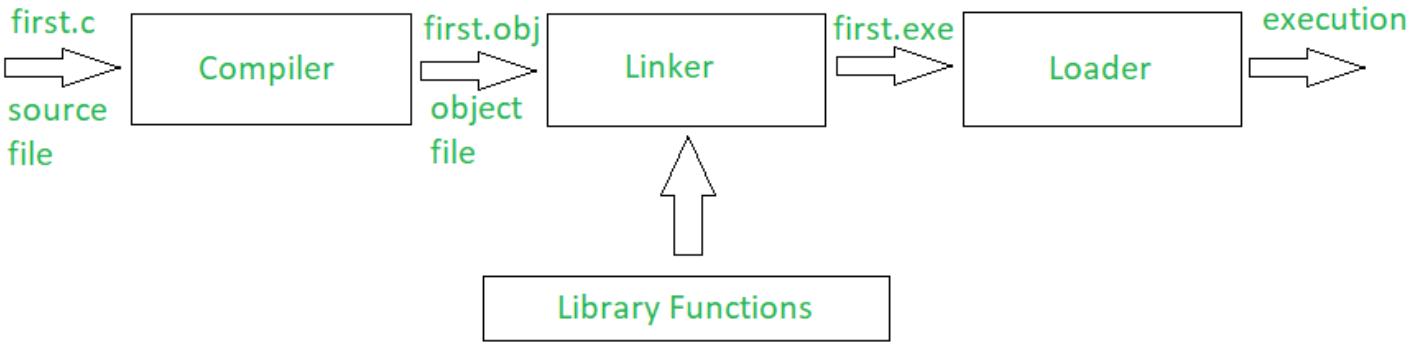
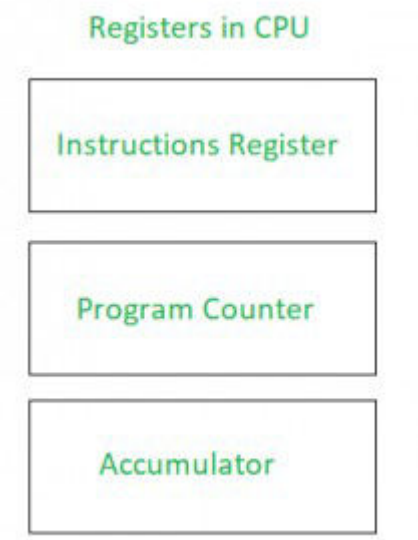
How does a C program executes?



- Whenever a C program file is compiled and executed, the compiler generates some files with the same name as that of the C program file but with different extensions.
- Every file that contains a C program must be saved with '.c' extension. This is necessary for the compiler to understand that this is a C program file. Suppose a program file is named, first.c. The file first.c is called the source file which keeps the code of the program. Now, when we compile the file, the C compiler looks for errors. If the C compiler reports no error, then it stores the file as a .obj file of the same name, called the object file. So, here it will create the first.obj. This .obj file is not executable. The process is continued by the Linker which finally gives a .exe file which is executable.
- **Linker**: First of all, let us know that library functions are not a part of any C program but of the C software. Thus, the compiler doesn't know the operation of any function, whether it be printf or scanf. The definitions of these functions are stored in their respective library which the compiler should be able to link. This is what the Linker does. So, when we write

#include, it includes stdio.h library which gives access to Standard Input and Output. The linker links the object files to the library functions and the program becomes a .exe file. Here, first.exe will be created which is in an executable format.

- **Loader**: Whenever we give the command to execute a particular program, the loader comes into work. The loader will load the .exe file in RAM and inform the CPU with the starting point of the address where this program is loaded



- **Instruction Register**: It holds the current instructions to be executed by the CPU.
  Program Counter: It contains the address of the next instructions to be executed by the CPU.
  Accumulator: It stores the information related to calculations.
- The loader informs Program Counter about the first instruction and initiates the execution. Then onwards, Program Counter handles the task.
- **LOADER**
- Loader loads the executable module to the main memory for execution.
- Loader takes executable module generated by a linker as input.
- Loader allocates the addresses to an executable module in main memory for execution

- The types of Loader are Absolute loading, Relocatable loading and Dynamic Run-time loading.
- Loader is the program of the operating system which loads the executable from the disk into the primary memory(RAM) for execution. It allocates the memory space to the executable module in main memory and then transfers control to the beginning instruction of the program .
- What really happens while running the executable: One could also use strace command for the same.

The first call which is displayed is ‘**execve()**‘ which actually is the loader . This loader creates the process which involves:

- Reading the file and creating an address space for the process.
- Page table entries for the instructions, data and program stack are created and the register set is initialized.
- Then, Executes a jump instruction to the first instruction of the program which generally causes a page fault and the first page of your instructions is brought into memory.

