Pointers and storage classes

Owner	A Angel David Sanchez
Tags	Embedded software

Pointers

- *A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.*

As the pointers in C store the memory addresses, their size is independent of the type of data they are pointing to. This size of pointers in C only depends on the system architecture.

Syntax of C Pointers

The syntax of pointers is similar to the variable declaration in C, but we use the **(*) dereferencing operator** in the pointer declaration.

```
datatype*ptr;
```

where

- **ptr** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

The above syntax is used to define a pointer to a variable. We can also define pointers to functions, structures, etc.

How to Use Pointers?

The use of pointers in C can be divided into three steps:

1. **Pointer Declaration**
2. **Pointer Initialization**
3. **Pointer Dereferencing**

1. Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the **(*) dereference operator** before its name.

Example

```
int *ptr;
```

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called wild pointers.

2. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the **(&) addressof operator** to get the memory address of a variable and then store it in the pointer variable.

Example

```
int var = 10;
int *ptr;
ptr =&var;
```

We can also declare and initialize the pointer in a single step. This method is called **pointer definition** as the pointer is declared and initialized at the same time.

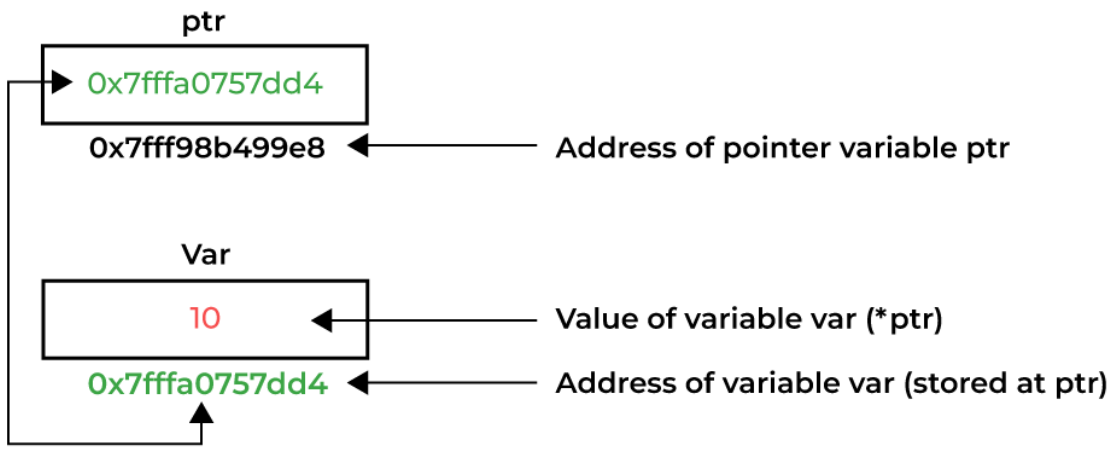
Example

```
int *ptr = &var;
```

> Note: It is recommended that the pointers should always be initialized to some value before starting using it. Otherwise, it may lead to number of errors.

Pointer Dereferencing

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same **(*) dereferencing operator** that we used in the pointer declaration.


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (stored at ptr)

Dereferencing a Pointer in C

C Pointer Example

- C

```
// C program to illustrate Pointers
#include <stdio.h>

void geeks()
{
  int var = 10;
  // declare pointer variable
  int * ptr;
  // note that data type of ptr and var must be same
  ptr = &var;
  // assign the address of a variable to a pointer
  printf ( "Value at ptr = %p \n" , ptr);
  printf ( "Value at var = %d \n" , var);
  printf ( "Value at *ptr = %d \n" , *ptr);
}
// Driver program
int main()
{
  geeks();
  return 0;
}
```

Output

```
Value at ptr = 0x7fff1038675c
Value at var = 10
Value at *ptr = 10
```

Types of Pointers in C

Pointers in C can be classified into many different types based on the parameter on which we are defining their types. If we consider the type of variable stored in the memory location pointed by the pointer, then the pointers can be classified into the following types:

1. Integer Pointers

As the name suggests, these are the pointers that point to the integer values.

