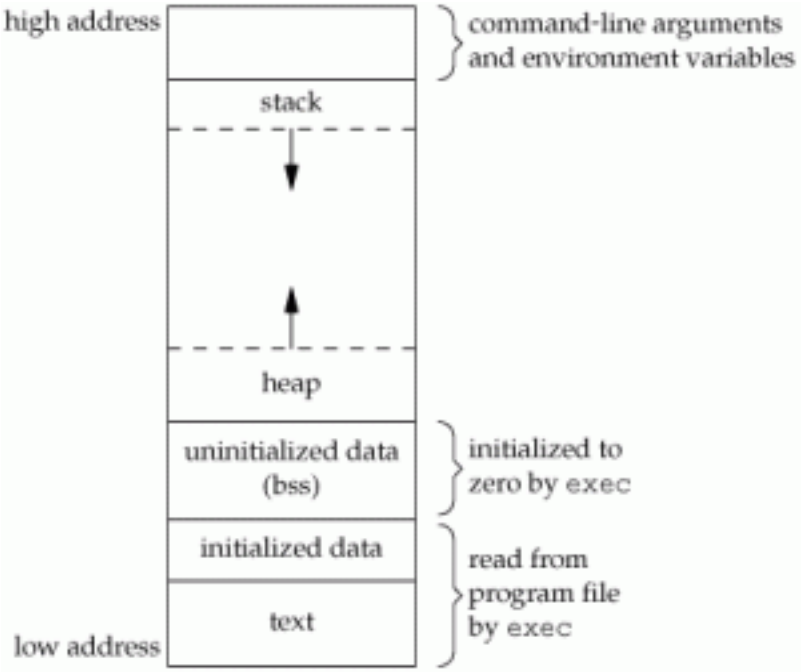
1. Storage Class Specifier

Memory Layout of C Programs

A typical memory representation of C program consists of following sections.

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Heap
5. Stack



A typical memory layout of a running process

1. Text Segment / Code Segment:

A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

2. Initialized Data Segment:

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the defined by

char s [] = “hello world”; (global string) stored in initialized read-write area
int debug = 1; (outside the main i.e. global) initialized read-write area.
const char* string = “hello world”; (global) makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.
Ex: 1. static int i = 10; stored in data segment
2. global int i = 10; stored in data segment

3. Uninitialized Data Segment:

Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing.

Uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

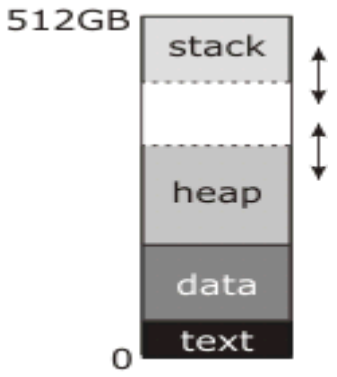
For instance a variable declared static int i; would be contained in the BSS segment.

For instance a global variable declared int j; would be contained in the BSS segment.

4. Heap:

Heap is the segment where dynamic memory allocation usually takes place.

The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

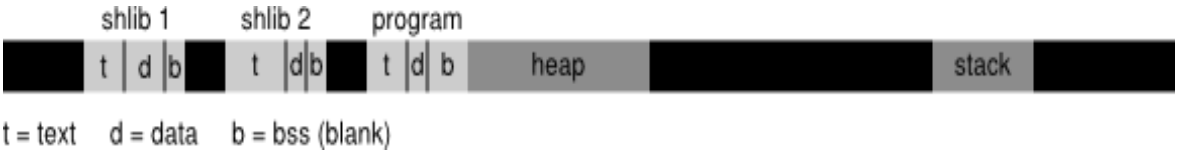


The "break"--the address manipulated by `brk` and `sbrk`--is the dotted line at the top of the *heap*. The documentation you've read describes this as the end of the "data segment" because in traditional (pre-shared-libraries, pre-`mmap`) Unix the data segment was continuous with the heap; before program start, the kernel would load the "text" and "data" blocks into RAM starting at address zero (actually a little above address zero, so that the NULL pointer genuinely didn't point to anything) and set the break address to the end of the data segment. The first call to `malloc` would then use `sbrk` to move the break up and create the heap *in between* the top of the data segment and the new, higher break address, as shown in the diagram, and subsequent use of `malloc` would use it to make the heap bigger as necessary.

Meantime, the stack starts at the top of memory and grows down. The stack doesn't need explicit system calls to make it bigger; either it starts off with as much RAM allocated to it as it can ever have (this was the traditional approach) or there is a region of reserved addresses below the stack, to which the kernel automatically allocates RAM when it notices an attempt to write there (this is the modern approach). Either way, there

may or may not be a "guard" region at the bottom of the address space that can be used for stack. If this region exists (all modern systems do this) it is permanently unmapped; if *either* the stack or the heap tries to grow into it, you get a segmentation fault. Traditionally, though, the kernel made no attempt to enforce a boundary; the stack could grow into the heap, or the heap could grow into the stack, and either way they would scribble over each other's data and the program would crash. If you were very lucky it would crash immediately.

I'm not sure where the number 512GB in this diagram comes from. It implies a 64-bit virtual address space, which is inconsistent with the very simple memory map you have there. A real 64-bit address space looks more like this:



This is not remotely to scale, and it shouldn't be interpreted as exactly how any given OS does stuff (after I drew it I discovered that Linux actually puts the executable much closer to address zero than I thought it did, and the shared libraries at surprisingly high addresses). The black regions of this diagram are unmapped -- any access causes an immediate segfault -- and they are *gigantic* relative to the gray areas. The light-gray regions are the program and its shared libraries (there can be dozens of shared libraries); each has an *independent* text and data segment (and "bss" segment, which also contains global data but is initialized to all-bits-zero rather than taking up space in the executable or library on disk). The heap is no longer necessarily continous with the executable's data segment -- I drew it that way, but it looks like Linux, at least, doesn't do that. The stack is no longer pegged to the top of the virtual address space, and the distance between the heap and the stack is so enormous that you don't have to worry about crossing it.

The break is still the upper limit of the heap. However, what I didn't show is that there could be dozens of independent allocations of memory off there in the black somewhere, made with `mmap` instead of `brk`. (The OS will try to keep these far away from the `brk` area so they don't collide.)

5. Stack:

The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern

large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)

The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; a stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

Summary

- global variables ➔ Data
- static variables ➔ data
- Constant data types ➔ code and/or data. Consider string literals for a situation when a constant itself would be stored in the data segment, and references to it would be embedded in the code
    - local constant variables ➔ stack
    - initialized global constant variable ➔ data segment
    - uninitialized global constant variable ➔ bss
- local variables(declared and defined in functions) ➔ stack
- variables declared and defined in `main` function ➔ stack
- pointers(ex: `char *arr`, `int *arr`) ➔ data or stack, depending on the context. C lets you declare a global or a `static` pointer, in which case the pointer itself would end up in the data segment.
- dynamically allocated space(using `malloc`, `calloc`, `realloc`) ➔ heap

It is worth mentioning that "stack" is officially called "automatic storage class".

Examples.

The size (1) command reports the sizes (in bytes) of the text, data, and bss segments.

1. Check the following simple C program

```
#include <stdio.h>
 int main(void)
{
    return 0;
}
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        248          8       1216        4c0    memory-layout
```

2. Let us add one global variable in program, now check the size of bss (highlighted in red color).

```
#include <stdio.h>
 int global; /* Uninitialized variable stored in bss*/
 int main(void)
{
    return 0;
}
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
 960        248         12       1220        4c4    memory-layout
```

3. Let us add one static variable which is also stored in bss.

```
#include <stdio.h>
 int global; /* Uninitialized variable stored in bss*/
 int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    return 0;
}
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
 960        248         16       1224        4c8    memory-layout
```

4. Let us initialize the static variable which will then be stored in Data Segment (DS)

```
#include <stdio.h>
 int global; /* Uninitialized variable stored in bss*/
 int main(void)
```

```
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        252         12         1224        4c8    memory-layout
```

5. Let us initialize the global variable which will then be stored in Data Segment (DS)

```
#include <stdio.h>
int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        256         8          1224        4c8    memory-layout
```

Scope rules in C

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. C scope rules can be covered under following two categories.

Global Scope/ Extern: Can be accessed anywhere in a program.

```
// filename: file1.c
int a;
int main(void)
{
   a = 2;
}
// filename: file2.c
// When this file is linked with file1.c, functions of this file can
access a
extern int a;
int myfun()
{
   a = 2;
}
```

To restrict access to current file only, global variables can be marked as static.

Block Scope: A Block is a set of statements enclosed within left and right braces ({and} respectively). Blocks may be nested in C (a block may contain other blocks inside it). A variable declared in a block is accessible in the block and all inner blocks of that block, but not accessible outside the block.

What if the inner block itself has one variable with the same name?
If an inner block declares a variable with the same name as the variable declared by the outer block, then the visibility of the outer block variable ends at the pint of declaration by inner block.

```
int main()
{
  {
      int x = 10, y  = 20;
      {
          // The outer block contains declaration of x and y, so
          // following statement is valid and prints 10 and 20
          printf("x = %d, y = %d\n", x, y);
          {
              // y is declared again, so outer block y is not accessible
              // in this block
              int y = 40;

              x++;  // Changes the outer block variable x to 11
              y++;  // Changes this block's variable y to 41

              printf("x = %d, y = %d\n", x, y);
          }

          // This statement accesses only outer block's variables
          printf("x = %d, y = %d\n", x, y);
      }
  }
  return 0;
}
```

Output:

```
x = 10, y = 20
x = 11, y = 41
x = 11, y = 20
```

What about functions and parameters passed to functions?

A function itself is a block. Parameters and other local variables of a function follow the same block scope rules.

Can variables of block be accessed in another subsequent block?

No, a variabled declared in a block can only be accessed inside the block and all inner blocks of this block. For example, following program produces compiler error.

```
int main()
{
  {
      int x = 10;
  }
  {
      printf("%d", x);  // Error: x is not accessible here
  }
  return 0;
}
```

Output:

```
error: 'x' undeclared (first use in this function)
```

As an exercise, predict the output of following program.

```
int main()
{
  int x = 1, y = 2, z = 3;
  printf(" x = %d, y = %d, z = %d \n", x, y, z);
  {
       int x = 10;
       float y = 20;
       printf(" x = %d, y = %f, z = %d \n", x, y, z);
       {
             int z = 100;
             printf(" x = %d, y = %f, z = %d \n", x, y, z);
       }
  }
  return 0;
}
```

Storage Classes

1. Auto keyword

The default storage class of any variable is Automatic. It is created as soon as the declaration statement is encountered and is destroyed as soon as the program control leaves the block which contains your variable.

Keyword used – auto

For example, here we define two variables, count, which is a local variable and second, counter inside a block (local variable using auto class).

```
{
  int count;
  auto int counter;
}
```

2. Extern keyword

Extern keyword applies to C variables (data objects) and C functions. Basically extern keyword extends the visibility of the C variables and C functions.

Declaration of a variable/function simply declares that the variable/function exists somewhere in the program but the memory is not allocated for them. But the declaration of a variable/function serves an important role. And that is the type of the variable/function. Therefore, when a variable is declared, the program knows the data type of that variable. In case of function declaration, the program knows what the arguments to that functions are, their data types, the order of arguments and the return type of the function. So that's all about declaration. Coming to the definition, when we define a variable/function, apart from the role of declaration, it also allocates memory for that variable/function. Therefore, we can think of definition as a super set of declaration. (Or declaration as a subset of definition). From this explanation, it should be obvious that a variable/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/function).

i. Use of extern with C functions.

By default, the declaration and definition of a C function have "extern" prepended with them. It means even though we don't use extern with the declaration/definition of C functions, it is present there. For example, when we write

```
int foo(int arg1, char arg2);
```

There's an extern present in the beginning which is hidden and the compiler treats it as below.

```
extern int foo(int arg1, char arg2);
```

Same is the case with the definition of a C function (definition of a C function means writing the body of the function). Therefore whenever we define a C function, an extern is present there in the beginning of the function definition. Since the declaration can be done any number of times and definition can be done only once, we can notice that declaration of a function can be added in several C/H files or in a single C/H file several times. But we notice the actual definition of the function only once (i.e. in one file only). And as the extern extends the visibility to the whole program, the functions can be used (called) anywhere in any of the files of the whole program provided the declaration of the function is known. (By knowing the declaration of the function, C compiler knows that the definition of the function exists and it goes ahead to compile the program). So that's all about extern with C functions.

ii. Use of extern with C variables.

It is more interesting and information than the previous case where extern is present by default with C functions. So let me ask the question, how would you declare a C variable without defining it? Many of you would see it trivial but its important question to understand extern with C variables. The answer goes as follows.

```
extern int var;
```

Here, an integer type variable called var has been declared (remember no definition i.e. no memory allocation for var so far). And we can do this declaration as many times as needed. (Remember that declaration can be done any number of times)

Now how would you define a variable? Now I agree that it is the most trivial question in programming and the answer is as follows.

```
int var;
```

Here, an integer type variable called var has been declared as well as defined. (Remember that definition is the super set of declaration). Here the memory for var is also allocated. Now here comes the surprise, when we declared/defined a C function, we saw that an extern was present by default. While defining a function, we can prepend it with extern without any issues. But it is not the case with C variables. If we put the

presence of extern in variable as default then the memory for them will not be allocated ever, they will be declared only. Therefore, we put extern explicitly for C variables when we want to declare them without defining them. Also, as the extern extends the visibility to the whole program, by externing a variable we can use the variables anywhere in the program provided we know the declaration of them and the variable is defined somewhere.

Example 1:

```
int var;
int main(void)
{
   var = 10;
   return 0;
}
```

Analysis: This program is compiled successfully. Here var is defined (and declared implicitly) globally.

Example 2:

```
extern int var;
int main(void)
{
  return 0;
}
```

Analysis: This program is compiled successfully. Here var is declared only. Notice var is never used so no problems.

Example 3:

```
extern int var;
int main(void)
{
 var = 10;
 return 0;
}
```

Analysis: This program throws error in compilation. 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:

```
#include "somefile.h"
extern int var;
int main(void)
{
```

```
 var = 10;
 return 0;
}
```

Analysis: Supposing that somefile.h has the definition of var. This program will be compiled successfully.

Example 5:

```
extern int var = 0;
int main(void)
{
 var = 10;
 return 0;
}
```

Analysis: Guess 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 i.e. that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work.

For example, let's have two files, first which would have our main function (main.c) and second one having a function named extern_storage (extern.c)

First file : main.c

```
int counter = 1 ; // Global Variable

void main()
{
  extern_storage();
  counter++;
  extern_storage();
  counter--;
  extern_storage();
}
```

Second file : extern.c

```
void extern_storage(void);

extern int counter;

void extern_storage(void)
{
   printf("Current value of extern counter is %d n", &counter);
}
```

When the above two files are executed, the function extern_storage picks up the current value from our main.c file. So, the following output will be produced.

```
Current value of extern counter is 1
Current value of extern counter is 2
Current value of extern counter is 1
```

In short, we can say

1. Declaration can be done any number of times but definition only once.
2. “extern” keyword is used to extend the visibility of variables/functions().
3. Since functions are visible through out the program by default. The use of extern is not needed in function declaration/definition. Its use is redundant.
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 definition of the variable as well.

How Linkers Resolve Global Symbols Defined at Multiple Places?

At compile time, the compiler exports each global symbol to the assembler as either strong or weak, and the assembler encodes this information implicitly in the symbol table of the relocatable object file. Functions and initialized global variables get strong symbols. Uninitialized global variables get weak symbols.

For the following example programs, buf, bufp0, main, and swap are strong symbols; bufp1 is a weak symbol.

```
/* main.c */
 void swap();
 int buf[2] = {1, 2};
 int main()
 {
   swap();
   return 0;
 }
  /* swap.c */
 extern int buf[];

 int *bufp0 = &buf[0];
 int *bufp1;

 void swap()
 {
   int temp;
    bufp1 = &buf[1];
   temp = *bufp0;
   *bufp0 = *bufp1;
```

```
    *bufp1 = temp;
}
```

Given this notion of strong and weak symbols, Unix linkers use the following rules for dealing with multiply defined symbols:

Rule1: Multiple strong symbols are not allowed.
Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol.
Rule 3: Given multiple weak symbols, choose any of the weak symbols.