When do we pass arguments by reference or pointer?

- To modify local variables of the caller function
- For passing large sized arguments

Redeclaration of global variable in C

```
int x;

int x = 5;

int main()

{

printf("%d", x);

}
```

C allows a global variable to be declared again when first declaration doesn’t initialize the variable.

What is the size_t data type in C?

It is guaranteed to be big enough to contain the size of the biggest object the host system can handle. Basically the maximum permissible size is dependent on the compiler; if the compiler is 32 bit then it is simply a typedef(i.e., alias) for unsigned int but if the compiler is 64 bit then it would be a typedef for unsigned long long. The size_t data type is never negative.

Therefore many C library functions like *malloc, memcpy and strlen* declare their arguments and return type as size_t.

- If no data type is given to a variable, then the compiler automatically converted it to int data type
- We can't use any modifiers in float data type. If programmer try to use it then compiler automatically give compile time error.
- signed float a;

Interesting Facts about Macros and Preprocessors in C

In a C program, all lines that start with # are processed by preprocessor which is a special program invoked by the compiler. In a very basic term, preprocessor takes a C program and produces another C program without any #.

Following are some interesting facts about preprocessors in C.
1) When we use *include* directive, the contents of included header file (after preprocessing) are copied to the current file.

Angular brackets < and > instruct the preprocessor to look in the standard folder where all header files are held. Double quotes " and " instruct the preprocessor to look into the current folder (current directory).

2) When we use *define* for a constant, the preprocessor produces a C program where the defined constant is searched and matching tokens are replaced with the given expression. For example in the following program *max* is defined as 100.

How to concatenate two integer arrays without using loop in C ?

```
// arr1[] is of size m+n and arr2[] is of size n. This function
// appends contents of arr2[] at the end of arr1[]
void concatenate(int arr1[], int arr2[], int m, int n)
{
  memcpy(arr1 + m, arr2, sizeof(arr2));
}
```

Const Qualifier in C

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed (Which depends upon where const variables are stored, we may change value of const variable by using pointer). The result is implementation-defined if an attempt is made to change a const.

Const are stored in text section (it's read only and contain all executable instruction).

We can change the value of constant with the help of pointer. Constant are fast compare to variable and processor generates more optimize code with the constant.

When to use Const?

- In call by reference function argument, if you do not want to change the actual value which has passed in function (e.g., int Display (const char *pcMessage)) .
- In the case of I/O memory mapped register , fields of structure and union should be constant.

Pointer to constant

const int *ptr;

int const *ptr;

We can change pointer to point to any other integer variable, but cannot change value of object (entity) pointed using pointer ptr. Pointer is stored in read-write area (stack in present case). Object pointed may be in read only or read write area

This is a pointer to a constant character. You cannot change the value pointed by ptr, but you can change the pointer itself. “const char *” is a (non-const) pointer to a const char.

Constant pointer to variable

int *const ptr;

Above declaration is constant pointer to integer variable, means we can change value of object pointed by pointer, but cannot change the pointer to point another variable.

This is a constant pointer to non-constant character. You cannot change the pointer p, but can change the value pointed by ptr

constant pointer to constant

const int *const ptr;

Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable

This is a constant pointer to constant character. You can neither change the value pointed by ptr nor the pointer ptr.

- int const* is pointer to const int
- int *const is const pointer to int
- int const* const is const pointer to const int

Diffference between #define and const in C?

#define is a preprocessor directive. Things defined by #define are replaced by the preprocessor before compilation begins.

const variables are actual variables like other normal variable.

The big advantage of const over #define is type checking. We can also have poitners to const varaibles, we can pass them around, typecast them and any other thing that can be done with a normal variable. One disadvantage that one could think of is extra space for variable which is immaterial due to optimizations done by compilers.

What is the difference between 'function pointer' and 'pointer to a function'?

What is the difference between passing pointer to a function and a function pointer?

The answer to this question would be that passing pointer to a function is just passing the address of some variable which is stored in a pointer variable of that specific data type by value to a function.

While a function pointer is a variable that stores address to a function’s executable code in the memory.

How do we create a pointer to an integer and character in c??

int * iptr; char * cptr;

This is nothing but the pointer to an integer and pointer to a character respectively.

Now,

how we declare a function?

int myFunction(int);

Here, myFunction() is a function that returns int and take one argument of type int.

Now, For function pointer,

int * myFunction(int); //Here myFunction() returns integer pointer.

This is known as Function pointer.

