    return 0;

}
```

This program compiles successfully. var is defined (and declared implicitly) globally.
Example 2:

- C

```
extern int var;

int main(void)

{

  return 0;

}
```

This program compiles successfully. Here var is declared only. Notice var is never used so no problems arise.
Example 3:

- C

```
extern int var;

int main(void)

{

  var = 10;

  return 0;

}
```

This program throws an error in the compilation(during the linking phase, more info here) because var is declared but not defined anywhere. Essentially, the var isn't allocated any memory. And the program is trying to change the value to 10 of a variable that doesn't exist at all.
Example 4:

- C

```
#include "somefile.h"

extern int var;

int main(void)

{

 var = 10;

 return 0;

}
```

Assuming that somefile.h contains the definition of var, this program will compile successfully.
Example 5:

- C

```
extern int var = 0;

int main(void)

{

 var = 10;

 return 0;

}
```

Do you think this program will work? Well, here comes another surprise from C standards. They say that..if a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated– in other words, that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work.

So that was a preliminary look at the extern keyword in C.
In short, we can say:

1. A declaration can be done any number of times but definition only once.
2. the extern keyword is used to extend the visibility of variables/functions.
3. Since functions are visible throughout the program by default, the use of extern is not needed in function declarations or definitions. Its use is implicit.
4. When extern is used with a variable, it’s only declared, not defined.
5. As an exception, when an extern variable is declared with initialization, it is taken as the definition of the variable as well.