For example, suppose we attempt to compile and link the following two C modules:

```
/* foo1.c */
int main()
{
  return 0;
}
 /* bar1.c */
int main()
{
  return 0;
}
```

In this case, the linker will generate an error message because the strong symbol main is defined multiple times (rule 1):

```
unix> gcc foo1.c bar1.c
/tmp/cca015022.o: In function ‘main’:
/tmp/cca015022.o(.text+0x0): multiple definition of ‘main’
/tmp/cca015021.o(.text+0x0): first defined here
```

Similarly, the linker will generate an error message for the following modules because the strong symbol x is defined twice (rule 1):

```
/* foo2.c */
int x = 15213;
int main()
{
  return 0;
}
 /* bar2.c */
int x = 15213;
void f()
{
}
```

However, if x is uninitialized in one module, then the linker will quietly choose the strong symbol defined in the other (rule 2) as is the case in following program:

```
/* foo3.c */
#include <stdio.h>
```

```
void f(void);
int x = 15213;
int main()
{
  f();
  printf("x = %d\n", x);
  return 0;
}
 /* bar3.c */
int x;
void f()
{
  x = 15212;
}
```

At run time, function f() changes the value of x from 15213 to 15212, which might come as a unwelcome surprise to the author of function main! Notice that the linker normally gives no indication that it has detected multiple definitions of x.

```
unix> gcc -o foobar3 foo3.c bar3.c
unix> ./foobar3
x = 15212
```

The same thing can happen if there are two weak definitions of x (rule 3).

3. Register keyword

Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using register keyword. The keyword register hints to compiler that a given variable can be put in a register. Its compiler's choice to put it in a register or not. Generally, compilers themselves do optimizations and put the variables in register.

1) If you use & operator with a register variable then compiler may give an error or warning (depending upon the compiler you are using), because when we say a variable is a register, it may be stored in a register instead of memory and accessing address of a register is invalid. Try below program.

```
int main(){
  register int i = 10;
  int *a = &i;
  printf("%d", *a);
  getchar();
  return 0;
}
```

2) Register keyword can be used with pointer variables. Obviously, a register can have address of a memory location. There would not be any problem with the below program.

```
int main(){
```

```
  int i = 10;
  register int *a = &i;
  printf("%d", *a);
  getchar();
  return 0;
}
```

3) Register is a storage class, and C doesn't allow multiple storage class specifiers for a variable. So, register can not be used with static. Try below program.

```
int main()
{
  int i = 10;
  register static int *a = &i;
  printf("%d", *a);
  getchar();
  return 0;
}
```

4) There is no limit on number of register variables in a C program, but the point is compiler may put some variables in register and some not

4. Static Variables and Static Functions

Before moving ahead, let's quickly understand the difference between life time and scope of a variable. A region in code where a variable can be accessed is known as its scope and the duration during which a variable remains active is known as its life time.

i. Static Variables

a. Impact on Life Time

Static variables are those variables whose life time remains equal to the life time of the program. Any local or global variable can be made static depending upon what the logic expects out of that variable. Let's consider the following example:

```
#include<stdio.h>

char** func_Str();

int main(void)
{
    char **ptr = NULL;
    ptr = func_Str();
    printf("\n [%s] \n",*ptr);

    return 0;
}

char** func_Str()
```

```
{
    char *p = "Linux";
    return &p;
}
```

In the code above, the function ‘func_str()’ returns the address of the pointer ‘p’ to the calling function which uses it further to print the string ‘Linux’ to the user through ‘printf()’. Lets look at the output :

```
$ ./static

 [Linux]
$
```

The output above is as expected. So, is everything fine here? Well, there is a hidden problem in the code. More specifically, it’s the return value of the function ‘func_Str()’. The value being returned is the address of the local pointer variable ‘p’. Since ‘p’ is local to the function so as soon as the function returns, the lifetime of this variable is over and hence its memory location becomes free for any further modifications.

Let’s prove this observation. Look at the code below:

```
#include<stdio.h>

char** func1_Str();
char** func2_Str();

int main(void)
{
    char **ptr1 = NULL;
    char **ptr2 = NULL;

    ptr1 = func1_Str();
    printf("\n [%s] \n",*ptr1);

    ptr2 = func2_Str();
    printf("\n [%s] \n",*ptr2);

    printf("\n [%s] \n",*ptr1);

    return 0;
}

char** func1_Str()
{
    char *p = "Linux";
    return &p;
}
```

```
char** func2_Str()
{
    char *p = "Windows";
    return &p;
}
```

In the code above, now there are two functions ‘func1_Str()’ and ‘func2_Str()’. The logical problem remains the same here too. Each of these function returns the address of its local variable. In the main() function, the address returned by the func1_Str() is used to print the string ‘Linux’ (as pointed by its local pointer variable) and the address returned by the function func2_Str() is used to print the string ‘Windows’ (as pointed by its local pointer variable). An extra step towards the end of the main() function is done by again using the address returned by func1_Str() to print the string ‘Linux’.

Now, let’s see the output:

```
$ ./static
 [Linux]
 [Windows]
 [Windows]
$
```

The output above is not as per expectations. The third print should have been ‘Linux’ instead of ‘Windows’. Well, I’d rather say that the above output was expected. It’s just the correct scenario that exposed the loophole in the code.

Let’s go a bit deeper to see what happened after address of local variable was returned. See the code below:

```
#include<stdio.h>

char** func1_Str();
char** func2_Str();

int main(void)
{
    char **ptr1 = NULL;
    char **ptr2 = NULL;

    ptr1 = func1_Str();
    printf("\n [%s] :: func1_Str() address = [%p], its returned address
is [%p]\n",*ptr1,(void*)func1_Str,(void*)ptr1);

    ptr2 = func2_Str();
    printf("\n [%s] :: func2_Str()address = [%p], its returned address
is [%p]\n",*ptr2,(void*)func2_Str,(void*)ptr2);
```

```
    printf("\n [%s] [%p]\n",*ptr1,(void*)ptr1);

    return 0;
}

char** func1_Str()
{
    char *p = "Linux";
    return &p;
}

char** func2_Str()
{
    char *p = "Windows";
    return &p;
}
```

The code is above is modified to print the address of the functions and the address of their respective local pointer variables. Here is the output:

```
$ ./static

 [Linux] :: func1_Str() address = [0x4005d5], its returned address is
[0x7fff705e9378]

 [Windows] :: func2_Str()address = [0x4005e7], its returned address is
[0x7fff705e9378]

 [Windows] [0x7fff705e9378]
$
```

The above output makes it clear that once the lifetime of the local variable of the function ‘func1_Str()’ gets over then same memory address is being used for the local pointer variable of the function ‘func2_Str()’ and hence the third print is ‘Windows’ and not ‘Linux’.

So, now we see what that the root of the problem is the life time of the pointer variables. This is where the ‘static’ storage class comes to rescue. As already discussed the static storage class makes the lifetime of a variable equal to that of the program. So, let’s make the local pointer variables as static and then see the output:

```
#include<stdio.h>

char** func1_Str();
char** func2_Str();

int main(void)
```

```
{
    char **ptr1 = NULL;
    char **ptr2 = NULL;

    ptr1 = func1_Str();
    printf("\n [%s] :: func1_Str() address = [%p], its returned address
is [%p]\n",*ptr1,(void*)func1_Str,(void*)ptr1);

    ptr2 = func2_Str();
    printf("\n [%s] :: func2_Str()address = [%p], its returned address
is [%p]\n",*ptr2,(void*)func2_Str,(void*)ptr2);

    printf("\n [%s] [%p]\n",*ptr1,(void*)ptr1);

    return 0;
}

char** func1_Str()
{
    static char *p = "Linux";
    return &p;
}

char** func2_Str()
{
    static char *p = "Windows";
    return &p;
}
```

Note that in code above, the pointers were made static. Here is the output:

```
$ ./static

 [Linux] :: func1_Str() address = [0x4005d5], its returned address is
[0x601028]

 [Windows] :: func2_Str()address = [0x4005e0], its returned address is
[0x601020]

 [Linux] [0x601028]
```

So we see that after making the variables as static, the lifetime of the variables becomes equal to that of the program.

b. Impact on Scope

In case where code is spread over multiple files, the static storage type can be used to limit the scope of a variable to a particular file. For example, if we have a variable ‘count’ in one file and we want to have another variable with same name in some other

file, then in that case one of the variable has to be made static. The following example illustrates it:

Here we use two files (static.c and static_1.c)

```
//static.c

#include<stdio.h>
int count = 1;

int main(void)
{
    printf("\n count = [%d]\n",count);
    return 0;
}
// static_1.c

#include<stdio.h>
int count = 4;

int func(void)
{
    printf("\n count = [%d]\n",count);
    return 0;
}
```

Now, when both the files are compiled and linked together to form a single executable, here is the error that is thrown by gcc :

```
$ gcc -Wall static.c static_1.c -o static
/tmp/ccwO66em.o:(.data+0x0): multiple definition of `count'
/tmp/ccGwx5t4.o:(.data+0x0): first defined here
collect2: ld returned 1 exit status
$
```

So we see that gcc complains of multiple declarations of the variable ‘count’.

As a corrective measure, this time one of the ‘count’ variable is made static:

```
//static.c

#include<stdio.h>
static int count = 1;

int main(void)
{
    printf("\n count = [%d]\n",count);
    return 0;
}
// static_1.c
```

```
#include<stdio.h>
int count = 4;

int func(void)
{
    printf("\n count = [%d]\n",count);
    return 0;
}
```

Now, if both the files are compiled and linked together:

```
$ gcc -Wall static.c static_1.c -o static
$
```

So we see that no error is thrown this time because static limited the scope of the variable ‘count’ in file static.c to the file itself.

Initialization of static variables in C

In C, static variables can only be initialized using constant literals. For example, following program fails in compilation.

```
#include<stdio.h>
int initializer(void)
{
    return 50;
}

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

If we change the program to following, then it works without any error.

```
#include<stdio.h>
int main()
{
    static int i = 50;
    printf(" value of i = %d", i);
    getchar();
    return 0;
}
```

The reason for this is simple: All objects with static storage duration must be initialized (set to their initial values) before execution of main() starts. So a value which is not known at translation time cannot be used for initialization of static variables.

ii. Static Functions

By default any function that is defined in a C file is extern. This means that the function can be used in any other source file of the same code/project (which gets compiled as separate translational unit). Now, if there is a situation where the access to a function is to be limited to the file in which it is defined or if a function with same name is desired in some other file of the same code/project then the functions in C can be made static.

Extending the same example that was used in previous section, suppose we have two files :

```
//static.c

#include<stdio.h>
void func();

int main(void)
{    func();
    return 0;
}

void funcNew()
{    printf("\n Hi, I am a normal function\n");
}
// static_1.c

#include<stdio.h>
void funcNew();

int func(void)
{    funcNew();
    return 0;
}
```

If we compile, link and run the code above :

```
$ gcc -Wall static.c static_1.c -o static
$ ./static

 Hi, I am a normal function
$
```

So we see that the function funcNew() was defined in one file and successfully got called from the other. Now, if the file static_1.c wants to have its own funcNew(), ie :

```
// static_1.c

#include<stdio.h>
```

```
void funcNew();

int func(void)
{
    funcNew();
    return 0;
}

void funcNew()
{
    printf("\n Hi, I am a normal function\n");
}
```

Now, if both the files are compiled and linked together :

```
$gcc -Wall static.c static_1.c -o static

/tmp/ccqI0jsP.o: In function `funcNew':

static_1.c:(.text+0x15): multiple definition of `funcNew'

/tmp/ccUO2XFS.o:static.c:(.text+0x15): first defined here

collect2: ld returned 1 exit status

$
```

So we see that the compiler complains of multiple definitions of the function funcNew(). So, we make the funcNew() in static_1.c as static :

```
// static_1.c

#include<stdio.h>

static void funcNew();

int func(void)
{
    funcNew();
    return 0;
}

static void funcNew()
{
    printf("\n Hi, I am also a normal function\n");
}
```

Now, if we compile, then we see that the compiler never complains:

```
$ gcc -Wall static.c static_1.c -o static
$ ./static
```

```
 Hi, I am also a normal function
$
```

Similarly, if static.c wants that its funcNew() should be accessible from within static.c only then in that case funcNew() in static.c can be made static.

Static Variables

There are 2 distinct uses of the static keyword in C:

- Static declarations in a function's scope
  Inside a function it means that the static variable will remain in existence after the function has exited
- Static declarations outside of a function's scope
  It means that the static variable or function is local to that compilation unit (“file”), i.e. not externally visible

EVIL: Static Variables in a function

A static variable in a function is used as a "memory" state.

Basically, your variable is initialized to your default value only the first time you call it, and then retains its previous value in all the future calls.

It is potentially useful if you need to remember such state, but the use of such statics is usually frowned upon because they are pretty much global variables in disguise: they will consume your memory until the termination of your process once.

So, in general, making localized functions is EVIL / BAD.

Example:

```
#include <stdio.h>
int  ping() {
  static int counter = 0;
  return (++counter);
}
int   main(int ac, char **av) {
  printf("%d\n", ping()); // outputs 1
  printf("%d\n", ping()); // outputs 2
  return (0);
}
```

Output:

```
1
2
```

GOOD: Static Variables outside of a function's scope

You can use static outside of a function on a variable or function (which, after all, is sort of a variable as well and points to a memory address).

What it does is limit the use of that variable to the file containing it. You cannot call it from somewhere else. While it still means that that function/var is "global" in the sense that it consumes your memory until your program's termination, at least it has the decency to not pollute your "namespace".

This is interesting because that way you can have small utility functions with identical names in different files of your project.

So, *in general*, making localized functions is **GOOD**.

Example:

example.h

```
#ifndef __EXAMPLE_H__
# define __EXAMPLE_H__

void  function_in_other_file(void);

#endif
```

file1.c

```
#include <stdio.h>
#include "example.h"

static void test(void);

void test(void) {
  printf("file1.c: test()\n");
}

int main(int ac, char **av) {
  test();  // calls the test function declared above (prints "file1.c:
test()")
  function_in_other_file();
  return (0);
}
```

file2.c

```
#include <stdio.h>
#include "example.h"

static void test(void); // that's a different test!!

void test(void) {
  printf("file2.c: test()\n");
```

```
}

void function_in_other_file(void) {
  test();  // prints file2.c: test()
  return (0);
}
```

Output:

```
file1.c: test()
file2.c: test()
```

Format Specifier

%c single character
%d and %i both used for integer type
%u used for representing unsigned integer
%o octal integer unsigned
%x,%X used for representing hex unsigned integer
%e, %E, %f, %g, %G floating type
%s strings that is sequence of characters
format specifiers for integer data type

short signed %d or %i
short unsigned %u
long signed %ld
long unsigned %lu
unsigned hexadecimal %x
unsigned octal %o

format specifiers for real data type

float %f
double %lf

format specifiers for character/string data type

signed character %c
unsigned character %c
string %s

Understanding "volatile" qualifier in C

The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

Objects declared as volatile are omitted from optimization because their values can be changed by code outside the scope of current code at any time. The system always reads the current value of a volatile object from the memory location rather than keeping its value in temporary register at the point it is requested, even if a previous instruction asked for a value from the same object. So the simple question is, how can value of a variable change in such a way that compiler cannot predict. Consider the following cases for answer to this question.

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

2) Global variables within a multi-threaded application: There are multiple ways for threads communication, viz, message passing, shared memory, mail boxes, etc. A global variable is weak form of shared memory. When two threads sharing information via global variable, they need to be qualified with volatile. Since threads run asynchronously, any update of global variable due to one thread should be fetched freshly by another consumer thread. Compiler can read the global variable and can place them in temporary variable of current thread context. To nullify the effect of compiler optimizations, such global variables to be qualified as volatile

If we do not use volatile qualifier, the following problems may arise
1) Code may not work as expected when optimization is turned on.
2) Code may not work as expected when interrupts are enabled and used.

Let us see an example to understand how compilers interpret volatile keyword. Consider below code, we are changing value of const object using pointer and we are compiling code without optimization option. Hence compiler won't do any optimization and will change value of const object.