Now, Pointer to a Function,

Let us take one example,

```
#include<stdio.h>
//A normal function having int parameter.
void myFunction(int num)
{
printf(“Number is : %d”,num);
}
int main()
{
//Here, fun_ptr is a pointer to a myFunction()
void (*fun_ptr)(int);

fun_ptr=&myFunction; //holding address of myFunction().
//invoke myFunction() using fun_ptr.
(*fun_ptr)(10);
return 0;
}
```

OutPut : 10

This is known as pointer to function.

A function to pointer is one which returns a pointer.

Ex: int *add(int num1,int num2) {

.

.
}

A pointer to function points to a function.

Ex: int add();
int (*addi)();
addi =&add;
then (*addi)() represents add();

A pointer to a function is a pointer that points to a function. A function pointer is a pointer that either has an indeterminate value, or has a null pointer value, or points to a function.

Following are some interesting facts about function pointers

- Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
- Unlike normal pointers, we do not allocate de-allocate memory using function pointers.
- A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing *, the program still works.
- Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.
- Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks

Little endian to Big endian conversion and its Importance

sometimes endianness became crucial issues in your program when you transmit data serially over the network from one computer to another computer.

So endianness is mainly important when you are reading and writing the data across the network from one system to another. If the sender and receiver computer have the different endianness, then receiver system would not receive the actual data which transmitted by the sender.

How is the data stored in memory it depends on the endianness of processor?

Endianness

The endianness is the bytes order in which data stored in the memory of the computer and it also describes the order of byte transmission over a digital link. In memory data store in which order it depends on the endianness of the system, if the system is big-endian then the MSB byte store first (means at lower address) and if the system is the little endian then LSB byte store first (means at lower address).

It matters in network programming: Suppose you write integers to file on a little endian machine and you transfer this file to a big endian machine. Unless there is little endian to big endian transformation, big endian machine will read the file in reverse order. You can find such a practical example here.

What are bi-endians?

Bi-endian processors can run in both modes little and big endian.

Volatile

A volatile keyword is a qualifier which prevents the objects, from the compiler optimization and tells the compiler that the value of the object can be change at any time without any action being taken by the code. It prevents from the cache a variable into a register and ensures that on every access variable is fetched from the memory.

The volatile keyword is mainly used where we directly deal with GPIO, interrupt or flag Register. It is also used where a global variable or buffer is shared between the threads.

Declaration of volatile in c

```
int volatile iData;
volatile int iData;
```

We can also use the volatile keyword with pointers same like as variables.

```
int volatile *piData;
volatile int *piData;
```

```
volatile int iValue;
volatile int* piData = & iValue;
```

When need to use volatile keyword

Program works fine but when increase the optimization level of compiler its behavior change and not work as per the desire.

volatile in C actually came into existence for the purpose of not caching the values of the variable automatically. It will tell the machine not to cache the value of this variable. So it will take the value of the given volatile variable from the main memory every time it encounters it. This mechanism is used because at any time the value can be modified by the OS or any interrupt. So using volatile will help us accessing the value afresh every time.

Volatile tells the compiler not to optimize anything that has to do with the volatile variable.

Volatile is qualifier for a variable as well as pointer.

if we apply volatile qualifier to any variable then we instruct the compiler that don't apply **optimization rule on the variable.** every time when you are going to access the variable, it should be accessed from the memory directly.

what is optimization ?

Optimization generally used to speed up the processing of operations.

if we need to access some variable from the memory then compiler 1st time access it from memory and store it in temporary register.

again if he want to access the same variable then it will access the variable form temporary register instead of the accessing from the memory. in this way compiler do optimization as to reduce the time and executive the operation fast.

where volatile can be use full ?

1. multi threaded application
2. Global variables
3. variable used in ISR.

```
unsigned char FLAG_REG;   // Hardware flag register


void func (void)
{
while (FLAG_REG & 0x01)   // Repeat while bit 0 is set
  {
    //Perform any operation
  }
}
```

When we increase the compiler optimization level for a program then for the better performance compiler load the FLAG_REG value in a register and does not re-read again although the value of FLAG_REG change by the hardware. In that situation, your code would not be work as per your expectation.

when you qualify the FLAG_REG from the volatile keyword then compiler understand that may be the value of FLAG_REG change by the outer word so it avoids implementing any optimization on it.

Can a variable be both const and volatile?

Yes. Const qualifier makes sure the variable declared isn't changed by the code, say assignment or something similar. This doesn't restrict the variable from being modified external to the code. Take a peripheral's status register as an example. If we don't want the user to modify the contents of the register (Read-only mode), it is made const. Now if there is a chance for this register to be updated outside the code, the variable shouldn't be cached. Make it volatile.

Volatile keyword tells the compiler that the value of the variable can change (may be frequently) so that it can avoid the optimization.

Constant means variable can not be assigned a new value the catch here is using the pointer we can change the value. So when we say a variable is both a constant and volatile that means we are instructing the compiler that the variable can not be assigned a new value and its value can be changed by the use of pointer so don't do any optimization.

