WIKIPEDIA

External variable

In the C programming language, an **external variable** is a variable defined outside any function block. On the other hand, a local (automatic) variable is a variable defined inside a function block.

> As an alternative to automatic variables, it is possible to define variables that are external to all functions, that is, variables that can be accessed by name by any function. (This mechanism is rather like Fortran COMMON or Pascal variables declared in the outermost block.) Because external variables are globally accessible, they can be used instead of argument lists to communicate data between functions. Furthermore, because external variables remain in existence permanently, rather than appearing and disappearing as functions are called and exited, they retain their values even after the functions that set them have returned.
>
> — The C Programming Language

Contents

Definition, declaration and the `extern` keyword

To understand how external variables relate to the `extern` keyword, it is necessary to know the difference between defining and declaring a variable. When a variable is defined, the compiler allocates memory for that variable and possibly also initializes its contents to some value. When a variable is declared, the compiler requires that the variable be defined elsewhere. The declaration informs the compiler that a variable by that name and type exists, but the compiler does not need to allocate memory for it since it is allocated elsewhere. The `extern` keyword means "declare without defining". In other words, it is a way to explicitly declare a variable, or to force a declaration without a definition. It is also possible to explicitly define a variable, i.e. to force a definition. It is done by assigning an initialization value to a variable. If neither the `extern` keyword nor an initialization value are present, the statement can be either a declaration or a definition. It is up to the compiler to analyse the modules of the program and decide.

A variable must be defined exactly once in one of the modules of the program. If there is no definition or more than one, an error is produced, possibly in the linking stage. A variable may be declared many times, as long as the declarations are consistent with each other and with the definition (something which header files facilitate greatly). It may be declared in many modules, including the module where it was defined, and even many times in the same module. But it is usually pointless to declare it more than once in a module.

An external variable may also be declared inside a function. In this case the `extern` keyword must be used, otherwise the compiler will consider it a definition of a local (automatic) variable, which has a different scope, lifetime and initial value. This declaration will only be visible inside the function instead of throughout the function's module.

The `extern` keyword applied to a function prototype does absolutely nothing (the `extern` keyword applied to a function definition is, of course, non-sensical). A function prototype is always a declaration and never a definition. Also, in standard C, a function is always external, but some compiler extensions allow a function to be defined inside a function.

> An external variable must be defined, exactly once, outside of any function; this sets aside storage for it. The variable must also be declared in each function that wants to access it; this states the type of the variable. The declaration may be an explicit `extern` statement or may be implicit from context. ... You should note that we are using the words definition and declaration carefully when we refer to external variables in this section. Definition refers to the place where the variable is created or assigned storage; declaration refers to places where the nature of the variable is stated but no storage is allocated.
>
> — The C Programming Language

Scope, lifetime and the `static` keyword

An external variable can be accessed by all the functions in all the modules of a program. It is a global variable. For a function to be able to use the variable, a declaration or the definition of the external variable must lie before the function definition in the source code. Or there must be a declaration of the variable, with the keyword `extern`, inside the function.

The `static` keyword (`static` and `extern` are mutually exclusive), applied to the definition of an external variable, changes this a bit: the variable can only be accessed by the functions in the same module where it was defined. But it is possible for a function in the same module to pass a reference (pointer) of the variable to another function in another module. In this case, even though the function is in another module, it can read and modify the contents of the variable —it just cannot refer to it by name.

It is also possible to use the `static` keyword on the definition of a local variable. Without the `static` keyword, the variable is automatically allocated when the function is called and released when the function exits (thus the name "automatic variable"). Its value is not retained between function calls. With the `static` keyword, the variable is allocated when the program starts and released when the program ends. Its value is not lost between function calls. The variable is still local, since it can only be accessed by name inside the function that defined it. But a reference (pointer) to it can be passed to another function, allowing it to read and modify the contents of the variable (again without referring to it by name).

External variables are allocated and initialized when the program starts, and the memory is only released when the program ends. Their lifetime is the same as the program's.

If the initialization is not done explicitly, external (static or not) and local static variables are initialized to zero. Local automatic variables are uninitialized, i.e. contain "trash" values.

The `static` keyword applied to a function definition changes the linkage of the function so that it is only visible from the translation unit where its definition is located. This prevents the function from being called by name from outside its module (it remains possible to pass a function pointer out of the module and use that to invoke the function). Declaring a function using the `static` keyword is also a good way to keep its name short while avoiding name clashes.

Example (C programming language)

File 1:

```
// Explicit definition, this actually allocates
// as well as describing
int Global_Variable;

// Function prototype (declaration), assumes
// defined elsewhere, normally from include file.
void SomeFunction(void);

int main(void) {
  Global_Variable = 1;
  SomeFunction();
  return 0;
}
```

File 2:

```
// Implicit declaration, this only describes and
// assumes allocated elsewhere, normally from include
extern int Global_Variable;

// Function header (definition)
void SomeFunction(void) {
  ++Global_Variable;
}
```

In this example, the variable Global_Variable is defined in File 1. In order to utilize the same variable in File 2, it must be declared. Regardless of the number of files, a global variable is only defined once; however, it must be declared in any file outside of the one containing the definition.

> If the program is in several source files, and a variable is defined in file1 and used in file2 and file3, then extern declarations are needed in file2 and file3 to connect the occurrences of the variable. The usual practice is to collect extern declarations of variables and functions in a separate file, historically called a header, that is included by #include at the front of each source file. The suffix .h is conventional for header names.
>
> — The C Programming Language

The normal methodology is for allocation and actual definitions to go into .c files, but mere declarations and prototypes do not allocate and just describe the types and parameters so that the compiler can work correctly, and that information belongs in a .h header file that others can safely include without any possible conflict.

See also

- *The C Programming Language*
- Declaration
- Global variable
- Local variable
- Static variable
- Scope
- Function prototype

References

External links

- Microsoft C Language Reference: The extern Storage-Class Specifier (http://msdn.microsoft.com/en-us/library/2tx32sw2(VS.80).aspx)
- "The C Standard (C99 with Technical corrigenda TC1, TC2, and TC3 included)" (http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf) (PDF). (3.61 MiB). In particular, see sections 6.2.2 (Linkage of identifiers), 6.2.4 (Storage duration of objects), 6.7.1 (Storage-class specifiers) and 6.9 (External definitions).