

## Definitions

- **Scope** : Scope of an identifier is the part of the program where the identifier may directly be accessible. In C, all identifiers are lexically (or statically) scoped.
- **Linkage** : Linkage describes how names can or can not refer to the same entity throughout the whole program or one single translation unit.  
The above sounds similar to Scope, but it is not so. To understand what the above means, let us dig deeper into the compilation process.
- **Translation Unit** : A translation unit is a file containing source code, header files and other dependencies. All of these sources are grouped together in a file for they are used to produce one single executable object. It is important to link the sources together in a meaningful way. For example, the compiler should know that `printf` definition lies in `stdio` header file.

In C and C++, a program that consists of multiple source code files is compiled *one at a time*. Until the compilation process, a variable can be described by its scope. It is only when the linking process starts, that linkage property comes into play. Thus, **scope is a property handled by compiler, whereas linkage is a property handled by linker.**

The Linker links the resources together in the *linking* stage of compilation process. The Linker is a program that takes multiple machine code files as input, and produces an executable object code. It resolves symbols (i.e, fetches definition of symbols such as "+" etc..) and arranges objects in address space.

Linkage is a property that describes how variables should be linked by the linker. Should a variable be available for another file to use? Should a variable be used only in the file declared? Both are decided by linkage.

Linkage thus allows you to couple names together on a per file basis, scope determines visibility of those names.

**There are 2 types of linkage:**

1. **Internal Linkage**: An identifier implementing internal linkage is not accessible outside the translation unit it is declared in. Any identifier within the unit can access an identifier having internal linkage. It is implemented by the keyword `static`. An internally linked identifier is stored in initialized or uninitialized segment of RAM. (**note**: `static` also has a meaning in reference to scope, but that is not discussed here).

Some Examples:

**Animals.cpp**

```

// C code to illustrate Internal Linkage
#include <stdio.h>

static int animals = 8;
const int i = 5;

int call_me(void)
{
    printf("%d %d", i, animals);
}

```

The above code implements static linkage on identifier `animals`. Consider `Feed.cpp` is located in the same translation unit.

#### Feed.cpp

```

// C code to illustrate Internal Linkage
#include <stdio.h>

int main()
{
    call_me();
    animals = 2;
    printf("%d", animals);
    return 0;
}

```

On compiling `Animals.cpp` first and then `Feed.cpp`, we get

Output : 5 8 2

Now, consider that `Feed.cpp` is located in a different translation unit. It will compile and run as above only if we use `#include "Animals.cpp"`.

Consider `Wash.cpp` located in a 3rd translation unit.

#### Wash.cpp

```

// C code to illustrate Internal Linkage
#include <stdio.h>
#include "animal.cpp" // note that animal is included.

int main()
{
    call_me();
    printf("\n having fun washing!");
    animals = 10;
    printf("%d\n", animals);
    return 0;
}

```

On compiling, we get:

```
Output : 5 8
having fun washing!
10
```


There are 3 translation units (Animals, Feed, Wash) which are using `animals` code. This leads us to conclude that each translation unit accesses its own copy of `animals`. That is why we have `animals = 8` for `Animals.cpp`, `animals = 2` for `Feed.cpp` and `animals = 10` for `Wash.cpp`. A file. This behavior eats up memory and decreases performance.

Another property of internal linkage is that it is **only implemented when the variable has global scope**, and all constants are by default internally linked.

**Usage :** As we know, an internally linked variable is passed by copy. Thus, if a header file has a function `fun1()` and the source code in which it is included in also has `fun1()` but with a different definition, then the 2 functions will not clash with each other. Thus, we commonly use internal linkage to hide translation-unit-local helper functions from the global scope. For example, we might include a header file that contains a method to read input from the user, in a file that may describe another method to read input from the user. Both of these functions are independent of each other when linked.

2. **External Linkage:** An identifier implementing external linkage is visible to **every translation unit**. Externally linked identifiers are *shared* between translation units and are considered to be located at the outermost level of the program. In practice, this means that you must define an identifier in a place which is visible to all, such that it has only one visible definition. It is the default linkage for globally scoped variables and functions. Thus, all instances of a particular identifier with external linkage refer to the same identifier in the program. The keyword `extern` implements external linkage. When we use the keyword `extern`, we tell the linker to look for the definition elsewhere. Thus, the declaration of an externally linked identifier does not take up any space. `Extern` identifiers are generally stored in initialized/uninitialized or text segment of RAM.

It is possible to use an `extern` variable in a local scope. This shall further outline the differences between linkage and scope. Consider the following code:



```

#include <stdio.h>

void foo()
{
    int a;
    extern int b; // line 1
}

void bar()
{
    int c;
    c = b; // error
}

int main()
{
    foo();
    bar();
}

```

Error: 'b' was not declared in this scope

**Explanation :** The variable `b` has local scope in the function `foo`, even though it is an `extern` variable. Note that compilation takes place before linking; i.e scope is a concept that can be used only during compile phase. After the program is compiled there is no such concept as “scope of variable”.

During compilation, scope of `b` is considered. It has local scope in `foo()`. When the compiler sees the `extern` declaration, it trusts that there is a definition of `b` somewhere and lets the linker handle the rest.

However, the same compiler will go through the `bar()` function and try to find variable `b`. Since `b` has been declared `extern`, it has not been given memory yet by the compiler; it does not exist yet. The compiler will let the linker find the definition of `b` in the translation unit, and then the linker will assign `b` the value specified in definition. It is only then that `b` will exist and be assigned memory. However, since there is no declaration given at compile time within the scope of `bar()`, or even in global scope, the compiler complains with the error above.

Given that it is the compiler’s job to make sure that all variables are used within their scopes, it complains when it sees `b` in `bar()`, when `b` has been declared in `foo()`’s scope. The compiler will stop compiling and the program will not be passed to the linker.

We can fix the program by declaring `b` as a global variable, by moving line 1 to before `foo`’s definition.

Let us look at another example

```
// C code to illustrate External Linkage
#include <stdio.h>

int x = 10;
int z = 5;

int main()
{
    extern int y; // line 2
    extern int z;
    printf("%d %d %d", x, y, z);
}

int y = 2;
```

Output: 10 2 5

We can explain the output by observing behaviour of external linkage. We define 2 variables *x* and *z* in *global* scope. By default, both of them have external linkage. Now, when we declare *y* as *extern*, we tell the compiler that there exists a *y* with some definition within the same translation unit. Note that this is during the compile time phase, where the compiler trusts the *extern* keyword and compiles the rest of the program. The next line, *extern int z* has no effect on *z*, as *z* is externally linked by default when we declared it as a global variable outside the program. When we encounter *printf* line, the compiler sees 3 variables, all 3 having been declared before, and all 3 being used within their scopes (in the *printf* function). The program thus compiles successfully, even though the compiler does not know the definition of *y*

The next phase is linking. The linker goes through the compiled code and finds *x* and *z* first. As they are global variables, they are externally linked by default. The linker then updates value of *x* and *z* throughout the entire translation unit as 10 and 5. If there are any references to *x* and *z* in any other file in the translation unit, they are set to 10 and 5

Now, the linker comes to *extern int y* and tries to find any definition of *y* within the translation unit. It looks through every file in the translation unit to find definition of *y*. If it does not find any definition, a linker error will be thrown. In our program, we have given the definition outside *main()*, which has already been compiled for us. Thus, the linker finds that definition and updates *y*.