Example

1) Global variables modified by an interrupt service routine outside the scope: For example, a global variable can represent a data port (usually global pointer referred as memory mapped IO) which will be updated dynamically. The code reading data port must be declared as volatile in order to fetch latest data available at the port. Failing to declare variable as volatile, the compiler will

optimize the code in such a way that it will read the port only once and keeps using the same value in a temporary register to speed up the program (speed optimization). In general, an ISR used to update these data port when there is an interrupt due to availability of new data

Initialization of global and static variables in C

Is not possible

gets() is risky to use!

The code looks simple, it reads string from standard input and prints the entered string, but it suffers from Buffer Overflow as gets() doesn't do any array bound testing. gets() keeps on reading until it sees a newline character.

To avoid Buffer Overflow, fgets() should be used instead of gets() as fgets() makes sure that not more than MAX_LIMIT characters are read.

Difference between while(1) and while(0) in C language

While(1)

It is an infinite loop which will run till a break statement is issued explicitly. Interestingly not while(1) but any integer which is non-zero will give the similar effect as while(1). Therefore, while(1), while(2) or while(-255), all will give infinite loop only.

A simple usage of while(1) can be in the Client-Server program. In the program, the server runs in an infinite while loop to receive the packets sent form the clients.

But practically, it is not advisable to use while(1) in realworld because it increases the CPU usage and also blocks the code i.e one cannot come out from the while(1) until the program is closed manually.

while(1) can be used at a place where condition needs to be true always.

while(0)

It is opposite of while(1). It means condition will always be false and thus code in while will never get executed.

Why strcpy and strncpy are not safe to use?

The strcpy() function is used to copy the source string to destination string. If the buffer size of dest string is more then src string, then copy the src string to dest string with terminating NULL character. But if dest buffer is less, then src then it will copy the content without terminating NULL character. The strings may not overlap, and the destination string must be large enough to receive the copy.

Problem with strcpy(): The strcpy() function does not specify the size of the destination array, so buffer overrun is often a risk. Using strcpy() function to copy a large character array into smaller one is dangerous, but if the string will fit, then it will not worth the risk. If destination string is not large enough to store the source string then the behavior of strcpy() is unspecified or undefined.

strncpy() function

The strncpy() function is similar to strcpy() function, except that at most n bytes of src are copied. If there is no NULL character among the first n character of src, the string placed in dest will not be NULL-terminated. If the length of src is less than n, strncpy() writes additional NULL character to dest to ensure that a total of n character are written.

Problem with strncpy(): If there is no null character among the first n character of src, the string placed in dest will not be null-terminated. So strncpy() does not guarantee that the destination string will be NULL terminated. The strlen() non-terminated string can cause segfault. In other words, non-terminated string in C/C++ is a time-bomb just waiting to destroy code.

Now, the next question is, any function which guarantees that destination string will be NULL terminated and no chance of buffer overrun?

So, the answer of above question is “YES”, there are several function in “stdio.h” library which guarantee the above condition will be satisfied.

- **snprintf**
- **strlcpy**

Both functions guarantee that the destination string will be NULL terminated.Similarly, snprintf() function, strlcpy function copied at most dest_size-1 characters (dest_size is the size of the destination string buffer) from src to dst, truncating src if necessary. The result is always null-terminated. The function returns strlen(src). Buffer overflow can be checked as follows:

```
if (strlcpy(dst, src, dstsize) >= dest_size)
    return -1;
```

Ranking the functions according to level of sanity:

strcpy < strncpy < snprintf < strlcpy

What is the difference between “char a” and “char a[1]”?

1) “char a” represents a character variable and “char a[1]” represents a char array of size 1.

2) If we print value of char a, we get ASCII value of the character (if %d is used). And if we print value of char a[1], we get address of the only element in array.

How ++a is different from a++ as lvalue?

It is because ++a returns an *lvalue*, which is basically a reference to the variable to which we can further assign — just like an ordinary variable. It could also be assigned to a reference as follows:

```
int &ref = ++a; // valid
int &ref = a++; // invalid
```

Whereas if you recall how a++ works, it doesn’t immediately increment the value it holds. For brevity, you can think of it as getting incremented in the next statement. So what basically happens is that a++ returns an *rvalue*, which is

basically just a value like the value of an expression which is not stored. You can think of a++ = 20; as follows after being processed:

```
int a = 10;

// On compilation, a++ is replaced by the value of a which is an rvalue:
10 = 20; // Invalid

// Value of a is incremented
a = a + 1;
```

typedef versus #define in C

- typedef is limited to giving symbolic names to types only, where as #define can be used to define alias for values as well, e.g., you can define 1 as ONE, 3.14 as PI, etc.
- typedef interpretation is performed by the compiler where #define statements are performed by preprocessor.
- #define should not be terminated with semicolon, but typedef should be terminated with semicolon.
- #define will just copy-paste the definition values at the point of use, while typedef is actual definition of a new type.
- typedef follows the scope rule which means if a new type is defined in a scope (inside a function), then the new type name will only be visible till the scope is there. In case of #define, when preprocessor encounters #define, it replaces all the occurrences, after that (No scope rule is followed).