Syntax

```
int *ptr;
```

These pointers are pronounced as **Pointer to Integer.**

Similarly, a pointer can point to any primitive data type. It can point also point to derived data types such as arrays and user-defined data types such as structures.

2. Array Pointer

Pointers and Array are closely related to each other. Even the array name is the pointer to its first element. They are also known as **Pointer to Arrays**. We can create a pointer to an array using the given syntax.

Syntax

```
char *ptr = &array_name;
```

Pointer to Arrays exhibits some interesting properties which we discussed later in this article.

3. Structure Pointer

The pointer pointing to the structure type is called **Structure Pointer** or Pointer to Structure. It can be declared in the same way as we declare the other primitive data types.

Syntax

```
structstruct_name *ptr;
```

In C, structure pointers are used in data structures such as linked lists, trees, etc.

4. Function Pointers

Function pointers point to the functions. They are different from the rest of the pointers in the sense that instead of pointing to the data, they point to the code. Let's consider a function prototype – **int func (int, char)**, the **function pointer** for this function will be

Syntax

```
int (*ptr)(int, char);
```

> Note: The syntax of the function pointers changes according to the function prototype.

5. Double Pointers

In C language, we can define a pointer that stores the memory address of another pointer. Such pointers are called double-pointers or **pointers-to-pointer**. Instead of pointing to a data value, they point to another pointer.

Syntax

```
datatype ** pointer_name;
```

Dereferencing Double Pointer

```
*pointer_name; // get the address stored in the inner level
pointer
**pointer_name; // get the value pointed by inner level poi
nter
```

> Note: In C, we can create multi-level pointers with any number of levels such as – ***ptr3, ****ptr4, ******ptr5 and so on.

6. NULL Pointer

The **Null Pointers** are those pointers that do not point to any memory location. They can be created by assigning a NULL value to the pointer. A pointer of any type can be assigned the NULL value.

Syntax

```
data_type *pointer_name = NULL;
        or
pointer_name = NULL
```

It is said to be good practice to assign NULL to the pointers currently not in use.

7. Void Pointer

The **Void pointers** in C are the pointers of type void. It means that they do not have any associated data type. They are also called **generic pointers** as they can point to any type and can be typecasted to any type.

Syntax

```
void *pointer_name;
```

One of the main properties of void pointers is that they cannot be dereferenced.

8. Wild Pointers

The **Wild Pointers** are pointers that have not been initialized with something yet. These types of C-pointers can cause problems in our programs and can eventually cause them to crash.

Example

```
int *ptr;
char *str;
```

9. Constant Pointers

In constant pointers, the memory address stored inside the pointer is constant and cannot be modified once it is defined. It will always point to the same memory address.

Syntax

```
data_type * constpointer_name;
```

10. Pointer to Constant

The pointers pointing to a constant value that cannot be modified are called pointers to a constant. Here we can only access the data pointed by the pointer, but cannot modify it. Although, we can change the address stored in the pointer to constant.

Syntax

```
constdata_type *pointer_name;
```

Other Types of Pointers in C:

There are also the following types of pointers available to use in C apart from those specified above:

- **Far pointer:** A far pointer is typically 32-bit that can access memory outside the current segment.
- **Dangling pointer:** A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer.
- **Huge pointer:** A huge pointer is 32-bit long containing segment address and offset address.
- **Complex pointer:** Pointers with multiple levels of indirection.
- **Near pointer:** Near pointer is used to store 16-bit addresses means within the current segment on a 16-bit machine.
- **Normalized pointer:** It is a 32-bit pointer, which has as much of its value in the segment register as possible.
- **File Pointer:** The pointer to a FILE data type is called a stream pointer or a file pointer.

Size of Pointers in C

The size of the pointers in C is equal for every pointer type. The size of the pointer does not depend on the type it is pointing to. It only depends on the operating system and CPU architecture. The size of pointers in C is

- **8 bytes** for a **64-bit System**
- **4 bytes** for a **32-bit System**

The reason for the same size is that the pointers store the memory addresses, no matter what type they are. As the space required to store the addresses of the different memory locations is the same, the memory required by one pointer type will be equal to the memory required by other pointer types.

How to find the size of pointers in C?