```
/* Compile code without optimization option */
#include <stdio.h>
int main(void)
{
    const int local = 10;
    int *ptr = (int*) &local;

    printf("Initial value of local : %d \n", local);

    *ptr = 100;

    printf("Modified value of local: %d \n", local);

    return 0;
}
```

When we compile code with "–save-temps" option of gcc it generates 3 output files

1) preprocessed code (having .i extention)
2) assembly code (having .s extention) and
3) object code (having .o option).

We compile code without optimization, that's why the size of assembly code will be larger (which is highlighted in red color below).

Output:

```
[narendra@ubuntu]$ gcc volatile.c -o volatile -save-temps
[narendra@ubuntu]$ ./volatile
Initial value of local : 10
Modified value of local: 100
[narendra@ubuntu]$ ls -l volatile.s
-rw-r-r- 1 narendra narendra 731 2016-11-19 16:19 volatile.s
[narendra@ubuntu]$
```

Let us compile same code with optimization option (i.e. -O option). In thr below code, "local" is declared as const (and non-volatile), GCC compiler does optimization and ignores the instructions which try to change value of const object. Hence value of const object remains same.

```
/* Compile code with optimization option */
#include <stdio.h>
```

```
int main(void)
{
    const int local = 10;
    int *ptr = (int*) &local;

    printf("Initial value of local : %d \n", local);

    *ptr = 100;

    printf("Modified value of local: %d \n", local);

    return 0;
}
```

For above code, compiler does optimization, that's why the size of assembly code will reduce.

Output:

```
  [narendra@ubuntu]$ gcc -O3 volatile.c -o volatile -save-temps
  [narendra@ubuntu]$ ./volatile
  Initial value of local : 10
  Modified value of local: 10
  [narendra@ubuntu]$ ls -l volatile.s
  -rw-r-r- 1 narendra narendra 626 2016-11-19 16:21 volatile.s
```

Let us declare const object as volatile and compile code with optimization option. Although we compile code with optimization option, value of const object will change, because variable is declared as volatile that means don't do any optimization.

```
/* Compile code with optimization option */
#include <stdio.h>

int main(void)
{
    const volatile int local = 10;
    int *ptr = (int*) &local;

    printf("Initial value of local : %d \n", local);

    *ptr = 100;

    printf("Modified value of local: %d \n", local);

    return 0;
}
```

Output:

```
  [narendra@ubuntu]$ gcc -O3 volatile.c -o volatile -save-temp
  [narendra@ubuntu]$ ./volatile
```

```
Initial value of local : 10
Modified value of local: 100
[narendra@ubuntu]$ ls -l volatile.s
-rw-r-r- 1 narendra narendra 711 2016-11-19 16:22 volatile.s
[narendra@ubuntu]$
```

The above example may not be a good practical example; the purpose was to explain how compilers interpret volatile keyword. As a practical example, think of touch sensor on mobile phones. The driver abstracting touch sensor will read the location of touch and send it to higher level applications. The driver itself should not modify (const-ness) the read location, and make sure it reads the touch input every time fresh (volatile-ness). Such driver must read the touch sensor input in const volatile manner.

Refer following links for more details on volatile keyword:

Volatile: A programmer's best friend

Do not use volatile as a synchronization primitive

What is the strict aliasing rule?

So What's The Problem?

There's a lot of confusion about strict aliasing rules. The main source of people's confusion is that there are two different audiences that talk about aliasing, developers who use compilers, and compiler writers. In this document I'm going to try to clear it all up for you. The things that I'm going to cover are based on the aliasing rules in C89/90 (6.3), C98/99 (6.5/7) as well as in C++98 (3.10/15), and C++11 (3.10/10). To find the aliasing rules in any current version of the C or C++ standards, search for "may not be aliased", which will find a footnote that refers back up to the section on allowable forms of aliasing. For information about what was on the mind of the creators of the spec, see C89 Rationale Section 3.3 Expressions, where they talk about why and how the aliasing rules came about.

Developers get interested in aliasing when a compiler gives them a warning about type punning and strict aliasing rules and they try to understand what the warnings mean. They Google for the warning message, they find references to the section on aliasing in one of the C or C++ specs and think, "Yes, that's what I'm trying to do, alias." Then they study that section of the appropriate spec like they're studying arcane runes and try to divine the rules that will let them do the things that they're trying to do. They think

that the aliasing rules are written to tell them how to do type punning. They couldn't be more wrong.

The compiler writers know what the strict aliasing rules are for. They are written to let compiler writers know when they can safely assume that a change made through one variable won't affect the value of another variable, and conversely when they have to assume that two variables might actually refer to the same spot in memory.

So this document is divided into two parts. First I'll talk about what strict aliasing is and why it exists, and then I'll talk about how to do the kinds of things developers need to do in ways that won't come in conflict with those rules.

1. What is aliasing exactly?

Aliasing is when more than one lvalue refers to the same memory location (when you hear lvalue, think of things (variables) that can be on the left-hand side of assignments), i.e. that are modifiable. As an example:

```
int anint;
int *intptr=&anint;
```

If you change the value of *intptr, the value referenced by anint also changes because *intptr aliases anint, it's just another name for the same thing. Another example is:

```
int anint;
void foo(int &i1, int &i2);
foo(anint,anint);
```

Within the body of foo since we used anint for both arguments, the two references, i1, and i2 alias, i.e. refer to the same location when foo is called this way.

What's the problem?

Examine the following code:

```
int anint;

void foo(double *dblptr)
{
    anint=1;
    *dblptr=3.14159;
    bar(anint);
}
```

Looking at this, it looks safe to assume that the argument to bar() is a constant 1. In the bad old days compiler writers had to make worst-case aliasing assumptions, to support lots of crazy wild west legacy code, and could not say that it was safe to assume the argument to bar was 1. They had to insert code to reload the value of anint for the call, because the intervening assignment through dblptr could have changed the value of anint if dblptr pointed to it. It's possible that the call to foowas foo((double *) &anint).

That's the problem that strict aliasing is intended to fix. There was low hanging fruit for compiler optimizer writers to pick and they wanted programmers to follow the aliasing rules so that they could pluck those fruit. Aliasing, and the problems it leads to, have been there as long as C has existed. The difference lately, is that compiler writers are being strict about the rules and enforcing them when optimization is in effect. In their respective standards, C and C++ include lists of the things that can legitimately alias, (see the next section), and in all other cases, compiler writers are allowed to assume no interactions between lvalues. Anything not on the list can be assumed to not alias, and compiler writers are free to do optimizations that make that assumption. For anything on the list, aliasing could possibly occur and compiler writers have to assume that it does. When compiler writers follow these lists, and assume that your code follows the rules, it's called strict-aliasing. Under strict aliasing, the compiler writer is free to optimize the function foo above because incompatible types,double and int, can't alias. That means that if you do call foo as foo((double *)&anint) something will go quickly wrong, but you get what you deserve.

So what can alias?

From C9899:201x 6.5 Expressions:

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

— a type compatible with the effective type of the object,

— a qualified version of a type compatible with the effective type of the object,

— a type that is the signed or unsigned type corresponding to the effective type of the object,

— a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or

— a character type.

These can be summarized as follows:

- Things that are compatible types or differ only by the addition of any combination of signed, unsigned, or volatile. For most purposes compatible type just means the same type. If you want more details you can read the specs. (Example: If you get a pointer to long, and a pointer to const unsigned long they could point to the same thing.)
- An aggregate (struct or class) or union type can alias types contained inside them. (Example: If a function gets passed a pointer to an int, and a pointer to a struct or union containing an int, or possibly containing another struct or unioncontaining an int, or containing...ad infinitum, it's possible that the int* points to an int contained inside the struct orunion pointed at by the other pointer.)
- A character type. A char*, signed char*, or unsigned char* is specifically allowed by the specs to point to anything. That means it can alias anything in memory.
- For C++ only, a possibly CV (const and/or volatile) qualified base class type of a dynamic type can alias the child type. (Example: if class dog has class animal for a base class, pointers or references to class dog and class animal can alias.)

Of course references have all these same issues and pointers and references can alias. Any lvalue has to be assumed to possibly alias to another lvalue if these rules say that they can alias. An aliasing issue is just as likely to come up with values passed by reference as it is with values passed as pointer to values. Additionally any combination of pointers and references have a possibility of aliasing, and you'd have to consult the aliasing rules to see if it might happen.

Part the second. How to do something the compiler doesn't like.

The following program swaps the halves of a 32 bit integer, and is typical of code you might use to handle data passed between a little-endian and big-endian machine. It also generates 6 warnings about breaking strict-aliasing rules. Many would dismiss them. The correct output of the program is:

00000020 00200000

but when optimization is turned on it's:

00000020 00000020

THAT's what the warning is trying to tell you, that the optimizer is going to do things that you don't like. Don't think this means that the optimizer broke your code. It's already broken. The optimizer just pointed it out for you.

Broken Version

```
uint32_t
swaphalves(uint32_t a)
{
    uint32_t acopy=a;
    uint16_t  *ptr=(uint16_t*)&acopy;//  can't  use  static_cast<>,  not
legal.
                                  // you should be warned by that.
    uint16_t tmp=ptr[0];
    ptr[0]=ptr[1];
    ptr[1]=tmp;
    return acopy;
}

int main()
{
    uint32_t a;
    a=32;
    cout << hex << setfill('0') << setw(8) << a << endl;
    a=swaphalves(a);
    cout << setw(8) << a << endl;
}
```

So what goes wrong? Since a uint16_t can't alias a uint32_t, under the rules, it's ignored in considering what to do withacopy. Since it sees that nothing is done with acopy inside the swaphalvesfunction, it just returns the original value of a. Here's the (annotated) x86 assembler generated by gcc 4.4.1 for swaphalves, let's see what went wrong:

```
_Z10swaphalvesj:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    8(%ebp), %eax   # get a in %eax
    movl    %eax, -8(%ebp)  # and store it in acopy
    leal    -8(%ebp), %eax  # now get eax pointing at acopy (ptr=&acopy)
    movl    %eax, -12(%ebp) # save that ptr at -12(%ebp)
    movl    -12(%ebp), %eax # get the ptr back in %eax
```

```
    movzwl  (%eax), %eax    # get 16 bits from ptr[0] in eax
    movw    %ax, -2(%ebp)   # store the 16 bits into tmp
    movl    -12(%ebp), %eax # get the ptr back in eax
    addl    $2, %eax        # bump up by two to get to ptr[1]
    movzwl  (%eax), %edx    # get that 16 bits into %edx
    movl    -12(%ebp), %eax # get ptr into eax
    movw    %dx, (%eax)     # store the 16 bits into ptr[1]
    movl    -12(%ebp), %eax # get the ptr again
    leal    2(%eax), %edx   # get the address of ptr[1] into edx
    movzwl  -2(%ebp), %eax  # get tmp into eax
    movw    %ax, (%edx)     # store into ptr[1]
    movl    -8(%ebp), %eax  # forget all that, return original a.
    leave
    ret
```

Scary, isn't it? Of course, if you are using gcc, you could use -fno-strict-aliasing to get the output you expect, but the generated code won't be as good, and you're just treating the symptom instead of curing the problem. A better way to accomplish the same thing without the warnings or the incorrect output is to define swaphalves like this. N.B. this is supported in C99 and later C specs, as noted in this footnote to 6.5.2.3 Structure and union members :

85. If the member used to access the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type as described in 6.2.6 (a process sometimes called ‘‘type punning’’). This might be a trap representation.

but your mileage may vary in C++. All C++ compilers that I know of support it, but the C++ spec doesn't allow it, so it would be risky to count on it. Right after this discussion I'll have another solution with memcpy that may be, (but probably isn't), slightly less efficient, and is supported by both C and C++):

Another Broken version, referencing a twice.

```
uint32_t
swaphalves(uint32_t a)
{
    a = (a >>= 16) | ( a <<= 16);
    return a;
}
```

This version looks reasonable, but you don't know if the right and left sides of the | will each get the original version of a or if one of them will get the result of the other. There's no sequence point here, so we don't know anything about the order of operations here,

and you may get different results from the same compiler using different levels of optimization.

Union version. Fixed for C but not guaranteed portable to C++.

```
uint32_t
swaphalves(uint32_t a)
{
    typedef                             union                               {
        uint32_t                                                          as32bit;
        uint16_t                                                        as16bit[2];
    } swapem;

    swapem s={a};
    uint16_t tmp;
    tmp=s.as16bit[0];
    s.as16bit[0]=s.as16bit[1];
    s.as16bit[1]=tmp;
    return s.as32bit;
}
```

The C++ compiler knows that members of a union fill the same memory, and this helps the compiler generate MUCH better code:

```
_Z10swaphalvesj:
    pushl   %ebp                # save the original value of ebp
    movl    %esp, %ebp          # point ebp at the stack frame
    movl    8(%ebp), %eax       # get a in eax
    popl    %ebp                # get the original ebp value back
    roll    $16, %eax             # swap the two halves of a and return
it
    ret
```

So do it wrong, via strange casts and get incorrect code, or by turning off strict-aliasing get inefficient code, or do it right and get efficient code.

You can also accomplish the same thing by using memcpy with char* to move the data around for the swap, and it will probably be as efficient. Wait, you ask me, how can that be? The will be at least two calls to memcpy added to the mix! Well gcc and other modern compilers have smart optimizers and will, in many cases, (including this one), elide the calls to memcpy. That makes it the most portable, and as efficient as any other method. Here's how it would look:

memcpy version, compliant to C and C++ specs and efficient

```
uint32_t
swaphalves(uint32_t a)
{
```

```
    uint16_t as16bit[2],tmp;

    memcpy(as16bit, &a, sizeof(a));
    tmp = as16bit[0];
    as16bit[0] = as16bit[1];
    as16bit[1] = tmp;
    memcpy(&a, as16bit, sizeof(a));
    return a;
}
```

For the above code, a C compiler will generate code similar to the previous solution, but with the addition of two calls to memcpy (possibly optimized out). gcc generates code identical to the previous solution. You can imagine other variants that substitute reading and writing through a char pointer locally for the calls to memcpy.

Similar issues arrive from networking code where you don't know what type of packet you have until you examine it. unions and/or memcpy are your friends here as well.

The restrict keyword

In C99 and later C Standards, but not in any C++ you can promise the compiler that a pointer to something is not aliased with the restrict qualifier keyword. In a situation where the compiler would have to expect that things could alias, you can tell the compiler that you promise it will not be so. So in this:

```
void foo(int * restrict i1, int * restrict i2);
```

you're telling the compiler that you promise that i1 and i2 will never point at the same memory. You have to know well the implementation of foo and only pass into it things that will keep the promise that things accessed through i1 and i2 will never alias. The compiler believes you and may be able to do a better job of optimization. If you break the promise your mileage may vary (and by that I mean that you will almost certainly cry).

Current C++ Standards specify that when C libraries are used from C++ the restrict qualifier shall be omitted. restrict is not a keyword for C++ and is not part of the C++ Standard in any version. Nonetheless, as pointed out by Ian Mallett, many compilers, as a non-standard extension, allow the use of __restrict__ or __restrict as qualifiers. Since restrict is not a keyword of C++ you can't use it directly, but in g++, clang, and MSVC you can do something like#define restrict __restrict and accomplish the same thing. In spite of this, they are still required by the C++ Standard to omit the qualifier from linked C libraries. Use at your own risk;)

What is the strict aliasing rule?

A typical situation you encounter strict aliasing problems is when overlaying a struct (like a device/network msg) onto a buffer of the word size of your system (like a pointer to uint32_ts or uint16_ts). When you overlay a struct onto such a buffer, or a buffer onto such a struct through pointer casting you can easily violate strict aliasing rules.

So in this kind of setup, if I want to send a message to something I'd have to have two incompatible pointers pointing to the same chunk of memory. I might then naively code something like this:

```
struct Msg
{
   unsigned int a;
   unsigned int b;
};

int main()
{
   // Get a 32-bit buffer from the system
   uint32_t* buff = malloc(sizeof(Msg));

   // Alias that buffer through message
   Msg* msg = (Msg*)(buff);

   // Send a bunch of messages
   for (int i =0; i < 10; ++i)
   {
      msg->a = i;
      msg->b = i+1;
      SendWord(buff[0] );
      SendWord(buff[1] );
   }
}
```

The strict aliasing rule makes this setup illegal: dereferencing a pointer that aliases another of an incompatible type is undefined behavior. Unfortunately, you can still code this way, maybe* get some warnings, have it compile fine, only to have weird unexpected behavior when you run the code.

*(gcc appears pretty inconsistent in its ability to give aliasing warnings, giving us a friendly warninghere but not here)

To see why this behavior is undefined, we have to think about what the strict aliasing rule buys the compiler. Basically, with this rule, it doesn't have to think about inserting instructions to refresh the contents of buff every run of the loop. Instead, when

optimizing, with some annoyingly unenforced assumptions about aliasing, it can omit those instructions, load buff[0] and buff[1] once before the loop is run, and speed up the body of the loop. Before strict aliasing was introduced, the compiler had to live in a state of paranoia that the contents of buff could change at anytime from anywhere by anybody. So to get an extra performance edge, and assuming most people don't type-pun pointers, the strict aliasing rule was introduced.

Keep in mind, if you think the example is contrived, this might even happen if you're passing a buffer to another function doing the sending for you, if instead you have.