#ifdef: This directive is the simplest conditional directive. This block is called a conditional group. The controlled text will get included in the preprocessor output iff the macroname is defined. The controlled text inside a conditional will embrace preprocessing directives. They are executed only if the conditional succeeds. You can nest these in multiple layers, but they must be completely nested. In other words, ‘#endif’ always matches the nearest ‘#ifdef’ (or

‘#ifndef’, or ‘#if’). Also, you can’t begin a conditional group in one file and finish it in another.

Syntax:

```
#ifdef MACRO
   controlled text
#endif /* macroname */
```

- **#ifndef**: We know that the in #ifdef directive if the macroname is defined, then the block of statements following the #ifdef directive will execute normally but if it is not defined, the compiler will simply skip this block of statements. The #ifndef directive is simply opposite to that of the #ifdef directive. In case of #ifndef , the block of statements between #ifndef and #endif will be executed only if the macro or the identifier with #ifndef is not defined.
  Syntax :

```
ifndef macro_name
   statement1;
   statement2;
   statement3;
   .
   .
   .
   statementN;
endif
```

If the macro with name as ‘macroname‘ is not defined using the #define directive then only the block of statements will execute.

- **#if, #else and #elif**: These directives works together and control compilation of portions of the program using some conditions. If the condition with the #if directive evaluates to a non zero value, then the group of line immediately after the #if directive will be executed otherwise if the condition with the #elif directive evaluates to a non zero value, then the group of line immediately after the #elif directive will be executed else the lines after #else directive will be executed.
  Syntax:

```
#if macro_condition
```

```
   statements
#elif macro_condition
   statements
#else
   statements
#endif
```

Important Points about Header files:
The creation of header files are needed generally while writing large C programs so that the modules can share the function definitions, prototypes etc.

- Function and type declarations, global variables, structure declarations and in some cases, inline functions; definitions which need to be centralized in one file.
- In a header file, do not use redundant or other header files; only minimal set of statements.
- Don’t put function definitions in a header. Put these things in a separate .c file.
- Include Declarations for functions and variables whose definitions will be visible to the linker. Also, definitions of data structures and enumerations that are shared among multiple source files.
- In short, Put only what is necessary and keep the header file concised.

Difference between Header file and Library

Header Files : The files that tell the compiler how to call some functionality (without knowing how the functionality actually works) are called header files. They contain the function prototypes. They also contain Data types and constants used with the libraries. We use #include to use these header files in programs. These files end with .h extension.
Library : Library is the place where the actual functionality is implemented i.e. they contain function body. Libraries have mainly two categories :

- Static
- Shared or Dynamic

Static : Static libraries contains object code linked with an end user application and then they become the part of the executable. These libraries are specifically used at compile time which means the library should be present in correct location when user wants to compile his/her C or C++ program. In windows they end with .lib extension and with .a for MacOS.

Shared or Dynamic : These libraries are only required at run-time i.e, user can compile his/her code without using these libraries. In short these libraries are linked against at compile time to resolve undefined references and then its distributed to the application so that application can load it at run time. For example, when we open our game folders we can find many .dll(dynamic link libraries) files. As these libraries can be shared by multiple programs, they are also called as shared libraries.These files end with .dll or .lib extensions. In windows they end with .dll extension.

Example : Math.h is a header file which includes the prototype for function calls like sqrt(), pow() etc, whereas libm.lib, libmmd.lib, libmmd.dll are some of the math libraries. In simple terms a header file is like a visiting card and libraries are like a real person, so we use visiting card(Header file) to reach to the actual person(Library).

Return values of printf() and scanf() in C/C++

What values do the printf() and scanf() functions return ?

- printf() : It returns total number of Characters Printed, Or negative value if an output error or an encoding error
- scanf() : It returns total number of Inputs Scanned successfully, or EOF if input failure occurs before the first receiving argument was assigned.

- **scanf()** reads input until it encounters whitespace, newline or End Of File(EOF) whereas **gets()** reads input until it encounters newline or End

Of File(EOF), gets() does not stop reading input when it encounters whitespace instead it takes whitespace as a string.

- **scanf** can read multiple values of different data types whereas **gets()** will only get character string data.