We can find the size of pointers using the **sizeof operator** as shown in the following program:

Example: C Program to find the size of different pointer types.

- C

```
// C Program to find the size of different pointers types
#include <stdio.h>
// dummy structure
struct str {
};
// dummy function
void func(int a, int b){};
int main()
{
 // dummy variables definitions
 int a = 10;
 char c = 'G';
 struct str x;
 // pointer definitions of different types
 int* ptr_int = &a;
 char* ptr_char = &c;
 struct str* ptr_str = &x;

 void (*ptr_func)( int , int ) = &func;

 void * ptr_vn = NULL;
 // printing sizes
 printf ( "Size of Integer Pointer  \t:\t%d bytes\n" ,
 sizeof (ptr_int));
 printf ( "Size of Character Pointer\t:\t%d bytes\n" ,
 sizeof (ptr_char));
 printf ( "Size of Structure Pointer\t:\t%d bytes\n" ,
 sizeof (ptr_str));
 printf ( "Size of Function Pointer\t:\t%d bytes\n" ,
 sizeof (ptr_func));
 printf ( "Size of NULL Void Pointer\t:\t%d bytes" ,
 sizeof (ptr_vn));

 return 0;
}
```

Output

```
 Size of Integer Pointer      :    8 bytes
Size of Character Pointer    :    8 bytes
Size of Structure Pointer    :    8 bytes
Size of Function Pointer    :    8 bytes
Size of NULL Void Pointer    :    8 bytes
```

As we can see, no matter what the type of pointer it is, the size of each and every pointer is the same.

Now, one may wonder that if the size of all the pointers is the same, then why do we need to declare the pointer type in the declaration? **The type declaration is needed in the pointer for dereferencing and pointer arithmetic purposes.**

C Pointer Arithmetic

The **Pointer Arithmetic** refers to the legal or valid arithmetic operations that can be performed on a pointer. It is slightly different from the ones that we generally use for mathematical calculations as only a limited set of operations can be performed on pointers. These operations include:

- Increment in a Pointer

- Decrement in a Pointer
- Addition of integer to a pointer
- Subtraction of integer to a pointer
- Subtracting two pointers of the same type
- Comparison of pointers of the same type.
- Assignment of pointers of the same type.
- C

```c
// C program to illustrate Pointer Arithmetic
#include <stdio.h>

int main()
{
 // Declare an array
 int v[3] = { 10, 100, 200 };
 // Declare pointer variable
 int * ptr;
 // Assign the address of v[0] to ptr
 ptr = v;
 for ( int i = 0; i < 3; i++) {
 // print value at address which is stored in ptr
 printf ( "Value of *ptr = %d\n" , *ptr);
 // print value of ptr
 printf ( "Value of ptr = %p\n\n" , ptr);
 // Increment pointer ptr by 1
 ptr++;
 }
 return 0;
}
```

Output

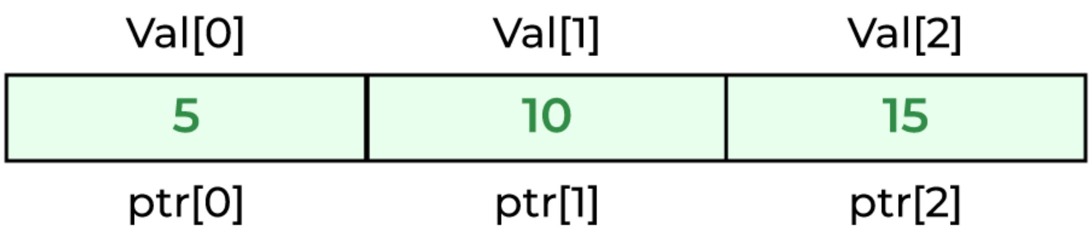
```
Value of *ptr = 10
Value of ptr = 0x7ffe8ba7ec50

Value of *ptr = 100
Value of ptr = 0x7ffe8ba7ec54
Value of *ptr = 200
Value of ptr = 0x7ffe8ba7ec58
```

C Pointers and Arrays

In C programming language, pointers and arrays are closely related. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

If we assign this value to a non-constant pointer of the same type, then we can access the elements of the array using this pointer.