```
void SendMessage(uint32_t* buff, size_t size32)
{
   for (int i = 0; i < size32; ++i)
   {
      SendWord(buff[i]);
   }
}
```

And rewrote our earlier loop to take advantage of this convenient function

```
  for (int i =0; i < 10; ++i)
  {
     msg->a = i;
     msg->b = i+1;
     SendMessage(buff, 2);
  }
```

The compiler may or may not be able to or smart enough to try to inline SendMessage and it may or may not decide to load or not load buff again. If SendMessage is part of another API that's compiled separately, it probably has instructions to load buff's contents. Then again, maybe you're in C++ and this is some templated header only implementation that the compiler thinks it can inline. Or maybe it's just something you wrote in your .c file for your own convenience. Anyway undefined behavior might still ensue. Even when we know some of what's happening under the hood, it's still a violation of the rule so no well defined behavior is guaranteed. So just by wrapping in a function that takes our word delimited buffer doesn't necessarily help.

So how do I get around this?

- Use a union. Most compilers support this without complaining about strict aliasing. This is allowed in C99 and explicitly allowed in C11.

```
   union {
       Msg msg;
       unsigned int asBuffer[sizeof(Msg)];
   };
```

- You can disable strict aliasing in your compiler (f[no-]strict-aliasing in gcc))
- You can use char* for aliasing instead of your system's word. The rules allow an exception for char* (including signed char and unsigned char). It's always assumed that char* aliases other types. However this won't work the other way: there's no assumption that your struct aliases a buffer of chars.

Beginner beware

This is only one potential minefield when overlaying two types onto each other. You should also learn about endianness, word alignment, and how to deal with alignment issues through packing structs correctly.

Default Functions

Returned values of printf()

In C, printf() returns the number of **characters** successfully written on the output and scanf() returns number of **items** successfully read.

Associativity of code given below

```
printf(" %d %d",printf(" geeks"),printf("geeksforgeeks"));
```

output is:

```
geeksforgeeks geeks 6 13
```

First of all start from the inner most printf statement. then move to the subsequent statements. So that is why in output "geeksforgeeks and geeks is printed now as %d is integer value so it will just count the character of each string" printf the first %d will count the letters in " geeks" including space which is 6 and the second %d will count the letters in the second one "geeksforgeeks" and both prinf will also do its job so we will have 4 outputs.

Scansets in C

Scanf family functions support scanset specifiers which are represented by %[]. Inside scanset, we can specify single character or range of characters. While processing scanset, scanf will process only those characters which are part of scanset. We can define scanset by putting characters inside squre brackets. Please note that the scansets are case-sensitive.

Let us see with example. Below example will store only capital letters to character array ‘str’, any other character will not be stored inside character array.

```
/* A simple scanset example */
#include <stdio.h>

int main(void)
{
    char str[128];
    printf("Enter a string: ");
    scanf("%[A-Z]s", str);
    printf("You entered: %s\n", str);
```

```
    return 0;
}
  [root@centos-6 C]# ./scan-set
  Enter a string: GEEKs_for_geeks
  You entered: GEEK
```

If first character of scanset is '^', then the specifier will stop reading after first occurence of that character. For example, given below scanset will read all characters but stops after first occurence of 'o'

```
scanf("%[^o]s", str);
```

Let us see with example.

```
/* Another scanset example with ^ */
#include <stdio.h>

int main(void)
{
    char str[128];
     printf("Enter a string: ");
    scanf("%[^o]s", str);
    printf("You entered: %s\n", str);

    return 0;
}
  [root@centos-6 C]# ./scan-set
  Enter a string: http://geeks for geeks
  You entered: http://geeks f
  [root@centos-6 C]#
```

Let us implement gets() function by using scan set. gets() fucntion reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF found.

```
/* implementation of gets() function using scanset */
#include <stdio.h>

int main(void)
{
    char str[128];
    printf("Enter a string with spaces: ");
    scanf("%[^\n]s", str);
    printf("You entered: %s\n", str);

    return 0;
}
  [root@centos-6 C]# ./gets
  Enter a string with spaces: Geeks For Geeks
  You entered: Geeks For Geeks
  [root@centos-6 C]#
```

As a side note, using gets() may not be a good idea in general. Check below note from Linux man page.

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

What is return type of getchar(), fgetc() and getc() ?

In C, return type of getchar(), fgetc() and getc() is int (not char). So it is recommended to assign the returned values of these functions to an integer type variable.

gets() is risky to use!

Consider the below program.

```
void read()
{
   char str[20];
   gets(str);
   printf("%s", str);
   return;
}
```

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.

```
#define MAX_LIMIT 20
void read()
{
   char str[MAX_LIMIT];
   fgets(str, MAX_LIMIT, stdin);
   printf("%s", str);

   getchar();
   return;
}
```

Use of getch(),getche() and getchar() in C

Most of the program is ending with getch(),and so we think that getch() is used to display the output...but it is wrong.It is used to get a single character from the console. Just see the behaviors of various single character input functions in c. getchar() getche() getch()

Function : getchar()

getchar() is used to get or read the input (i.e a single character) at run time.

Library:

Declaration:

```
int getchar(void);
```

Example Declaration:

```
char ch;
ch = getchar();
```

Return Value:

This function return the character read from the keyboard.

Example Program:

```
void main()
{
char ch;
ch = getchar();
printf("Input Char Is :%c",ch);
}
```

Program Explanation:

Here, declare the variable ch as char data type, and then get a value through getchar() library function and store it in the variable ch. And then, print the value of variable ch.

During the program execution, a single character is get or read through the getchar(). The given value is displayed on the screen and the compiler wait for another character

to be typed. If you press the enter key/any other characters and then only the given character is printed through the printf function.

Function : getche()

getche() is used to get a character from console, and echoes to the screen.

Library:

Declaration:

```
int getche(void);
```

Example Declaration:

```
char ch;
ch = getche();
```

Remarks:

getche reads a single character from the keyboard and echoes it to the current text window, using direct video or BIOS.

Return Value:

This function return the character read from the keyboard.

Example Program:

```
void main()
{
char ch;
ch = getche();
printf("Input Char Is :%c",ch);
}
```

Program Explanation:

Here, declare the variable ch as char data type, and then get a value through getche () library function and store it in the variable ch.And then, print the value of variable ch.

During the program execution, a single character is get or read through the getche(). The given value is displayed on the screen and the compiler does not wait for another character to be typed. Then, after wards the character is printed through the printf function.

Function : getch()

getch() is used to get a character from console but does not echo to the screen.
Library:

Declaration:

```
int getch(void);
```

Example Declaration:

```
char ch;
ch = getch(); (or ) getch();
```

Remarks:

getch reads a single character directly from the keyboard, without echoing to the screen.
Return Value:

This function return the character read from the keyboard.

Example Program:

```
void main()
{
char ch;
ch = getch();
printf("Input Char Is :%c",ch);
}
```

Program Explanation:

Here, declare the variable ch as char data type, and then get a value through getch() library function and store it in the variable ch. And then, print the value of variable ch.

During the program execution, a single character is get or read through the getch(). The given value is not displayed on the screen and the compiler does not wait for another character to be typed. And then, the given character is printed through the printf function.

puts() vs printf() for printing a string

In C, given a string variable str, which of the following two should be preferred to print it to stdout?

1) puts(str);

2) printf(str);

puts() can be preferred for printing a string because it is generally less expensive (implementation of puts() is generally simpler than printf()), and if the string has formatting characters like '%', then printf() would give unexpected results. Also, if str is a user input string, then use of printf() might cause security issues (see this for details).
Also note that puts() moves the cursor to next line. If you do not want the cursor to be moved to next line, then you can use following variation of puts().

```
   fputs(str, stdout)
```

You can try following programs for testing the above discussed differences between puts() and printf().

Program 1

```
#include<stdio.h>
int main()
{
    puts("Geeksfor");
    puts("Geeks");

    getchar();
    return 0;
}
```

Program 2

```
#include<stdio.h>
int main()
{
    fputs("Geeksfor", stdout);
    fputs("Geeks", stdout);

    getchar();
    return 0;
}
```

Program 3

```
#include<stdio.h>
int main()
{
    // % is intentionally put here to show side effects of using
printf(str)
    printf("Geek%sforGeek%s");
    getchar();
    return 0;
}
```

Program 4

```
#include<stdio.h>
int main()
{
    puts("Geek%sforGeek%s");
    getchar();
    return 0;
}
```

3. Strings, Pointers and Arrays

Strings and Pointers

In C, a string can be referred either using a character pointer or as a character array.

Strings as character arrays

```
char str[4] = "GfG"; /*One extra for string terminator*/
/*    OR    */
char str[4] = {'G', 'f', 'G', '\0'}; /* '\0' is string terminator */
```

When strings are declared as character arrays, they are stored like other types of arrays in C. For example, if str[] is an auto variable then string is stored in stack segment, if it's a global or static variable then stored in data segment, etc.

Strings using character pointers

1) Read only string in a shared segment. (Pointer to constant)

When string value is directly assigned to a pointer, in most of the compilers, it's stored in a read only block (generally in data segment) that is shared among functions.

```
char *str  =  "GfG";
```

In the above line "GfG" is stored in a shared read only location, but pointer str is stored in a read-write memory. You can change str to point something else but cannot change value at present str. So this kind of string should only be used when we don't want to modify string at a later stage in program, ie

```
*(str +1)= 'n'; // Leads to Segfault as char *str is pointer to constant
```

2) Dynamically allocated in heap segment.

Strings are stored like other dynamically allocated things in C and can be shared among functions.

```
char *str;
int size = 4; /*one extra for '\0'*/
str = (char *)malloc(sizeof(char)*size);
*(str+0) = 'G';
*(str+1) = 'f';
*(str+2) = 'G';
*(str+3) = '\0';
```

Let us see some examples to better understand above ways to store strings.

Example 1. (Try to modify string)
the below program may crash (gives segmentation fault error) because the line *(str+1) = ‘n’ tries to write a read only memory.\

```
int main()
{
 char *str;
 str = "GfG";     /* Stored in read only part of data segment */
 *(str+1) = 'n'; /* Problem:  trying to modify read only memory */
 getchar();
 return 0;
}
```

Below program works perfectly fine as str [] is stored in writable stack segment.

```
int main()
{
 char str[] = "GfG"; /* Stored in stack segment like other
                        auto variables */
 *(str+1) = 'n';     /* No problem: String is now GnG */
 getchar();
 return 0;
}
```

Below program also works perfectly fine as data at str is stored in writable heap segment.

```
int main()
{
  int size = 4;

  /* Stored in heap segment like other dynamically allocated things */
  char *str = (char *)malloc(sizeof(char)*size);
  *(str+0) = 'G';
  *(str+1) = 'f';
  *(str+2) = 'G';
  *(str+3) = '\0';
  *(str+1) = 'n';  /* No problem: String is now GnG */
   getchar();
   return 0;
}
```

Example 2 (Try to return string from a function)
The below program works perfectly fine as the string is stored in a shared segment and data stored remains there even after return of getString()

```
char *getString()
{
  char *str = "GfG"; /* Stored in read only part of shared segment */
```

```
  /* No problem: remains at address str after getString() returns*/
  return str;
}

int main()
{
  printf("%s", getString());
  getchar();
  return 0;
}
```

The below program also works perfectly fine as the string is stored in heap segment and data stored in heap segment persists even after return of getString()

```
char *getString()
{
  int size = 4;
  char  *str  =  (char  *)malloc(sizeof(char)*size);  /*Stored  in  heap
segment*/
  *(str+0) = 'G';
  *(str+1) = 'f';
  *(str+2) = 'G';
  *(str+3) = '\0';

  /* No problem: string remains at str after getString() returns */
  return str;
}
int main()
{
  printf("%s", getString());
  getchar();
  return 0;
}
```

But, the below program may print some garbage data as string is stored in stack frame of function getString() and data may not be there after getString() returns.

```
char *getString()
{
  char str[] = "GfG"; /* Stored in stack segment */

  /* Problem: string may not be present after getSting() returns */
  return str;
}
int main()
{
  printf("%s", getString());
  getchar();
  return 0;
}
```

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 (See this forum topic).

1) Pointer to variable.

```
int *ptr;
```

We can change the value of ptr and we can also change the value of object ptr pointing to. Pointer and value pointed by pointer both are stored in read-write area. See the following code fragment.

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    int j = 20;
    int *ptr = &i;        /* pointer to integer */
    printf("*ptr: %d\n", *ptr);

    /* pointer is pointing to another variable */
    ptr = &j;
    printf("*ptr: %d\n", *ptr);

    /* we can change value stored by pointer */
    *ptr = 100;
    printf("*ptr: %d\n", *ptr);

    return 0;
}
```

Output:

```
    *ptr: 10
    *ptr: 20
    *ptr: 100
```

2) Pointer to constant.

Pointer to constant can be declared in following two ways.

```
const int * ptr;
       Or
int const * ptr;
       or
char *ptr = "gfd";
```

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. Let us see following examples.

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    int j = 20;
    const int *ptr = &i;    /* ptr is pointer to constant */

    printf("ptr: %d\n", *ptr);
    *ptr = 100;        /* error: object pointed cannot be modified
                  using the pointer ptr */

    ptr = &j;          /* valid */
    printf("ptr: %d\n", *ptr);

    return 0;
}
```

Output:

```
 error: assignment of read-only location '*ptr'
```

Following is another example where variable i itself is constant.

```
#include <stdio.h>

int main(void)
{
    int const i = 10;    /* i is stored in read only area*/
    int j = 20;

    int const *ptr = &i;        /* pointer to integer constant. Here i
                                   is of type "const int", and &i is of
                                   type "const int *".
                                   And p is of type
                                  "const int", types are matching no issue */

    printf("ptr: %d\n", *ptr);

    *ptr = 100;        /* error */

    ptr = &j;          /* valid. We call it as up qualification. In
                          C/C++, the type of "int *" is allowed to up
                          qualify to the type "const int *". The type of
                          &j is "int *" and is implicitly up qualified by
                          the compiler to "cons tint *" */

    printf("ptr: %d\n", *ptr);

    return 0;
}
```

Output:

```
 error: assignment of read-only location '*ptr'
```

Down qualification is not allowed in C++ and may cause warnings in C. Following is another example with down qualification.

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int const j = 20;

    /* ptr is pointing an integer object */
    int *ptr = &i;

    printf("*ptr: %d\n", *ptr);

    /* The below assignment is invalid in C++, results in error
       In C, the compiler *may* throw a warning, but casting is
       implicitly allowed */
    ptr = &j;

    /* In C++, it is called 'down qualification'. The type of expression
       &j is "const int *" and the type of ptr is "int *". The
       assignment "ptr = &j" causes to implicitly remove const-ness
       from the expression &j. C++ being more type restrictive, will not
       allow implicit down qualification. However, C++ allows implicit
       up qualification. The reason being, const qualified identifiers
       are bound to be placed in read-only memory (but not always). If
       C++ allows above kind of assignment (ptr = &j), we can use 'ptr'
       to modify value of j which is in read-only memory. The
       consequences are implementation dependent, the program may fail
       at runtime. So strict type checking helps clean code. */

    printf("*ptr: %d\n", *ptr);

    return 0;
}
```

//Reference
//http://www.dansaks.com/articles/1999-02%20const%20T%20vs%20T%20const.pdf
// More interesting stuff on C/C++ @
http://www.dansaks.com/articles.htm

3) Constant pointer to variable.

```
int *const ptr;
char *const ptr = &ch;
```

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.

```
#include <stdio.h>

int main(void)
{
   int i = 10;
   int j = 20;
   int *const ptr = &i;    /* constant pointer to integer */

   printf("ptr: %d\n", *ptr);

   *ptr = 100;    /* valid */
   printf("ptr: %d\n", *ptr);

   ptr = &j;        /* error */
   return 0;
}
```

Output:

```
 error: assignment of read-only variable ‘ptr’
```

4) 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. Let us see with example.

```
#include <stdio.h>

int main(void)
{
    int i = 10;
    int j = 20;
    const int *const ptr = &i;/* constant pointer to constant integer */
    printf("ptr: %d\n", *ptr);

    ptr = &j;            /* error */
    *ptr = 100;        /* error */

    return 0;
}
```

Output:

```
    error: assignment of read-only variable ‘ptr’
    error: assignment of read-only location ‘*ptr’
```

C Pointer to Pointer, Pointer to Functions, Array of Pointers

In C programming language, the concept of pointers is the most powerful concept that makes C stand apart from other programming languages. The following are explained with examples:

1. Constant pointer and pointer to constant.
2. Pointer to pointer with an example
3. Array of pointers with an example
4. Pointer to functions with an example

1. Constant Pointer and Pointer to Constant

As a developer, you should understand the difference between constant pointer and pointer to constant.

