# Compiling a C program

Then compile it using below command.

$ gcc –Wall filename.c –o filename

The option -Wall enables all compiler’s warning messages. This option is recommended to generate better code.

## What goes inside the compilation process?

Compiler converts a C program into an executable. There are four phases for a C program to become an executable:

1. Pre-processing
2. Compilation
3. Assembly
4. Linking

By executing below command, We get the all intermediate files in the current directory along with the executable.

$gcc –Wall –save-temps filename.c –o filename

## Pre-processing

This is the first phase through which source code is passed. This phase include:

* Removal of Comments
* Expansion of Macros
* Expansion of the included files.

The preprocessed output is stored in the filename.i. Let’s see what’s inside filename.i: using $vi filename.i

**$gcc -E filename.c**

In the above output, source file is filled with lots and lots of info, but at the end our code is preserved.  
Analysis:

* printf contains now a + b rather than add(a, b) that’s because macros have expanded.
* Comments are stripped off.
* #include<stdio.h> is missing instead we see lots of code. So header files has been expanded and included in our source file.

## Compiling

**$gcc -S filename.c**

The next step is to compile filename.i and produce an; intermediate compiled output file filename.s. This file is in assembly level instructions. Let’s see through this file using $vi filename.s

## Assembly

In this phase the filename.s is taken as input and turned into filename.o by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved. Let’s view this file using $vi filename.o

$gcc -C filename.c

## Linking

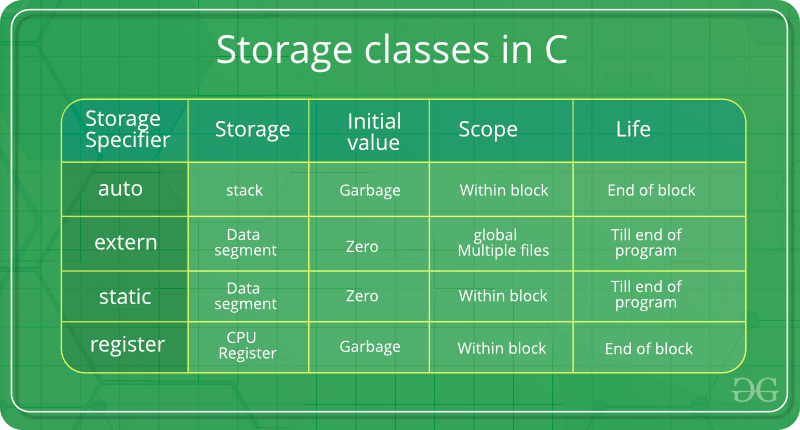
This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments. This task can be easily verified by using $size filename.o and $size filename. Through these commands, we know that how output file increases from an object file to an executable file. This is because of the extra code that linker adds with our program.

**$gcc filename.c -o exec\_filename**

# **Storage Classes in C**

# Storage Classes are used to describe about the features of a variable/function. These feature basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:



**Auto Storage Class**

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 resides. They are assigned a garbage value by default whenever they are declared.

The **auto** storage class is the default storage class for all local variables.

{

int mount;

auto int month;

}

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

**Extern Storage Class**

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

**First File: main.c**

#include <stdio.h>

int count ;

extern void write\_extern();

main() {

count = 5;

write\_extern();

}

**Second File: support.c**

#include <stdio.h>

extern int count;

void write\_extern(void) {

printf("count is %d\n", count);

}

Here, *extern* is being used to declare *count* in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows −

$gcc main.c support.c

It will produce the executable program **a.out**. When this program is executed, it produces the following result −

count is 5

## The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. .Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

#include <stdio.h>

/\* function declaration \*/

void func(void);

static int count = 5; /\* global variable \*/

main() {

while(count--) {

func();

}

return 0;

}

/\* function definition \*/

void func( void ) {

static int i = 5; /\* local static variable \*/

i++;

printf("i is %d and count is %d\n", i, count);

}

When the above code is compiled and executed, it produces the following result −

i is 6 and count is 4

i is 7 and count is 3

i is 8 and count is 2

i is 9 and count is 1

i is 10 and count is 0

## The register Storage Class

This storage class declares register variables which 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 register is not available, these are then stored in the memory only. Usually 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.

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location)

{

register int miles;

}

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

# Structure Padding

* In order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.
* Architecture of a computer processor is such a way that it can read 1 word (4 byte in 32 bit processor) from memory at a time.
* To make use of this advantage of processor, data are always aligned as 4 bytes package which leads to insert empty addresses between other member’s address.
* Because of this structure padding concept in C, size of the structure is always not same as what we think.

For example, please consider below structure that has 5 members.struct student

{

int id1;

int id2;

char a;

char b;

float percentage;

};..

As per C concepts, int and float datatypes occupy 4 bytes each and char datatype occupies 1 byte for 32 bit processor. So, only 14 bytes (4+4+1+1+4) should be allocated for above structure.

* But, this is wrong. Do you know why?
* Architecture of a computer processor is such a way that it can read 1 word from memory at a time.
* 1 word is equal to 4 bytes for 32 bit processor and 8 bytes for 64 bit processor. So, 32 bit processor always reads 4 bytes at a time and 64 bit processor always reads 8 bytes at a time.
* This concept is very useful to increase the processor speed.
* To make use of this advantage, memory is arranged as a group of 4 bytes in 32 bit processor and 8 bytes in 64 bit processor.
* So, to avoid structure padding we can use pragma pack as well as an attribute.

#**pragma pack** instructs the compiler to **pack** structure members with particular alignment. Most compilers, when you declare a struct, will insert padding between members to ensure that they are aligned to appropriate addresses in memory (usually a multiple of the type's size).

|  |
| --- |
| // C program to show an example  // of Structure padding  #include <stdio.h>    struct s {      int i;      char ch;      double d;  };    int main()  {      struct s A;      printf("Size of A is: %ld", sizeof(A));  } |

**Output:**

Size of A is: 16

|  |
| --- |
| // C program to avoid structure  // padding using pragma pack  #include <stdio.h>    // To force compiler to use 1 byte packaging  #pragma pack(1)  struct s {      int i;      char ch;      double d;  };    int main()  {      struct s A;      printf("Size of A is: %ld", sizeof(A));  } |

**Output:**

Size of A is: 13

**Difference Between Structure And Union**

**Structure and union** both are user defined data types which contains variables of **different** data types. ... In **union**, the total memory space allocated is equal to the member with largest size. All other members share the same memory space. This is the biggest **difference between structure and union**.

Difference between Structure and Union

Structure Union

In structure each member get separate space in memory. Take below example.

struct student { int rollno; char gender; float marks; }s1;

The total memory required to store a structure variable is equal to the sum of size of all the members. In above case 7 bytes (2+1+4) will be required to store structure variable s1.

In union, the total memory space allocated is equal to the member with largest size. All other members share the same memory space. This is the biggest difference between structure and union.

union student { int rollno; char gender; float marks; }s1;

In above example variable marks is of float type and have largest size (4 bytes). So the total memory required to store union variable s1 is 4 bytes.

We can access any member in any sequence.

s1.rollno = 20; s1.marks = 90.0; printf("%d",s1.rollno);

The above code will work fine but will show erroneous output in the case of union. We can access only that variable whose value is recently stored.

s1.rollno = 20; s1.marks = 90.0; printf("%d",s1.rollno);

The above code will show erroneous output. The value of rollno is lost as most recently we have stored value in marks. This is because all the members share same memory space.

All the members can be initialized while declaring the variable of structure. Only first member can be initialized while declaring the variable of union. In above example we can initialize only variable rollno at the time of declaration of variable.