Example 1: Accessing Array Elements using Pointer with Array Subscript



- C

```
// C Program to access array elements using pointer
#include <stdio.h>
void geeks()
{
 // Declare an array
 int val[3] = { 5, 10, 15 };
 // Declare pointer variable
 int * ptr;
 // Assign address of val[0] to ptr.
 // We can use ptr=&val[0];(both are same)
 ptr = val;
 printf ( "Elements of the array are: " );
 printf ( "%d, %d, %d" , ptr[0], ptr[1], ptr[2]);
```

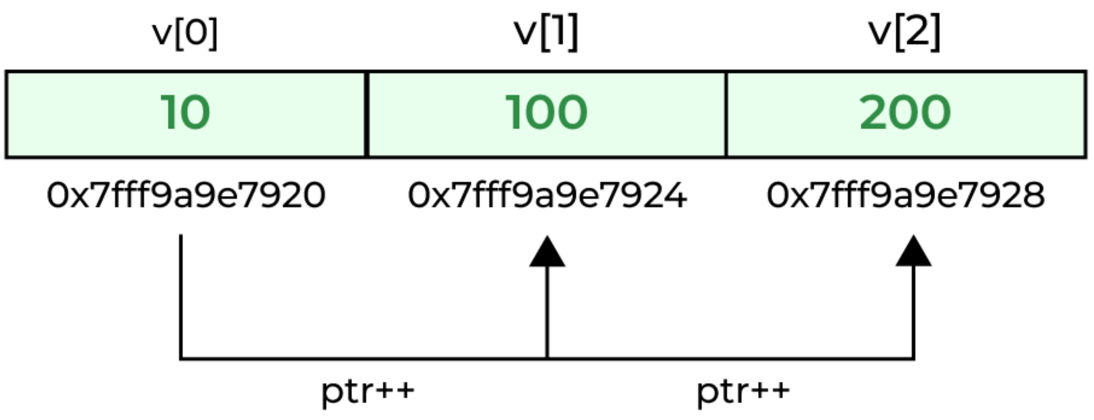
```
  return ;
}
// Driver program
int main()
{
  geeks();
  return 0;
}
```

Output

```
Elements of the array are: 5 10 15
```

Not only that, as the array elements are stored continuously, we can pointer arithmetic operations such as increment, decrement, addition, and subtraction of integers on pointer to move between array elements.

Example 2: Accessing Array Elements using Pointer Arithmetic



- C

```
// C Program to access array elements using pointers
#include <stdio.h>
int main()
{
  // defining array
  int arr[5] = { 1, 2, 3, 4, 5 };
  // defining the pointer to array
  int * ptr_arr = arr;
  // traversing array using pointer arithmetic
  for ( int i = 0; i < 5; i++) {
  printf ( "%d " , *ptr_arr++);
  }
  return 0;
}
```

Output

```
1 2 3 4 5
```

This concept is not limited to the one-dimensional array, we can refer to a multidimensional array element as well using pointers.

To know more about pointers to an array, refer to this article – **Pointer to an Array**

Uses of Pointers in C

The C pointer is a very powerful tool that is widely used in C programming to perform various useful operations. It is used to achieve the following functionalities in C:

1. Pass Arguments by Reference
2. Accessing Array Elements
3. **Return Multiple Values from Function**
4. **Dynamic Memory Allocation**
5. **Implementing Data Structures**
6. In System-Level Programming where memory addresses are useful.
7. In locating the exact value at some memory location.
8. To avoid compiler confusion for the same variable name.
9. To use in Control Tables.

Advantages of Pointers

Following are the major advantages of pointers in C:

- Pointers are used for dynamic memory allocation and deallocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