Use of fflush(stdin) in C

fflush() is typically used for output stream only. Its purpose is to clear (or flush) the output buffer and move the buffered data to console (in case of stdout) or disk (in case of file output stream).

Problem with scanf() when there is fgets()/gets()/scanf() after it

The problem with above code is scanf() reads an integer and leaves a newline character in buffer. So fgets() only reads newline and the string “test” is ignored by the program.

The similar problem occurs when scanf() is used in a loop.

We can notice that above program prints an extra “Enter a character” followed by an extra new line. This happens because every scanf() leaves a newline character in buffer that is read by next scanf.

How to solve above problem?

1. We can make scanf() to read a new line by using an extra “\n”, i.e., **scanf(“%d\n”, &x)** . In fact **scanf(“%d “, &x)** also works (Note extra space).
2. We can add a getchar() after scanf() to read an extra newline.

typedef and #define

- The typedef has the advantage that it obeys the scope rules. That means you can use the same name for the different types in different scope. It

can have file scope or block scope in which declare. In other words, #define does not follow the scope rule.
- typedef is compiler token while #define is a preprocessor token.
- typedef is ended with semicolon while #define does not terminate with a semicolon.
- typedef is used to give a new symbolic name to existing type while #define is used to create an alias of any type and value.

Advantages of typedef

- Increase the readability of the code. If you are using structure and function pointer in your code, you should use typedef it increaser the readability.
- typedef obeys the scope rules. That means you can use the same name for the different types in different scope.
- Using the typedef we can remove the extra keystroke, for example in structure if you will use the typedef then you do not require to write struct keyword at the time of variable declaration.
- It increases the portability of the code.

Little endian to Big endian conversion and its Importance

Generally, people who work on the high-level language do not take the interest in details of computer architecture and ignore the concept of endianness and also never think about the concept to convert little endian to big endian. But sometimes endianness became crucial issues in your program when you transmit data serially over the network from one computer to another computer.

So endianness is mainly important when you are reading and writing the data across the network from one system to another. If the sender and receiver computer have the different endianness, then receiver system would not receive the actual data which transmitted by the sender.

So to handle this scenario here, I am describing some important concept related to the endianness.

Endianness

The endianness is the bytes order in which data stored in the memory of the computer and it also describes the order of byte transmission over a digital link. In memory data store in which order it depends on the endianness of the system, if the system is big-endian then the MSB byte store first (means at lower address) and if the system is the little endian then LSB byte store first (means at lower address).

Storing of bytes in memory

In the previous article, “data alignment” we have already know that processor performs the operation on the word (the word can be 1, 2, 4 bytes), here I suppose word size is 4 bytes and data which need to store in memory is 32 bits means 4 bytes.

How is the data stored in memory it depends on the endianness of processor?

Suppose, 32 bits Data is 0x11223344.

Big-endian

The most significant byte of data stored at the lowest memory address.

Address	Value
00	0x11
01	0x22
02	0x33
03	0x44

Little-endian

The least significant byte of data stored at the lowest memory address.

Address	Value
00	0x44
01	0x33
02	0x22
03	0x11

Compilation Steps of C

- Preprocessing
- Compilation
- Assembly
- Linking

Preprocessing

The first stage of compilation is called preprocessing. In this stage, lines starting with a # character are interpreted by the *preprocessor* as *preprocessor commands*.Before interpreting commands, the preprocessor does some initial processing. This includes joining continued lines (lines ending with a \) and stripping comments.

To print the result of the preprocessing stage, pass the -E option to cc:

cc -E hello_world.c

Compilation

The second stage of compilation is confusingly enough called compilation. In this stage, the preprocessed code is translated to *assembly instructions* specific to the target processor architecture. These form an intermediate human readable language.

The existence of this step allows for C code to contain inline assembly instructions and for different *assemblers* to be used.

Some compilers also supports the use of an integrated assembler, in which the compilation stage generates *machine code* directly, avoiding the overhead of generating the intermediate assembly instructions and invoking the assembler.

To save the result of the compilation stage, pass the -S option to cc:

```
cc -S hello_world.c
```

Assembly

During the assembly stage, an assembler is used to translate the assembly instructions to machine code, or *object code*. The output consists of actual instructions to be run by the target processor.

To save the result of the assembly stage, pass the -c option to cc:

```
cc -c hello_world.c
```

Running the above command will create a file named hello_world.o, containing the object code of the program.