Constant pointer

A pointer is said to be constant pointer when the address it's pointing to cannot be changed.

Let's take an example:

```
char ch, c;
char *ptr = &ch
ptr = &c
```

In the above example we defined two characters ('ch' and 'c') and a character pointer 'ptr'. First, the pointer 'ptr' contained the address of 'ch' and in the next line it contained the address of 'c'. In other words, we can say that initially 'ptr' pointed to 'ch' and then it pointed to 'c'.

But in case of a constant pointer, once a pointer holds an address, it cannot change it. This means a constant pointer, if already pointing to an address, cannot point to a new address.

If we see the example above, then if 'ptr' would have been a constant pointer, then the third line would have not been valid.

A constant pointer is declared as :

```
<type-of-pointer> *const <name-of-pointer>
```

For example:

```
#include<stdio.h>

int main(void)
{
    char ch = 'c';
    char c = 'a';

    char *const ptr = &ch; // A constant pointer
    ptr = &c; // Trying to assign new address to a constant pointer.
WRONG!!!!

    return 0;
}
```

When the code above is compiled, compiler gives the following error :

```
$ gcc -Wall constptr.c -o constptr
constptr.c: In function 'main':
constptr.c:9: error: assignment of read-only variable 'ptr'
```

So we see that, as expected, compiler throws an error since we tried to change the address held by constant pointer.

Now, we should be clear with this concept. Let's move on.

Pointer to Constant

This concept is easy to understand as the name simplifies the concept. Yes, as the name itself suggests, this type of pointer cannot change the value at the address pointed by it.

Let's understand this through an example:

```
char ch = 'c';
char *ptr = &ch
*ptr = 'a';
```

In the above example, we used a character pointer 'ptr' that points to character 'ch'. In the last line, we change the value at address pointer by 'ptr'. But if this would have been a pointer to a constant, then the last line would have been invalid because a pointer to a constant cannot change the value at the address its pointing to.

A pointer to a constant is declared as:

```
const <type-of-pointer> *<name-of-pointer>;
```

For example:

```
#include<stdio.h>

int main(void)
{
    char ch = 'c';
    const char *ptr = &ch; // A constant pointer 'ptr' pointing to
                           'ch'
    *ptr = 'a';// WRONG!!! Cannot change the value at address pointed
by 'ptr'.

    return 0;
}
```

When the above code was compiled, compiler gave the following error:

```
$ gcc -Wall ptr2const.c -o ptr2const
ptr2const.c: In function ‘main’:
ptr2const.c:7: error: assignment of read-only location ‘*ptr’
```

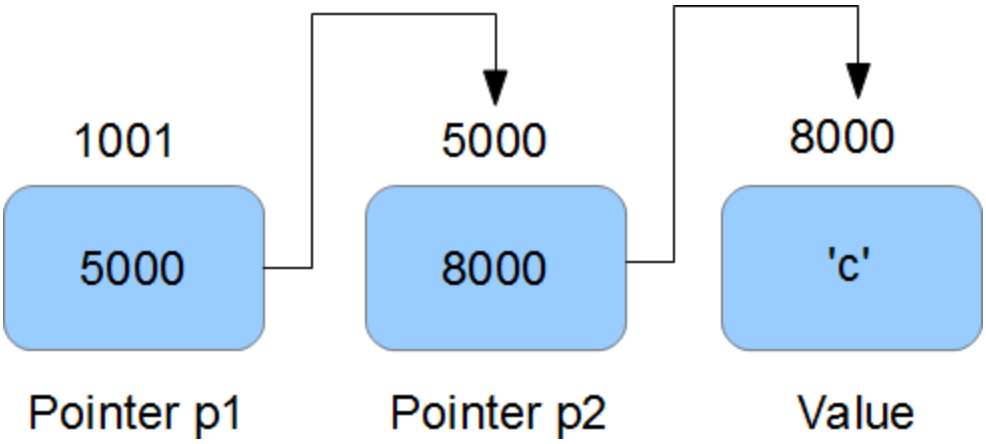
So now we know the reason behind the error above ie we cannot change the value pointed to by a constant pointer.

2. Pointer to Pointer

Till now we have used or learned pointer to a data type like character, integer etc. But in this section we will learn about pointers pointing to pointers.

As the definition of pointer says that it’s a special variable that can store the address of another variable. Then the other variable can very well be a pointer. This means that it’s perfectly legal for a pointer to be pointing to another pointer.

Let’s suppose we have a pointer ‘p1′ that points to yet another pointer ‘p2′ that points to a character ‘ch’. In memory, the three variables can be visualized as :



So we can see that in memory, pointer p1 holds the address of pointer p2. Pointer p2 holds the address of character ‘ch’.

So ‘p2′ is pointer to character ‘ch’, while ‘p1′ is pointer to ‘p2′ or we can also say that ‘p2′ is a pointer to pointer to character ‘ch’.

Now, in code ‘p2′ can be declared as :

```
char *p2 = &ch;
```

But ‘p1′ is declared as :

```
char **p1 = &p2;
```

So we see that ‘p1′ is a double pointer (ie pointer to a pointer to a character) and hence the two *s in declaration.

Now,

‘p1′ is the address of ‘p2′ ie 5000

‘*p1′ is the value held by ‘p2′ ie 8000

‘**p1′ is the value at 8000 ie ‘c’

I think that should pretty much clear the concept, let’s take a small example:

```
#include<stdio.h>

int main(void)
{
    char **ptr = NULL;
```

```
    char *p = NULL;

    char c = 'd';

    p = &c;
    ptr = &p;

    printf("\n c = [%c]\n",c);
    printf("\n *p = [%c]\n",*p);
    printf("\n **ptr = [%c]\n",**ptr);

    return 0;
}
```

Here is the output:

```
$ ./doubleptr

 c = [d]

 *p = [d]

 **ptr = [d]
```

3. Array of Pointers

Just like array of integers or characters, there can be array of pointers too.

An array of pointers can be declared as:

```
<type> *<name>[<number-of-elements];
```

For example:

```
char *ptr[3];
```

The above line declares an array of three character pointers.

Let’s take a working example:

```
#include<stdio.h>

int main(void)
{
    char *p1 = "Himanshu";
    char *p2 = "Arora";
    char *p3 = "India";

    char *arr[3];

    arr[0] = p1;
    arr[1] = p2;
```

```
    arr[2] = p3;

   printf("\n p1 = [%s] \n",p1);
   printf("\n p2 = [%s] \n",p2);
   printf("\n p3 = [%s] \n",p3);

   printf("\n arr[0] = [%s] \n",arr[0]);
   printf("\n arr[1] = [%s] \n",arr[1]);
   printf("\n arr[2] = [%s] \n",arr[2]);

   return 0;
}
```

In the above code, we took three pointers pointing to three strings. Then we declared an array that can contain three pointers. We assigned the pointers ‘p1′, ‘p2′ and ‘p3′ to the 0, 1 and 2 index of array. Let’s see the output:

```
$ ./arrayofptr

 p1 = [Himanshu]

 p2 = [Arora]

 p3 = [India]

 arr[0] = [Himanshu]

 arr[1] = [Arora]

 arr[2] = [India]
```

So we see that array now holds the address of strings.

4. Function Pointers

Just like pointer to characters, integers etc, we can have pointers to functions.
A function pointer can be declared as:

```
<return type of function> (*<name of pointer>) (type of function arguments)
```

For example:

```
int (*fptr)(int, int)
```

The above line declares a function pointer ‘fptr’ that can point to a function whose return type is ‘int’ and takes two integers as arguments.

Let’s take a working example:

```
#include<stdio.h>

int func (int a, int b)
{
    printf("\n a = %d\n",a);
    printf("\n b = %d\n",b);

    return 0;
}

int main(void)
{
    int(*fptr)(int,int); // Function pointer

    fptr = func; // Assign address to function pointer

    func(2,3);
    fptr(2,3);

    return 0;
}
```

In the above example, we defined a function ‘func’ that takes two integers as inputs and returns an integer. In the main() function, we declare a function pointer ‘fptr’ and then assign value to it. Note that, name of the function can be treated as starting address of the function so we can assign the address of function to function pointer using function’s name. Let’s see the output:

```
$ ./fptr

 a = 2

 b = 3

 a = 2

 b = 3
```

So from the output we see that calling the function through function pointer produces the same output as calling the function from its name.

Callback routines appear to be the most common scenario put forth thus far. However, there are many others ...

Finite State Machines where the elements of (multi-dimensional) arrays indicate the routine that processes/handles the next state. This keeps the definition of the FSM in one place (the array).

Enabling features and disabling of features can be done using function pointers. You may have features that you wish to enable or disable that do similar yet distinct things. Instead of populating and cluttering your code with if-else constructs testing variables, you can code it so that it uses a function pointer, and then you can enable/disable features by changing/assigning the function pointer. If you add new variants, you don't have to track down all your if-else or switch cases (and risk missing one); instead you just update your function pointer to enable the new feature, or disable the old one.

Reducing code clutter I touched upon this in the previous example. Examples such as...

```
switch (a) {
case 0:
    func0();
    break;
case 1:
    func1();
    break;
case 2:
    func2();
    break;
case 3:
    func3();
    break;
default:
    funcX();
    break;
}
```

Can be simplified to...

```
/* This declaration may be off a little, but I am after the essence of
the idea */
void (*funcArray)(void)[] = {func0, func1, func2, func3, funcX};
... appropriate bounds checking on 'a' ...
funcArray[a]();
```

C Constant Pointers and Pointer to Constants Examples

Pointers in C have always been a complex concept to understand for newbies. In this article, we will explain the difference between constant pointer, pointer to constant and constant pointer to constant.

1. Constant Pointers

Let's first understand what a constant pointer is. A constant pointer is a pointer that cannot change the address its holding. In other words, we can say that once a constant pointer points to a variable then it cannot point to any other variable.

A constant pointer is declared as follows:

```
<type of pointer> * const <name of pointer>
```

An example declaration would look like:

```
int * const ptr;
```

Let's take a small code to illustrate these types of pointers:

```
#include<stdio.h>

int main(void)
{
    int var1 = 0, var2 = 0;
    int *const ptr = &var1;
    ptr = &var2;
    printf("%d\n", *ptr);

    return 0;
}
```

In the above example:

- We declared two variables var1 and var2
- A constant pointer 'ptr' was declared and made to point var1
- Next, ptr is made to point var2.
- Finally, we try to print the value ptr is pointing to.

So, in a nutshell, we assigned an address to a constant pointer and then tried to change the address by assigning the address of some other variable to the same constant pointer.

Let's now compile the program:

```
$ gcc -Wall constptr.c -o constptr
```

```
constptr.c: In function ‘main’:
constptr.c:7: error: assignment of read-only variable ‘ptr’
```

So we see that while compiling the compiler complains about ‘ptr’ being a read only variable. This means that we cannot change the value ptr holds. Hence we conclude that a constant pointer which points to a variable cannot be made to point to any other variable.

2. Pointer to Constant

As evident from the name, a pointer through which one cannot change the value of variable it points is known as a pointer to constant. These type of pointers can change the address they point to but cannot change the value kept at those address.

A pointer to constant is defined as:

```
const <type of pointer>* <name of pointer>
```

An example of definition could be:

```
const int* ptr;
```

Let’s take a small code to illustrate a pointer to a constant:

```
#include<stdio.h>

int main(void)
{
    int var1 = 0;
    const int* ptr = &var1;
    *ptr = 1;
    printf("%d\n", *ptr);

    return 0;
}
```

In the code above:

- We defined a variable var1 with value 0
- we defined a pointer to a constant which points to variable var1
- Now, through this pointer we tried to change the value of var1
- Used printf to print the new value.

Now, when the above program is compiled:

```
$ gcc -Wall constptr.c -o constptr
constptr.c: In function ‘main’:
constptr.c:7: error: assignment of read-only location ‘*ptr’
```

So we see that the compiler complains about ‘*ptr’ being read-only. This means that we cannot change the value using pointer ‘ptr’ since it is defined a pointer to a constant.

3. Constant Pointer to a Constant

If you have understood the above two types then this one is very easy to understand as it’s a mixture of the above two types of pointers. A constant pointer to constant is a pointer that can neither change the address it’s pointing to and nor it can change the value kept at that address.

A constant pointer to constant is defined as:

```
const <type of pointer>* const <name of pointer>
```

for example :

```
const int* const ptr;
```

Let’s look at a piece of code to understand this:

```
#include<stdio.h>

int main(void)
{
    int var1 = 0,var2 = 0;
    const int* const ptr = &var1;
    *ptr = 1;
    ptr = &var2;
    printf("%d\n", *ptr);

    return 0;
}
```

In the code above:

- We declared two variables var1 and var2.
- We declared a constant pointer to a constant and made it to point to var1
- Now in the next two lines we tried to change the address and value pointed by the pointer.

When the code was compiled:

```
$ gcc -Wall constptr.c -o constptr
constptr.c: In function ‘main’:
constptr.c:7: error: assignment of read-only location ‘*ptr’
constptr.c:8: error: assignment of read-only variable ‘ptr’
```

So we see that the compiler complained about both the value and address being changed. Hence we conclude that a constant pointer to a constant cannot change the address and value pointed by it

C Arrays Basics

Here are times while writing C code, you may want to store multiple items of same type as contiguous bytes in memory so that searching and sorting of items becomes easy. For example:

1. Storing a string that contains series of characters. Like storing a name in memory.
2. Storing multiple strings. Like storing multiple names.

C programming language provides the concept of arrays to help you with these scenarios.

1. What is an Array?

An array is a collection of same type of elements which are sheltered under a common name.

An array can be visualized as a row in a table, whose each successive block can be thought of as memory bytes containing one element. Look at the figure below:

An Array of four elements:

```
+=================================================+
| elem1     | elem2      | elem3      | elem4      |
+=================================================+
```

The number of 8 bit bytes that each element occupies depends on the type of array. If type of array is ‘char’ then it means the array stores character elements. Since each character occupies one byte so elements of a character array occupy one byte each.

2. How to Define an Array?

An array is defined as following:

```
<type-of-array> <name-of-array> [<number of elements in array>];
```

- type-of-array: It is the type of elements that an array stores. If array stores character elements then type of array is ‘char’. If array stores integer elements then type of array is ‘int’. Besides these native types, if type of elements in array is structure objects then type of array becomes the structure.
- name-of-array: This is the name that is given to array. It can be any string but it is usually suggested that some can of standard should be followed while naming arrays. At least the name should be in context with what is being stored in the array.
- [number of elements]: This value in subscripts [] indicates the number of elements the array stores.

For example, an array of five characters can be defined as:

```
char arr[5];
```

3. How to Initialize an Array?

An array can be initialized in many ways as shown in the code-snippets below.

i. Initializing each element separately.

For example:

```
int arr[10];
int i = 0;
for(i=0;i<sizeof(arr);i++)
{
  arr[i] = i; // Initializing each element seperately
}
```

ii. Initializing array at the time of declaration

For example:

```
int arr[] = {'1','2','3','4','5'};
```

In the above example an array of five integers is declared. Note that since we are initializing at the time of declaration so there is no need to mention any value in the subscripts []. The size will automatically be calculated from the number of values. In this case, the size will be 5.

iii. Initializing array with a string (Method 1):

Strings in C language are nothing but a series of characters followed by a null byte. So to store a string, we need an array of characters followed by a null byte. This makes the initialization of strings a bit different. Let us take a look:

Since strings are nothing but a series of characters so the array containing a string will be containing characters

```
char arr[] = {'c','o','d','e','\0'};
```

In the above declaration/initialization, we have initialized array with a series of character followed by a '\0′ (null) byte. The null byte is required as a terminating byte when string is read as a whole.

iv. Initializing array with a string (Method 2):

```
char arr[] = "code";
```

Here we neither require explicitly wrapping single quotes around each character nor writing a null character. The double quotes do the trick for us.