Disadvantages of Pointers

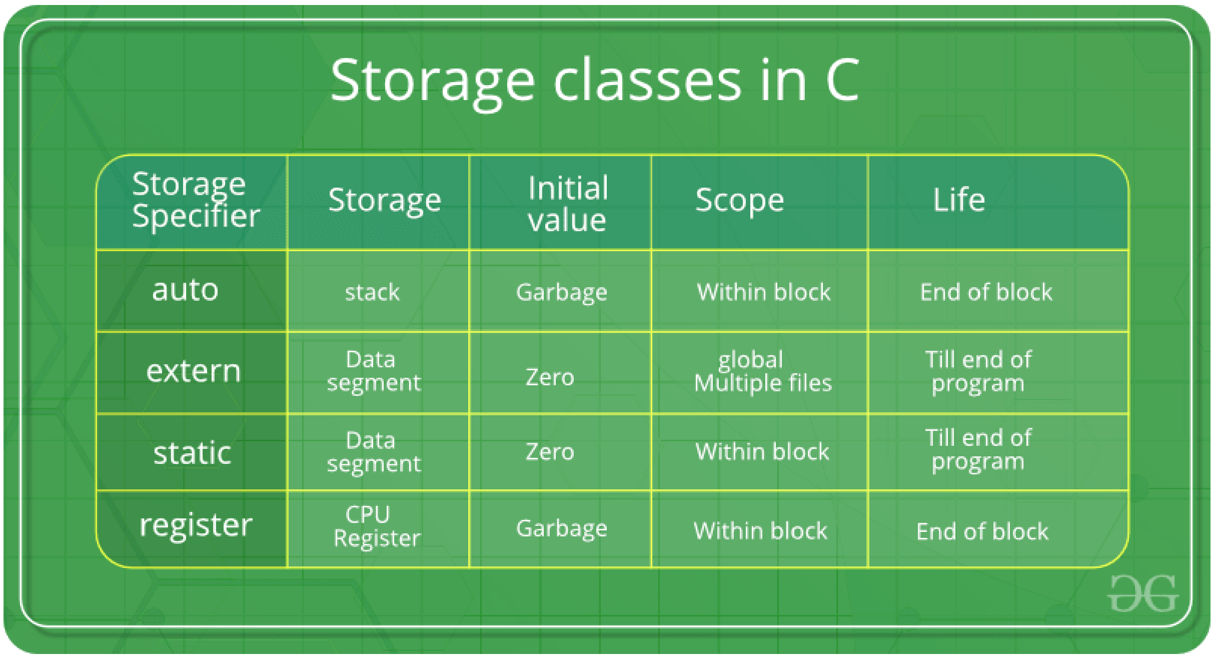
Pointers are vulnerable to errors and have following disadvantages:

- Memory corruption can occur if an incorrect value is provided to pointers.
- Pointers are a little bit complex to understand.
- Pointers are majorly responsible for **memory leaks in C**.
- Pointers are comparatively slower than variables in C.
- Uninitialized pointers might cause a segmentation fault.

Storage Classes

C Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility, and lifetime which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:



Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

1. auto

This is the default storage class for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs in C language. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared.

However, they can be accessed outside their scope as well using the concept of pointers given here by pointing to the very exact memory location where the variables reside. They are assigned a garbage value by default whenever they are declared.

2. extern

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block.

Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead, we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

3. static

This storage class is used to declare static variables which are popularly used while writing programs in C language. Static variables have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist till the termination of the program. Thus, no new memory is allocated because they are not re-declared.

Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

4. register

This storage class declares register variables that have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.

If a free registration is not available, these are then stored in the memory only. Usually, a few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

Syntax

To specify the storage class for a variable, the following syntax is to be followed:

```
storage_class var_data_type var_name;
```

Extern keyword

Extern keyword in C 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, let's clarify it

- **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 the 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 the definition as a superset of the declaration (or declaration as a subset of the definition).
- Extern is a short name for external.
- The extern variable is used when a particular files need to access a variable from another file.

Syntax of extern in C

The syntax to define an extern variable in C is just to use the extern keyword before the variable declaration.

```
externdata_type variable_name;
```

Example of extern Variable in C

- C

```
#include <stdio.h>

extern int a;     // int var;  ->  declaration and definition

 // extern int var;  -> declaration

int main()

{
```

```
printf ( "%d" , a);
return 0;
}
```

Properties of extern Variable in C

- When we write extern some_data_type some_variable_name; no memory is allocated. Only the property of the variable is announced.
- Multiple declarations of extern variable is allowed within the file. This is not the case with automatic variables.
- The extern variable says to the compiler “Go outside my scope and you will find the definition of the variable that I declared.”
- The compiler believes that whatever that extern variable said is true and produces no error. Linker throws an error when it finds no such variable exists.
- When an extern variable is initialized, then memory for this is allocated and it will be considered defined.