Linking

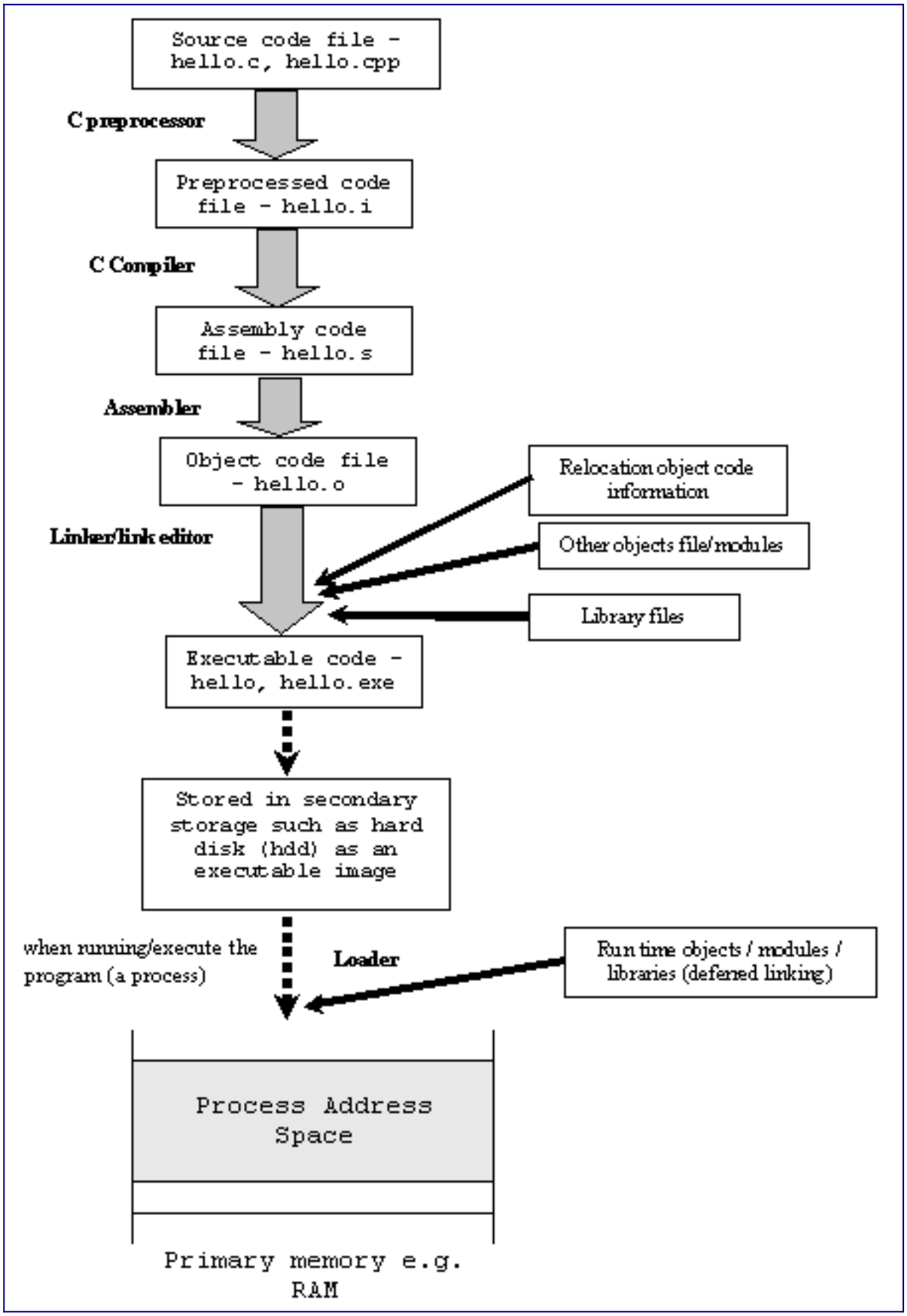
The object code generated in the assembly stage is composed of machine instructions that the processor understands but some pieces of the program are out of order or missing. To produce an executable program, the existing pieces have to be rearranged and the missing ones filled in. This process is called linking.

The *linker* will arrange the pieces of object code so that functions in some pieces can successfully call functions in other pieces. It will also add pieces containing the instructions for library functions used by the program. In the case of the “Hello, World!” program, the linker will add the object code for the puts function.

The result of this stage is the final executable program. When run without options, cc will name this file a.out. To name the file something else, pass the -ooption to cc:

cc -o hello_world hello_world.c

For your quick reference :



What is the (memmove vs memcpy) difference between memcpy and memmove?

Both copies function are used to copy n characters from the source object to destination object but they have some difference that is mentioned below.

- The memcpy copy function shows undefined behavior if the memory regions pointed to by the source and destination pointers overlap. The memmove function has the defined behavior in case of overlapping. So whenever in doubt, it is safer to use memmove in place of memcpy.

The memmove function is slower in comparison to memcpy because in memmove extra temporary array is used to copy n characters from the source and after that, it uses to copy the stored characters to the destination memory.

- The memcpy is useful in forwarding copy but memmove is useful in case of overlapping scenario.