4. Accessing Values in an Array

Now we know how to declare and initialize an array. Let's understand how to access array elements. An array element is accessed as:

```
int arr[10];
int i = 0;
for(i=0;i<sizeof(arr);i++)
{
  arr[i] = i; // Initializing each element separately
}
int j = arr[5]; // Accessing the 6th element of integer array arr and
assigning its value to integer 'j'.
```

As we can see above, the 6th element of array is accessed as 'arr[5]'.

Note that for an array declared as int arr[5]. The five values are represented as: arr[0] arr[1] arr[2] arr[3] arr[4] and not arr[1] arr[2] arr[3] arr[4] arr[5]

The first element of array always has a subscript of '0′

5. Array of Structures

The following program gives a brief idea of how to declare, initialize and use array of structures.

```
#include<stdio.h>

struct st{
    int a;
    char c;
};
```

```
int main()
{
    struct st st_arr[3]; // Declare an array of 3 structure objects

    struct st st_obj0; // first structure object
    st_obj0.a = 0;
    st_obj0.c = 'a';

    struct st st_obj1; //Second structure object
    st_obj1.a = 1;
    st_obj1.c = 'b';

    struct st st_obj2; // Third structure object
    st_obj2.a = 2;
    st_obj2.c = 'c';

    st_arr[0] = st_obj0; // Initializing first element of array with
                            First structure object
    st_arr[1] = st_obj1; // Initializing second element of array with
                            Second structure object
    st_arr[2] = st_obj2; // Initializing third element of array with
                            third structure object

    printf("\n First Element of array has values of a = [%d] and c =
[%c]\n", st_arr[0].a, st_arr[0].c);
    printf("\n Second Element of array has values of a = [%d] and c =
[%c]\n", st_arr[1].a, st_arr[1].c);
    printf("\n Third Element of array has values of a = [%d] and c =
[%c]\n", st_arr[2].a, st_arr[2].c);

    return 0;
}
```

The output of the above program comes out to be:

```
$ ./strucarr

 First Element of array has values of a = [0] and c = [a]

 Second Element of array has values of a = [1] and c = [b]

 Third Element of array has values of a = [2] and c = [c]
```

6. Array of Char Pointers

The following program gives a brief idea of how to declare an array of char pointers:

```
#include<stdio.h>

int main()
{
    // Declaring/Initializing three characters pointers
```

```
    char *ptr1 = "Himanshu";
    char *ptr2 = "Arora";
    char *ptr3 = "TheGeekStuff";

    //Declaring an array of 3 char pointers
    char* arr[3];

    // Initializing the array with values
    arr[0] = ptr1;
    arr[1] = ptr2;
    arr[2] = ptr3;

    //Printing the values stored in array
    printf("\n [%s]\n", arr[0]);
    printf("\n [%s]\n", arr[1]);
    printf("\n [%s]\n", arr[2]);

    return 0;
}
```

The output of the above program is:

```
$ ./charptrarr

 [Himanshu]

 [Arora]

 [TheGeekStuff]
```

7. Pointer to Arrays

Pointers in C Programming language are very powerful. Combining pointers with arrays can be very helpful in certain situations.

As to any kind of data type, we can have pointers to arrays also. A pointer to array is declared as:

```
<data type> (*<name of ptr>)[<an integer>]
```

For example:

```
int (*ptr)[5];
```

The above example declares a pointer ptr to an array of 5 integers.

Let’s look at a small program for demonstrating this:

```
#include<stdio.h>

int main(void)
```

```
{
    char arr[3];
    char(*ptr)[3];

    arr[0] = 'a';
    arr[1] = 'b';
    arr[2] = 'c';

    ptr = &arr;

    return 0;
}
```

In the above program, we declared and initialized an array ‘arr’ and then declared a pointer ‘ptr’ to an array of 3 characters. Then we initialized ptr with the address of array ‘arr’.

8. Static vs. Dynamic Arrays

Static arrays are the ones that reside on stack. Like:

```
char arr[10];
```

Dynamic arrays is a popular name given to a series of bytes allocated on heap. this is achieved through malloc() function. Like :

```
char *ptr = (char*)malloc(10);
```

The above line allocates a memory of 10 bytes on heap and we have taken the starting address of this series of bytes in a character pointer ptr.

Static arrays are used when we know the amount of bytes in array at compile time while the dynamic array is used where we come to know about the size on run time.

9. Decomposing Array into Pointers

Internally, arrays aren’t treated specially, they are decomposed into pointers and operated there-on. For example an array like :

```
char arr[10];
```

When accessed like :

```
arr[4] = 'e';
```

is decomposed as :

```
*(arr + 4) = 'e'
```

So we see above that the same old pointers techniques are used while accessing array elements.

10. Character Arrays and Strings

Mostly new programmers get confused between character arrays and strings. Well, there is a very thin line between the two. This thin line only comprises of a null character '\0′. If this is present after a series of characters in an array, then that array becomes a string.

This is an array:

```
char arr[] = {'a', 'b', 'c'};
```

This is a string:

```
char arr[] = {'a', 'b', 'c', '\0'};
```

Note : A string can be printed through %s format specifier in printf() while an printing an array through %s specifier in printf() is a wrong practice.

11. Bi-dimensional and Multi-dimensional Arrays

The type of array we discussed until now is single dimensional arrays. As we see earlier, we can store a set of characters or a string in a single dimensional array. What if we want to store multiple strings in an array? Well, that won't be possible using single dimensional arrays. We need to use bi-dimensional arrays in this case. Something like:

```
char arr[5][10];
```

The above declaration can be thought of as 5 rows and 10 columns. Where each row may contain a different name and columns may limit the number of characters in the name. So we can store 5 different names with max length of 10 characters each. Similarly, what if we want to store different names and their corresponding addresses also. Well this requirement cannot be catered even by bi-dimensional arrays. In this case we need tri-dimensional (or multi-dimensional in general) arrays. So we need something like :

```
char arr[5][10][50];
```

So we can have 5 names with max capacity of 10 characters for names and 50 characters for corresponding addresses.

Since this is an advanced topic, so we won't go into practical details here.

12. A Simple C Program using Arrays

Consider this simple program that copies a string into an array and then changes one of its characters:

```
#include<stdio.h>
#include<string.h>

int main(void)
{
    char arr[4];// for accommodating 3 characters and one null '\0' byte.
    char *ptr = "abc"; //a string containing 'a', 'b', 'c', '\0'

    memset(arr, '\0', sizeof(arr)); //reset all the bytes so that none
of the byte contains any junk value
    strncpy(arr,ptr,sizeof("abc")); // Copy the string "abc" into the
array arr

    printf("\n %s \n",arr); //print the array as string

    arr[0] = 'p'; // change the first character in the array

    printf("\n %s \n",arr);//again print the array as string
    return 0;
}
```

I think the program is self-explanatory as I have added plenty of comments. The output of the above program is:

```
$ ./array_pointer

 abc

 pbc
```

So we see that we successfully copied the string into array and then changed the first character in the array.

13. No Array Bound Check in a C Program

What is array bound check? Well this is the check for boundaries of array declared. For example:

```
char arr[5];
```

The above array ‘arr’ consumes 5 bytes on stack and through code we can access these bytes using:

```
arr[0], arr[1], arr[2], arr[3], arr[4]
```

Now, C provides open power to the programmer to write any index value in [] of an array. This is where we say that no array bound check is there in C. SO, misusing this power, we can access arr[-1] and also arr[6] or any other illegal location. Since these bytes are on stack, so by doing this we end up messing with other variables on stack. Consider the following example:

```
#include<stdio.h>

unsigned int count = 1;

int main(void)
{
    int b = 10;
    int a[3];
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;

    printf("\n b = %d \n",b);
    a[3] = 12;
    printf("\n b = %d \n",b);

    return 0;
}
```

In the above example, we have declared an array of 3 integers but try to access the location arr[3] (which is illegal but doable in C) and change the value kept there.

But, we end up messing with the value of variable ‘b’. Can’t believe it? Check the following output . We see that value of b changes from 10 to 12.

```
$ ./stk
 b = 10
 b = 12
```

Difference between pointer and array in C

Pointers are used for storing address of dynamically allocated arrays and for arrays which are passed as arguments to functions. In other contexts, arrays and pointer are two different things; see the following programs to justify this statement.

Behavior of sizeof operator

```
// 1st program to show that array and pointers are different
#include <stdio.h>
int main()
{
   int arr[] = {10, 20, 30, 40, 50, 60};
   int *ptr = arr;

   // sizof(int) * (number of element in arr[]) is printed
   printf("Size of arr[] %d\n", sizeof(arr));

   // sizeof a pointer is printed which is same for all type
   // of pointers (char *, void *, etc)
   printf("Size of ptr %d", sizeof(ptr));
   return 0;
}
```

Output:

```
Size of arr[] 24
Size of ptr 4
```

Assigning any address to an array variable is not allowed.

```
// IInd program to show that array and pointers are different
#include <stdio.h>
int main()
{
   int arr[] = {10, 20}, x = 10;
   int *ptr = &x; // This is fine
   arr = &x;  // Compiler Error
   return 0;
}
```

Output:

```
 Compiler Error: incompatible types when assigning to
                 type 'int[2]' from type 'int *'
```

Although array and pointer are different things, following properties of array make them look similar.

1) *Array name gives address of first element of array.*

Consider the following program for example.

```
#include <stdio.h>
int main()
{
   int arr[] = {10, 20, 30, 40, 50, 60};
   int *ptr = arr;  // Assigns address of array to ptr
   printf("Value of first element is %d", *ptr)
   return 0;
}
```

Output:

```
Value of first element is 10
```

2) *Array members are accessed using pointer arithmetic.*

Compiler uses pointer arithmetic to access array element. For example, an expression like “arr[i]” is treated as *(arr + i) by the compiler. That is why the expressions like *(arr + i) work for array arr, and expressions like ptr[i] also work for pointer ptr.

```
#include <stdio.h>
int main()
{
   int arr[] = {10, 20, 30, 40, 50, 60};
   int *ptr = arr;
   printf("arr[2] = %d\n", arr[2]);
   printf("*(ptr + 2) = %d\n", *(arr + 2));
   printf("ptr[2] = %d\n", ptr[2]);
   printf("*(ptr + 2) = %d\n", *(ptr + 2));
   return 0;
}
```

Output:

```
arr[2] = 30
*(ptr + 2) = 30
ptr[2] = 30
*(ptr + 2) = 30
```

3) *Array parameters are always passed as pointers, even when we use square brackets.*

```
#include <stdio.h>
```

```
int fun(int ptr[])
{
   int x = 10;

   // size of a pointer is printed
   printf("sizeof(ptr) = %d\n", sizeof(ptr));

   // This allowed because ptr is a pointer, not array
   ptr = &x;

   printf("*ptr = %d ", *ptr);

   return 0;
}
int main()
{
   int arr[] = {10, 20, 30, 40, 50, 60};
   fun(arr);
   return 0;
}
```

Output:

```
sizeof(ptr) = 4
*ptr = 10
```

Pointer and I-D Array

Consider the given array

```
int ages = {24, 65, 75, 27};
char *names = {“ram”, “sam”, “peter”, “jade”};
```

We can print the array in following ways:

```
1.
 int count = sizeof(ages) / sizeof(int);
 int i = 0;
 for(i = 0; i < count; i++) {
     printf("%s has %d years alive.\n",
             names[i], ages[i]);
 }
2.
 int *cur_age = ages;
 char **cur_name = names;
 for(i = 0; i < count; i++) {
     printf("%s is %d years old.\n",
             *(cur_name+i), *(cur_age+i));
 }
3.
    for(i = 0; i < count; i++) {
     printf("%s is %d years old again.\n",
             cur_name[i], cur_age[i]);
    }
4.
     for(i = 0; i < count; i++) {
     printf("%s is %d years old again.\n",
             *cur_name, *cur_age);
     cur_name++;
     cur_age++;
 }
```

Dynamically allocate a 2D array in C?

Following are different ways to create a 2D array on heap (or dynamically allocate a 2D array). In the following examples, we have considered '**r**' as number of rows, '**c**' as number of columns and we created a 2D array with r = 3, c = 4 and following values

1 2 3 4

5 6 7 8

1 10 11 12

1) Using a single pointer:

A simple way is to allocate memory block of size r*c and access elements using simple pointer arithmetic.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int r = 3, c = 4;
    int *arr = (int *)malloc(r * c * sizeof(int));

    int i, j, count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         *(arr + i*c + j) = ++count;

    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         printf("%d ", *(arr + i*c + j));

   /* Code for further processing and free the
      dynamically allocated memory */

   return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

2) Using an array of pointers.

We can create an array of pointers of size r. Note that from C99; C language allows variable sized arrays. After creating an array of pointers, we can dynamically allocate memory for every row.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int r = 3, c = 4, i, j, count;

    int *arr[r];
    for (i=0; i<r; i++)
         arr[i] = (int *)malloc(c * sizeof(int));

    // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         arr[i][j] = ++count; // Or *(*(arr+i)+j) = ++count

    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         printf("%d ", arr[i][j]);

    /* Code for further processing and free the
      dynamically allocated memory */

   return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

3) Using pointer to a pointer.

We can create an array of pointers also dynamically using a double pointer. Once we have an array pointers allocated dynamically, we can dynamically allocate memory and for every row like method 2.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int r = 3, c = 4, i, j, count;

    int **arr = (int **)malloc(r * sizeof(int *));
    for (i=0; i<r; i++)
         arr[i] = (int *)malloc(c * sizeof(int));

    // Note that arr[i][j] is same as *(*(arr+i)+j)
    count = 0;
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         arr[i][j] = ++count;  // OR *(*(arr+i)+j) = ++count
```

```
    for (i = 0; i <  r; i++)
      for (j = 0; j < c; j++)
         printf("%d ", arr[i][j]);

   /* Code for further processing and free the
      dynamically allocated memory */

   return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

Return a string from Function

Method 1

```
#include <stdio.h>
char *func()
{
        return "string return";
}
int main ()
{
       char *str =func();
       printf("the string is %s\n",str);
       printf("the string is %s\n",func());
       return 0;
}
```

Method 2

Declaring the char array globally;

```
#include <stdio.h>
char global_string[100];
int func()
{
        int i;
        for (i=0;i<9;i++)
        global_string[i] = 'a';
        global_string[i] = '\0';

        return 0;
}
int main (){
func();
printf("The global string is %s",global_string);
return 0;
}
```

Method 3

Use of static array;

```
char *fun()
{
       static char static_arr[100];
       ....
       return static_arr;
}
```

Return a 2-d Array

```
int **myFunc(int n) {
    int i, j, **x;
    n=6;

    // allocate block of memory
    int **x = (int **) malloc(n * sizeof(int*));


    // allocate block of memory
    for(i=0; i<6; i++){
        x[i] = (int* ) malloc( sizeof(int)*6 );


         for(j=0; j<6; j++){
             printf("A: ");
             scanf(" %d", &x[i][j]);
         }
    }

    // return the pointer
    return x;
}
```

The C language has a basic flaw: it is impossible to return arrays from functions. There are many workarounds for this; i'll describe three.

Replace by a pointer to an array

Return a pointer instead of an array itself. This leads to another problem in C: when a function returns a pointer to something, it should usually allocate the something dynamically. You should not forget to deallocate this later (when the array is not needed anymore).

```
typedef int (*pointer_to_array)[6][6];

pointer_to_array workaround1()
{
    pointer_to_array result = malloc(sizeof(*result));
    (*result)[0][0] = 0;
    (*result)[1][0] = 0;
    (*result)[2][0] = 0;
    (*result)[3][0] = 0;
    (*result)[4][0] = 0;
    (*result)[5][0] = 0;
    return result;
}
```

Replace by a pointer to int

A 2-D array appears just as a sequence of numbers in memory, so you can replace it by a pointer to first element. You clearly stated that you want to return an array, but your example code returns a pointer to int, so maybe you can change the rest of your code accordingly.

```
int *workaround2()
{
    int temp[6][6] = {{0}}; // initializes a temporary array to zeros
    int *result = malloc(sizeof(int) * 6 * 6); // allocates a one-dimensional
array
    memcpy(result, temp, sizeof(int) * 6 * 6); // copies stuff
    return result; // cannot return an array but can return a pointer!
}
```

Wrap with a structure

It sounds silly, but functions can return structures even though they cannot return arrays! Even if the returned structure contains an array.

```
struct array_inside
{
    int array[6][6];
};

struct array_inside workaround3()
{
    struct array_inside result = {{{0}}};
    return result;
}
```

IV. Snap Shots

Reasons For Segmentation Fault In C

There are times when you write a small or a big code and when you execute it you get a very small and precise output 'Segmentation fault'. In a small piece of code its still easy to debug the reason for this but as the code size grows it becomes very difficult to debug. Following are some example scenarios which will demonstrate some reasons because of which a segmentation fault can occur.