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.

Now, how would you *define* var? You would do this:

```
int var = 10;
```

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

```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

```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

```c
// As we are importing the file and henceforth the
// defination
#include "somefile.h"
// Declaring the same variable
extern int var;
  // int var;
  // It will throw compiler error as compiler will get
  // confused where the variable is defined
int main( void )
```

```
{
  var = 10;
  return 0;
}
// Now it will compile and run successfully
```

Output:

```
10
```

> Note: Here arises another scenario what if we do not declare with extern in the above code snippet,?

Considering an assumption that somefile.h contains the definition of var, this program will compile successfully. ‘extern’ keyword is used for a variable when we declare the variable in one file and define it in another file. But here when we import the same file in the file where it is declared, here compiler error will be generated.

It is because we still have to use the extern keyword in the file where we have declared that variable so as to tell our compiler that this variable is been defined somewhere else than only new memory space will not be allocated else it will create another memory block which usage of ‘extern’ keyword useless.

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 defined 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 the 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.

Static variables

Static variables have the property of preserving their value even after they are out of their scope! Hence, a static variable preserves its previous value in its previous scope and is not initialized again in the new scope.

Syntax:

```
static data_typevar_name = var_value;
```

Following are some interesting facts about static variables in C:

1) A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.

For example, we can use static int to count the number of times a function is called, but an auto variable can't be used for this purpose.

Example

- C

```
// C Program to illustrate the static variable lifetime
#include <stdio.h>
// function with static variable

int fun()
{
  static int count = 0;
  count++;

  return count;
}
int main()
{
  printf ( "%d " , fun());
  printf ( "%d " , fun());

  return 0;
}
```

Output

```
1 2
```

The above program prints 1 2 because static variables are only initialized once and live till the end of the program. That is why they can retain their value between multiple function calls.

Let’s try the same code for the local auto variable.

Example

- C

```
// C Program to illustrate local auto variable in comparison
// of static variable.
#include <stdio.h>
// Function
int fun()
{
  int count = 0;
  count++;

  return count;
}
// Driver Code
int main()
{
  printf ( "%d " , fun());
  printf ( "%d " , fun());
```

```
    return 0;
}
```

Output

```
1 1
```

2) Static variables are allocated memory in the data segment, not the stack segment. See the **memory layout of C programs** for details.

3) Static variables (like global variables) are initialized as 0 if not initialized explicitly. For example in the below program, the value of x is printed as 0, while the value of y is something garbage. See **this** for more details.

Example

- C

```
// C program to illustrate the default value of static
// variables
#include <stdio.h>

int main()
{
    static int x;
    int y;
    printf("%d \n%d", x, y);
}
```

Output

```
0
[some_garbage_value]
```

4) In C, static variables can only be initialized using constant literals. For example, the following program fails in the compilation. See **this** for more details.

Example

- C

```
#include<stdio.h>

int initializer(void)
```

```
{
 return 50;
}
int main()
{
 static int i = initializer();
 printf ( " value of i = %d" , i);
 getchar ();
 return 0;
}
```

Output

```
In function 'main':
9:5: error: initializer element is not constant
     static int i = initializer();
     ^
```

> Note: Please note that this condition doesn't hold in C++. So if you save the program as a C++ program, it would compile and run fine.

5) Static global variables and functions are also possible in C/C++. The purpose of these is to limit the scope of a variable or function to a file. Please refer to **Static functions in C** for more details.

6) Static variables should not be declared inside a structure. The reason is C compiler requires the entire structure elements to be placed together (i.e.) memory allocation for structure members should be contiguous. It is possible to declare structure inside the function (stack segment) or allocate memory dynamically(heap segment) or it can be even global (BSS or data segment). Whatever might be the case, all structure members should reside in the same memory segment because the value for the structure element is fetched by counting the offset of the element from the beginning address of the structure. Separating out one member alone to a data segment defeats the purpose of structure and it is possible to have an entire structure as static.