Disadvantage of memmove and memcpy

- Both memcpy and memmove does not check the terminating null character, so carefully using with strings.
- The behavior of memcpy or memmove can be undefined if you try to access the destination and source buffer beyond their length.
- memcpy or memmove does not check the validity of the destination buffer.

Introduction of internal, external and none linkage in c

External Linkage:

If an identifier has a file scope and does not use the static storage class specifier at the time of the first declaration, the identifier has the external linkage.

The externally linked identifier or function visible to all translation unit of the program that means we can access it any translation unit of the program.

By default, all global identifier has the external linkage and each declaration of a particular identifier with external linkage denotes the same object or function.

In C language, extern keyword establishes external linkage. When we use the extern keyword, we say to the linker that definition of the identifier can be in another file. The externally linked identifier is accessed by any translation unit that's why it generally stored in initialized/uninitialized or text segment of RAM.

Internal Linkage:

If a global identifier declares with static storage class, its linkage will be internal. An identifier implementing internal linkage is not accessible outside the translation unit in which it is declared.

An identifier with internal linkage denotes the same object or function within one translation unit if it is accessed by any function.

None linkage:

A local variable has no linkage and refers to unique entities. If an identifier has the same name in another scope, they do not refer to the same object.

Recursion in C

Advantage and disadvantage of recursion

A recursive function has a lot of advantage and disadvantage. here I am mentioning few advantage and disadvantage of the recursive function.

Advantage:

1. Using the recursion you can make your code simpler and you can solve problems in an easy way while its iterative solution is very big and complex.

2. Recursive functions are useful to solve many mathematical problems, like generating Fibonacci series, calculating factorial of a number.

3. Recursive functions convenient for recursively defined data structures like trees.

Disadvantage:

1. Recursion makes your code slow because each time needs to create a stack frame for the function call.

2. Recursion takes a lot of stack memory because it requires the stack space in every invocation.

3. A recursive function should have a base condition to break the recursive calling of the function either it will cause of the stack overflow.

4. If recursion is too deep, then there is a danger of running out of space on the stack and it can cause of the program crashes.

Recursion vs Iteration

1. Due to the overhead of maintaining the stack usually, recursion is slower as compared to the iteration.

2. Usually, recursion uses more memory as compared to the iteration.

3. Sometimes it is much simpler to write the algorithm using recursion as compared to the iteration.

Bit field in c

In C language structure and union support a very important feature that is the bit field. The bit field allows the packing of data in a structure or union and prevents the wastage of memory.

Some important points about bit field in c

- If we are compiled the same C program that uses the bit-field on a different system, the result of the program may vary (C program may not work properly).
- The order of allocation of bit-fields within a unit low-order to high-order or high-order to low-order (depend on endianness)is implementation-defined.

- We can not create a pointer to the bit-field and also not use the address-of operator (&) to the bit-field member.
- We can not create an array of a bit field in c.
- The bit fields must also be long enough to contain the bit pattern ex short b: 17; /* Illegal! */
- We can not calculate the size of the bit field in c using the sizeof operator.

What is flexible array member in c?

This feature enables the user to create an empty array in a structure, the size of the empty array can be changed at runtime as per the user requirements. This empty array should be declared as the last member of the structure and the structure must contain at least one more named member.

When a structure has a flexible array member, the entire structure becomes an incomplete type. Basically, an incomplete type structure is a type that has lack of information about its member.

There is some example of incomplete type

- An array type whose dimension is not specified.
- A structure type whose members not specified completely.
- A union type whose members not specified completely.

An example of flexible array in c

```
typedef struct
{
   int iEmpId;
   float fSalary;
   char acName[];
}sEmployInfo;
```

In C99, you can use a structure whose last member is an array with an unspecified dimension, see the below structure

When we calculating the size of the above structure then we found that size of the structure includes all the member's size including the padding bytes (if required) but not including the size of the flexible array member.

For example, if the size of an int and float is 4 bytes and the alignment of the sEmployInfo is also 4 bytes, then the compiler will not insert any padding bytes after the members iEmpId and fSalary. So when we calculate the size of the structure, then it will be 8 bytes.

sizeof(sEmployInfo) = sizeof(iEmpId)+ sizeof(fSalary);

Why is a flexible array member required?

To understand the above question we need to take an example. Suppose there is an application which has a structure which contains the character array whose size is 30 bytes. This array is used to store the address of the users. Here, problem is that when you create a variable using this structure then every time compiler reserve 30 bytes for the array.

If the length of the user address is less than 30 bytes, the extra remaining memory of array would be a waste.In some scenario, it can be possible the length of the address could be greater than 30 bytes, this type of situation creates a problem and might be you will get the boundary issues.