Meaning of Segmentation Fault

Before jumping on to the actual scenarios, lets quickly discuss what does Segmentation Fault means?

A segmentation fault occurs mainly when our code tries to access some memory location which it is not suppose to access.

For example :

1) Working on a dangling pointer.
2) Writing past the allocated area on heap.
3) Operating on an array without boundary checks.
4) Freeing a memory twice.
5) Working on Returned address of a local variable
6) Running out of memory(stack or heap)

1) Working on a dangling pointer.

What is dangling pointers? A pointer which holds memory address of a memory which is already freed is known as a dangling pointer. You cannot figure out whether a given pointer is dangling or not until you use it. When a dangling pointer is used, usually a segmentation fault is observed.

Now, lets look at a code to understand it :

Code:

```
#include<stdio.h>
#include<stdlib.h>

int main(void)
```

```
{
    char *p = malloc(3);

    *p = 'a';
    *(p+1) = 'b';
    *(p+2) = 'c';

    free(p);

    *p = 'a';

    return 0;
}
```

In the code above, we have malloc'd 3 bytes on heap and stored the address of first byte in a pointer 'p'. Next we initialized these three bytes. Next we freed this memory and after that we are trying to use this memory again. Well this is not permitted as once a memory is freed, it no longer belongs to our process. Though, if you run the above code, it may not give a segmentation fault immediately as free() returns the memory to heap and now its up to the implementation of heap to take it back to its pool. Once its taken back to heap by kernel then the code above will start giving segmentation faults. Lets take another example :

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(void)
{
    char *p ;

    strcat(p, "abc");
    printf("\n %s \n", p);

    return 0;
}
```

In the code above, we have pointer 'p', to which we have not allocated any memory. Now we use the garbage address held by the pointer 'p' in the function 'strcat()'. So in the implementation of strcat(), whenever 'p' is accessed, it will give a segmentation fault.
A yet another example could be :

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>


void func(char ** argv)
{
    char arr[2];
    strcpy(arr, argv[1]);

    return;
}

int main(int argc, char *argv[])
{
    func(argv);
    return 0;
}
```

In the above code, we try to access the second argument from command line in the function func() without even checking whether the user has even provided the second argument or not. If the user did not provide then argv[1] will point to a location that our code does not have access to. Hence, in that case we will definitely get a segmentation fault.

2)Writing past the allocated area on heap.

There are times when a logic inadvertently writes past the allocated area on heap. This may happen while performing some operations in a loop or not doing array bound checks etc. So this type of situation also results in a segmentation fault. For example, look at the following code :

Code:

```
#include<stdio.h>
#include<stdlib.h>

int main(void)
{
    char *p = malloc(3);

    int i = 0;

    for(i=0;i<0Xffffffff;i++)
    {
        p[i] = 'a';
    }

    printf("\n %s \n", p);

    return 0;
```

```
}
```

In the example above, we allocate some bytes to pointer 'p' but try to write way past these bytes in a loop. So, the result we get is a segmentation fault.

3) Operating on an array without boundary checks.

In this scenario, the logic is flawed in a way that an array is written out of its boundary limits and in a rare scenario (or in case of exploits), this buffer overflow may result in overwriting the return address(ie the address to return after executing the present function). And hence returning on a garbage address and executing the instruction kept there may very well cause segmentation fault.

Lets look the following code :

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>


void func(char ** argv)
{
    char arr[2];
    strcpy(arr, argv[1]);

    return;
}

int main(int argc, char *argv[])
{
    func(argv);
    return 0;
}
```

In the code above, we are passing the command line argument array to function func(). Inside the function func(), we try to copy the second command line argument (with index '1') into the array arr. The problem here is the function we use to copy. We use strcpy() which has no concern with the capacity of array arr. This function will not detect or prevent a buffer overflow. So if we try to enter very huge string through this logic presented above, we will definitely overwrite the return address kept in the stack of this function and will cause a segmentation fault to happen.

Here is the output of the code above(I tried to run it twice with different command line

args) :
Code:

```
 ~/practice $ ./segfault abc
  ~/practice $
```

Code:

```
~/practice $ ./segfault
abcjflcnmscn,snlkewfdebddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddd
 Segmentation fault
```

As can be clearly seen, in the first attempt, the code worked fine but in the second attempt, a large command line argument probably overwrote the return address stored on stack of the function func() and hence when the control went back to this overwritten value any damn thing could have caused a segmentation fault as this memory location mostly(until you are very lucky) does not belong to our process.

4) Freeing a memory twice.

This is a bit specific to function free() but is a very common reason for segmentation faults to occur. The specification of free() specifies that if this function is used again on an already freed pointer, the results are undefined and mostly we see a segmentation fault in this scenario.

Lets quickly see the code :

Code:

```
 #include<stdio.h>
 #include<stdlib.h>
 #include<string.h>

 int main(int argc, char *argv[])
 {
     char *p = malloc(8);


     free(p);
     free(p);
```

```
    return 0;
}
```

As clearly seen in the code above, we have allocated memory once but we have freed it back to back twice. This is wrong practice and should be avoided.

5) Working on Returned address of a local variable

I this scenario, the address of a local variable is returned to calling function and this address is used there. For example, consider the following code :

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>


char* func()
{
    char c = 'a';
    return &c;
}

int main(int argc, char *argv[])
{
    char *ptr = func();
    char arr[10];
    memset(arr,'0', sizeof(arr));
    arr[0] = *ptr;
    return 0;
}
```

In the code above, we did exactly the same now since we all know that the stack of the function is unwind-ed when func() returns. So, its fatal to use any address out of a function stack that has already been unwind-ed. This may cause a segmentation fault.

6) Running out of memory.

It may happen that we run out of memory. This memory can be a stack memory or a heap memory. Lets consider the following two examples to understand this :

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main()
```

```
{
    main();
}
```

If you execute the code above, it continuously calls on main() recursively. So with every call, a stack is formed and none of these stacks is ever unwind-ed as we never stop calling main. So at a point all of the process's stack memory gets eaten up and then we get segmentation fault.

Consider example of heap memory :

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>


int main()
{
   unsigned int i = 0;

   for(i=0; i< 0xFFFFFFFF;i++);
        char *p = malloc(100);

   return 0;
}
```

The code above may cause you program to segment fault once all the heap memory is consumed.
Conclusion

To conclude, in this article we studied the different ways in which one could screw up his/her program and get segmentation fault. This article should act as a good resource for those who are getting this error but not able to understand the reason.

How to Define C Macros (C Example Using #define and #ifdef)

Sometimes while programming, we stumble upon a condition where we want to use a value or a small piece of code many times in a code. Also there is a possibility that the in future, the piece of code or value would change. Then changing the value all over the code does not make any sense. There has to be a way out through which one can make

the change at one place and it would get reflected at all the places. This is where the concept of a macro fits in.

A Macro is typically an abbreviated name given to a piece of code or a value. Macros can also be defined without any value or piece of code but in that case they are used only for testing purpose.

Let's understand the concept of macros using some example codes.

Defining Macros without values

The most basic use of macros is to define them without values and use them as testing conditions. As an example, let's look at the following piece of code :

```
#include <stdio.h>

#define MACRO1
#define MACRO2

int main(void)
{
#ifdef MACRO1 // test whether MACRO1 is defined...
    printf("\nMACRO1 Defined\n");
#endif

#ifdef MACRO2 // test whether MACRO2 is defined...
    printf("\nMACRO2 Defined\n");
#endif
    return 0;
}
```

So, the above code just defines two macros MACRO1 and MACRO2.

As clear from the definition, the macros are without any values

Inside the main function, the macros are used only in testing conditions.

Now, if we look at the output, we will see:

```
$ ./macro

MACRO1 Defined

MACRO2 Defined
```

Since both of the macros are defined so both the printf statements executed.

Now, one would question where these testing macros are used. Well, mostly these types of testing macros are used in a big project involving many source and header files. In such big projects, to avoid including a single header more than once (directly and indirectly through another header file) a macro is defined in the original header and this macro is tested before including the header anywhere so as to be sure that if the macros is already defined then there is no need to include the header as it has already been included (directly or indirectly).

Defining Macros through command line

Another use of testing macros is where we want to enable debugging (or any other feature) in a code while compilation. In this case, a macro can be defined through compilation statement from command line. This definition of macro is reflected inside the code and accordingly the code is compiled.

As an example, I modified the code used in example of the last section in this way :

```
#include <stdio.h>

#define MACRO1

int main(void)
{
#ifdef MACRO1 // test whether MACRO1 is defined...
    printf("\nMACRO1 Defined\n");
#endif

#ifdef MACRO2 // test whether MACRO2 is defined...
    printf("\nMACRO2 Defined\n");
#endif
return 0;
}
```

- So now only MACRO1 is defined
- While MACRO2 is also being used under a condition.

If the above program is now compiled and run, we can see the following output :

```
$ ./macro

MACRO1 Defined
```

So we see that since only MACRO1 is defined so condition related to MACRO1 executed. Now, if we want to enable or define MACRO2 also then either we can do it

from within the code (as shown in first example) or we can define it through the command line. The command for compilation of the code in that case becomes :

```
$ gcc -Wall -DMACRO2 macro.c -o macro
```

and now if we run the code, the output is :

```
$ ./macro

MACRO1 Defined

MACRO2 Defined
```

So we see that MACRO2 got defined and hence the printf under the MACRO2 condition got executed.

Macros with values

As discussed in the introduction, there are macros that have some values associated with them. For example :

```
#define MACRO1 25
```

So, in the above example, we defined a macro MACRO1 which has value 25. The concept is that in the preprocessing stage of the compilation process, the name of this macro is replaced with macros value all over the code. For example :

```
#include <stdio.h>

#define MACRO1 25

int main(void)
{
#ifdef MACRO1 // test whether MACRO1 is defined...
    printf("\nMACRO1 Defined with value [%d]\n", MACRO1);
#endif

    return 0;
}
```

So in the code above, a value of 25 is given to the macro MACRO1. When the code above is run, we see the following output :

```
$ ./macro

MACRO1 Defined with value [25]
```

So we see that the macro name (MACRO1) was replaced by 25 in the code.

Defining macros with values from command line

Not only the macros can be defined from command line (as shown in one of the sections above) but also they can be given values from command line. Lets take the following example :

```
#include <stdio.h>

int main(void)
{
#ifdef MACRO1 // test whether MACRO1 is defined...
    printf("\nMACRO1 Defined with value [%d]\n", MACRO1);
#endif

    return 0;
}
```

In the code above, the macro MACRO1 is being tested and its value is being used but it is not defined anywhere. Lets define it from the command line :

```
$ gcc -Wall -DMACRO1=25 macro.c -o macro
$ ./macro

MACRO1 Defined with value [25]
```

So we see that through the command line option -D[Macroname]=[Value] it was made possible.

Macros with piece of code as their values

As discussed in the introduction part, macros can also contain small piece of code as their values. Those piece of code which are very small and are being used repetitively in the code are assigned to macros. For example :

```
#include <stdio.h>

#define MACRO(x)  x * (x+5)
int main(void)
{
#ifdef MACRO // test whether MACRO1 is defined...
    printf("\nMACRO Defined...\n");
#endif
    int res = MACRO(2);
    printf("\n res = [%d]\n", res);
    return 0;
```

```
}
```

- So, In the code above we defined a parametrized macro that accepts a value and has a small piece of code associated with it.
- This macro is being used in the code to calculate value for the variable ‘res’.

When the above code is compiled and run, we see :

```
$ ./macro

MACRO Defined...

res = [14]
```

So we see that a parametrized macro (that has a small piece of code logic associated with it) was used to calculate the value for ‘res’.

How to Use C Macros and C Inline Functions with C Code Examples

Many C and C++ programming beginners tend to confuse between the concept of macros and Inline functions.

Often the difference between the two is also asked in C interviews.

In this tutorial we intend to cover the basics of these two concepts along with working code samples.

1. The Concept of C Macros

Macros are generally used to define constant values that are being used repeatedly in program. Macros can even accept arguments and such macros are known as function-like macros. It can be useful if tokens are concatenated into code to simplify some complex declarations. Macros provide text replacement functionality at pre-processing time.

Here is an example of a simple macro:

```
 #define MAX_SIZE 10
```

The above macro (MAX_SIZE) has a value of 10.

Now let's see an example through which we will confirm that macros are replaced by their values at pre-processing time. Here is a C program :

```
#include<stdio.h>

#define MAX_SIZE 10

int main(void)
{
   int size = 0;
   size = size + MAX_SIZE;

   printf("\n The value of size is [%d]\n",size);

   return 0;
}
```

Now lets compile it with the flag -save-temps so that pre-processing output (a file with extension .i) is produced along with final executable :

```
$ gcc -Wall -save-temps macro.c -o macro
```

The command above will produce all the intermediate files in the gcc compilation process. One of these files will be macro.i. This is the file of our interest. If you open this file and get to the bottom of this file :

```
...
...
...
int main(void)
{
   int size = 0;
   size = size + 10;

   printf("\n The value of size is [%d]\n",size);

   return 0;
}
```

So you see that the macro MAX_SIZE was replaced with it's value (10) in preprocessing stage of the compilation process.

Macros are handled by the pre-compiler, and are thus guaranteed to be inlined. Macros are used for short operations and it avoids function call overhead. It can be used if any short operation is being done in program repeatedly. Function-like macros are very beneficial when the same block of code needs to be executed multiple times.

Here are some examples that define macros for swapping numbers, square of numbers, logging function, etc.

```
#define SWAP(a,b)({a ^= b; b ^= a; a ^= b;})
#define SQUARE(x) (x*x)
#define TRACE_LOG(msg) write_log(TRACE_LEVEL, msg)
```

Now, we will understand the below program which uses macro to define logging function. It allows variable arguments list and displays arguments on standard output as per format specified.

```
#include <stdio.h>
#define TRACE_LOG(fmt, args...) fprintf(stdout, fmt, ##args);

int main() {
int i=1;
TRACE_LOG("%s", "Sample macro\n");
TRACE_LOG("%d %s", i, "Sample macro\n");
return 0;
}
```

Here is the output:

```
$ ./macro2
Sample macro
1 Sample macro
```

Here, TRACE_LOG is the macro defined. First, character string is logged by TRACE_LOG macro, then multiple arguments of different types are also logged as shown in second call of TRACE_LOG macro. Variable arguments are supported with the use of "…" in input argument of macro and ##args in input argument of macro value.

C Conditional Macros

Conditional macros are very useful to apply conditions. Code snippets are guarded with a condition checking if a certain macro is defined or not. They are very helpful in large project having code segregated as per releases of project. If some part of code needs to be executed for release 1 of project and some other part of code needs to be executed for release 2, then it can be easily achieved through conditional macros.

Here is the syntax :

```
#ifdef PRJ_REL_01
..
.. code of REL 01 ..
..
#else
..
.. code of REL 02 ..
..
#endif
```

To comment multiples lines of code, macro is used commonly in way given below :

```
#if 0
..
.. code to be commented ..
..
#endif
```

Here, we will understand above features of macro through working program that is given below.

```
#include <stdio.h>

int main() {

#if 0
printf("commented code 1");
printf("commented code 2");
#endif

#define TEST1 1

#ifdef TEST1
printf("MACRO TEST1 is defined\n");
#endif

#ifdef TEST3
printf("MACRO TEST3 is defined\n");
#else
printf("MACRO TEST3 is NOT defined\n");
#endif

return 0;
}
```

Output:

```
$ ./macro
MACRO TEST1 is defined
MACRO TEST3 is NOT defined
```

Here, we can see that “commented code 1”, “commented code 2” are not printed because these lines of code are commented under #if 0 macro. And, TEST1 macro is defined so, string “MACRO TEST1 is defined” is printed and since macro TEST3 is not defined, so “MACRO TEST3 is defined” is not printed.

2. The Concept of C Inline Functions

Inline functions are those functions whose definition is small and can be substituted at the place where its function call is made. Basically they are inlined with its function call.

Even there is no guarantee that the function will actually be inlined. Compiler interprets the inline keyword as a mere hint or request to substitute the code of function into its function call. Usually people say that having an inline function increases performance by saving time of function call overhead (i.e. passing arguments variables, return address, return value, stack mantle and its dismantle, etc.) but whether an inline function serves your purpose in a positive or in a negative way depends purely on your code design and is largely debatable.

Compiler does inlining for performing optimizations. If compiler optimization has been disabled, then inline functions would not serve their purpose and their function call would not be replaced by their function definition.

To have GCC inline your function regardless of optimization level, declare the function with the “always_inline” attribute:

```
void func_test() __attribute__((always_inline));
```

Inline functions provides following advantages over macros.

- Since they are functions so type of arguments is checked by the compiler whether they are correct or not.
- There is no risk if called multiple times. But there is risk in macros which can be dangerous when the argument is an expression.
- They can include multiple lines of code without trailing backlashes.
- Inline functions have their own scope for variables and they can return a value.
- Debugging code is easy in case of Inline functions as compared to macros.

It is a common misconception that inlining always equals faster code. If there are many lines in inline function or there are more function calls, then inlining can cause wastage of space.

Now, we will understand how inline functions are defined. It is very simple. Only, we need to specify “inline” keyword in its definition. Once you specify “inline” keyword in its definition, it request compiler to do optimizations for this function to save time by avoiding function call overhead. Whenever calling to inline function is made, function call would be replaced by definition of inline function.

```
#include <stdio.h>

void inline test_inline_func1(int a, int b) {
    printf ("a=%d and b=%d\n", a, b);
```

```
}

int inline test_inline_func2(int x) {
    return x*x;
}

int main() {

    int tmp;

    test_inline_func1(2,4);
    tmp = test_inline_func2(5);

    printf("square val=%d\n", tmp);

    return 0;
}
```

Output:

```
$ ./inline
a=2 and b=4
square val=25
```

File Handling in C with Examples (fopen, fread, fwrite, fseek)

As with any OS, file handling is a core concept in Linux. Any system programmer would learn it as one of his/her initial programming assignments. This aspect of programming involves system files.

Through file handling, one can perform operations like create, modify, delete etc on system files. Here in this article I try to bring in the very basic of file handling. Hope this article will clear the top layer of this multilayer aspect.

fopen()

```
FILE *fopen(const char *path, const char *mode);
```

The fopen() function is used to open a file and associates an I/O stream with it. This function takes two arguments. The first argument is a pointer to a string containing name of the file to be opened while the second argument is the mode in which the file is to be opened. The mode can be :

- ‘r’ : Open text file for reading. The stream is positioned at the beginning of the file.
- ‘r+’: Open for reading and writing. The stream is positioned at the beginning of the file.
- ‘w’ : Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.
- ‘w+’: Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- ‘a’ : Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- ‘a+’: Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

The fopen() function returns a FILE stream pointer on success while it returns NULL in case of a failure.

fread() and fwrite()

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The functions fread/fwrite are used for reading/writing data from/to the file opened by fopen function. These functions accept three arguments. The first argument is a pointer to buffer used for reading/writing the data. The data read/written is in the form of ‘nmemb’ elements each ‘size’ bytes long.

In case of success, fread/fwrite return the number of bytes actually read/written from/to the stream opened by fopen function. In case of failure, a lesser number of byes (then requested to read/write) is returned.

fseek()

```
int fseek(FILE *stream, long offset, int whence);
```

The fseek() function is used to set the file position indicator for the stream to a new position. This function accepts three arguments. The first argument is the FILE stream pointer returned by the fopen() function. The second argument ‘offset’ tells the amount

of bytes to seek. The third argument ‘whence’ tells from where the seek of ‘offset’ number of bytes is to be done. The available values for whence are SEEK_SET, SEEK_CUR, or SEEK_END. These three values (in order) depict the start of the file, the current position and the end of the file.

Upon success, this function returns 0, otherwise it returns -1.

fclose()

```
int fclose(FILE *fp);
```

The fclose() function first flushes the stream opened by fopen() and then closes the underlying descriptor. Upon successful completion this function returns 0 else end of file (eof) is returned. In case of failure, if the stream is accessed further then the behavior remains undefined.

The code

```
#include<stdio.h>
#include<string.h>

#define SIZE 1
#define NUMELEM 5

int main(void)
{
    FILE* fd = NULL;
    char buff[100];
    memset(buff,0,sizeof(buff));

    fd = fopen("test.txt","rw+");

    if(NULL == fd)
    {
        printf("\n fopen() Error!!!\n");
        return 1;
    }

    printf("\n File opened successfully through fopen()\n");

    if(SIZE*NUMELEM != fread(buff,SIZE,NUMELEM,fd))
    {
        printf("\n fread() failed\n");
        return 1;
    }

    printf("\n Some bytes successfully read through fread()\n");
```

```
    printf("\n The bytes read are [%s]\n",buff);

    if(0 != fseek(fd,11,SEEK_CUR))
    {
        printf("\n fseek() failed\n");
        return 1;
    }

    printf("\n fseek() successful\n");

    if(SIZE*NUMELEM != fwrite(buff,SIZE,strlen(buff),fd))
    {
        printf("\n fwrite() failed\n");
        return 1;
    }

    printf("\n fwrite() successful, data written to text file\n");

    fclose(fd);

    printf("\n File stream closed through fclose()\n");

    return 0;
}
```

The code above assumes that you have a test file “test.txt” placed in the same location from where this executable will be run.

Initially the content in file is :

```
$ cat test.txt
hello everybody
```

Now, run the code :

```
$ ./fileHandling

 File opened successfully through fopen()

 Some bytes successfully read through fread()

 The bytes read are [hello]

 fseek() successful

 fwrite() successful, data written to text file

 File stream closed through fclose()
```

Again check the contents of the file test.txt. As you see below, the content of the file was modified.

```
$ cat test.txt
hello everybody
hello
```

C argc and argv Examples to Parse Command Line Arguments

Whenever you execute a program on a terminal, you can pass some arguments that are expected by the program, which can be used during the execution of the program. Here, system provides internal facility to maintain all arguments passed from user while executing program. These arguments are known as “Command line arguments”.

In this tutorial, we will map the understanding of command line arguments with working program to understand it better in crisp and clear way. But before jumping to program, we should know how system provides facility of command line arguments. As we know, Every C program must have main() function and the facility of command line arguments is provided by the main() function itself. When given below declaration is used in program, and then program has facility to use/manipulate command line arguments.

```
int main (int argc, char *argv[])
```

Here, argc parameter is the count of total command line arguments passed to executable on execution (including name of executable as first argument). argv parameter is the array of character string of each command line argument passed to executable on execution. If you are new to C programming, you should first understand how C array works.

Given below is the working program using command line argument.

```
#include <stdio.h>

int main (int argc, char *argv[]) {
int i=0;
printf("\ncmdline args count=%s", argc);

/* First argument is executable name only */
```

```
 printf("\nexe name=%s", argv[0]);

 for (i=1; i< argc; i++) {
     printf("\narg%d=%s", i, argv[i]);
 }

 printf("\n");
 return 0;
 }
```

Given below is output when program is executed.

```
$ ./cmdline_basic test1 test2 test3 test4 1234 56789
cmdline args count=7
 exe name=./cmdline_basic
 arg1=test1
 arg2=test2
 arg3=test3
 arg4=test4
 arg5=1234
 arg6=56789
```

In above output, we can see total arguments count is internally maintained by “argc” parameter of main() which holds value ‘7’ (in which one argument is executable name and ‘6’ are arguments passed to program).And, all argument values are stored in “argv” parameter of main() which is array of character strings. Here, main () function stores each argument value as character string. We can see, iterating over “argv” array, we can get all passed arguments in the program.

There is one more declaration of main () function that provides added facility to work on environment variables inside program. Like, arguments maintained in argv[] array, main() function has internal facility to maintain all system environment variables into array of character strings which can be taken as an main() function parameter. Given below is the declaration.

```
int main (int argc, char *argv[], char **envp)
```

Given below is the working program using command line argument along with environment variables.

```
#include <stdio.h>

int main (int argc, char *argv[], char **env_var_ptr) {
int i=0;
printf("\ncmdline args count=%d", argc);

/* First argument is executable name only */
```

```
printf("\nexe name=%s", argv[0]);

for (i=1; i< argc; i++) {
   printf("\narg%d=%s", i, argv[i]);
 }

i=0;
while (*env_var_ptr != NULL) {
    i++;
    printf ("\nenv var%d=>%s",i, *(env_var_ptr++));
 }

printf("\n");
return 0;
}
```

Output of above program is given below.

```
$ ./env test1 test2
cmdline args count=3
 exe name=./env
 arg1=test1
 arg2=test2
 env var1=>SSH_AGENT_PID=1575
 env var2=>KDE_MULTIHEAD=false
 env var3=>SHELL=/bin/bash
 env var4=>TERM=xterm
 env         var5=>XDG_SESSION_COOKIE=5edf27907e97deafc70d310550995c84-
1352614770.691861-1384749481
 env       var6=>GTK2_RC_FILES=/etc/gtk-2.0/gtkrc:/home/sitaram/.gtkrc-
2.0:/home/sitaram/.kde/share/config/gtkrc-2.0
 env var7=>KONSOLE_DBUS_SERVICE=:1.76
 env var8=>KONSOLE_PROFILE_NAME=Shell
 env var9=>GS_LIB=/home/sitaram/.fonts
 env
var10=>GTK_RC_FILES=/etc/gtk/gtkrc:/home/sitaram/.gtkrc:/home/sitaram/
.kde/share/config/gtkrc
 env var11=>WINDOWID=29360154
 env var12=>GNOME_KEYRING_CONTROL=/run/user/sitaram/keyring-2Qx7DW
 env var13=>SHELL_SESSION_ID=f7ac2d9459c74000b6fd9b2df1d48da4
 env var14=>GTK_MODULES=overlay-scrollbar
 env var15=>KDE_FULL_SESSION=true
 env var16=>http_proxy=http://10.0.0.17:8080/
 env var17=>USER=sitaram
 env
var18=>LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;
35:bd=40;33;01:cd=40;33;01:or=40;31;01:su=37;41:sg=30;43:ca=30;41:tw=3
0;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.
taz=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.zip=01;31
:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lz=01;31:*.xz=01;31:*.bz2
=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rp
m=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.
```

```
ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.jpg=01;35:*
.jpeg=01;35:*.gif=01;35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;3
5:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=0
1;35:*.svg=01;35:*.svgz=01;35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mp
g=01;35:*.mpeg=01;35:*.m2v=01;35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:
*.mp4=01;35:*.m4v=01;35:*.mp4v=01;35:*.vob=01;35:*.qt=01;35:*.nuv=01;3
5:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;35:*.flc=01;35:*.avi=01
;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;35:*.xwd=01
;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.axv=01;35:*.anx=01;35:*.ogv=
01;35:*.ogx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.mid=00;36:*.mi
di=00;36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.
wav=00;36:*.axa=00;36:*.oga=00;36:*.spx=00;36:*.xspf=00;36:
 env var19=>XDG_SESSION_PATH=/org/freedesktop/DisplayManager/Session0
 env var20=>XDG_SEAT_PATH=/org/freedesktop/DisplayManager/Seat0
 env var21=>SSH_AUTH_SOCK=/tmp/ssh-kIFY5HttOJxe/agent.1489
 env var22=>ftp_proxy=ftp://10.0.0.17:8080/
 env                    var23=>SESSION_MANAGER=local/Sitaram:@/tmp/.ICE-
unix/1716,unix/Sitaram:/tmp/.ICE-unix/1716
 env var24=>DEFAULTS_PATH=/usr/share/gconf/kde-plasma.default.path
 env var25=>XDG_CONFIG_DIRS=/etc/xdg/xdg-kde-plasma:/etc/xdg
 env var26=>DESKTOP_SESSION=kde-plasma
 env
var27=>PATH=/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/u
sr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
 env var28=>PWD=/home/sitaram/test_progs/cmdline
 env var29=>socks_proxy=socks://10.0.0.17:8080/
 env var30=>KONSOLE_DBUS_WINDOW=/Windows/1
 env var31=>KDE_SESSION_UID=1000
 env var32=>LANG=en_IN
 env var33=>GNOME_KEYRING_PID=1478
 env var34=>MANDATORY_PATH=/usr/share/gconf/kde-plasma.mandatory.path
 env var35=>UBUNTU_MENUPROXY=libappmenu.so
 env var36=>KONSOLE_DBUS_SESSION=/Sessions/1
 env var37=>https_proxy=https://10.0.0.17:8080/
 env var38=>GDMSESSION=kde-plasma
 env var39=>SHLVL=1
 env var40=>HOME=/home/sitaram
 env var41=>COLORFGBG=15;0
 env var42=>KDE_SESSION_VERSION=4
 env var43=>LANGUAGE=en_IN:en
 env var44=>XCURSOR_THEME=Oxygen_White
 env var45=>LOGNAME=sitaram
 env                                var46=>XDG_DATA_DIRS=/usr/share/kde-
plasma:/usr/local/share/:/usr/share/
 env           var47=>DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
mnJhMvd4jG,guid=435ddd41500fd6c5550ed8d2509f4374
 env var48=>LESSOPEN=| /usr/bin/lesspipe %s
 env var49=>PROFILEHOME=
 env var50=>XDG_RUNTIME_DIR=/run/user/sitaram
 env var51=>DISPLAY=:0
```

```
 env
var52=>QT_PLUGIN_PATH=/home/sitaram/.kde/lib/kde4/plugins/:/usr/lib/kd
e4/plugins/
 env var53=>LESSCLOSE=/usr/bin/lesspipe %s %s
 env var54=>XAUTHORITY=/tmp/kde-sitaram/xauth-1000-_0
 env var55=>_=./env
 env var56=>OLDPWD=/home/sitaram/test_progs
$
```

In above output, we can see all system environment variables can be obtained third parameter of main() function which are traversed in program and displayed in output.

Passing command line arguments to program and manipulate arguments

Given below is program working on command line arguments.

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
int i=0;
int d;
float f;
long int l;
FILE *file = NULL;
printf("\ncmdline args count=%d", argc);

/* First argument is executable name only */
printf("\nexe name=%s", argv[0]);

for (i=1; i< argc; i++) {
    printf("\narg%d=%s", i, argv[i]);
 }

/* Conversion string into int */
d = atoi(argv[1]);
printf("\nargv[1] in intger=%d",d);

/* Conversion string into float */
f = atof(argv[1]);
printf("\nargv[1] in float=%f",f);

/* Conversion string into long int */
l = strtol(argv[2], NULL, 0);
printf("\nargv[2] in long int=%ld",l);

/*Open file whose path is passed as an argument */
file = fopen( argv[3], "r" );

/* fopen returns NULL pointer on failure */
if ( file == NULL) {
```

```
    printf("\nCould not open file");
  }
else {
    printf("\nFile (%s) opened", argv[3]);
    /* Closing file */
    fclose(file);
  }

printf("\n");
return 0;
}
```

Output of above program is given below.

```
 $ ./cmdline_strfunc 1234test 12345678
 /home/sitaram/test_progs/cmdline/cmdline_strfunc.c
cmdline args count=4
 exe name=./cmdline_strfunc
 arg1=1234test
 arg2=12345678
 arg3=/home/sitaram/test_progs/cmdline/cmdline_strfunc.c
 argv[1] in intger=1234
 argv[1] in float=1234.000000
 argv[2] in long int=12345678
 File (/home/sitaram/test_progs/cmdline/cmdline_strfunc.c) opened
```

In above output, we can see that command line arguments can be manipulated in program; all arguments are obtained as character string which can be converted into integer, float, long as shown in program. Even any character string if passed as an path of any file that can be used by program to file handling operation oh that file. We can see in above program, (/home/sitaram/test_progs/cmdline/cmdline_strfunc.c) file path is passed as an command line argument which is used inside program to open the file and close the file.

Getopt() API

If we explore more on command line arguments, we have very powerful API – getopt(). It facilitates programmer to parse command line options. Programmer can give list of mandatory or optional command line options to getopt(). It can determine whether command line option is either valid or invalid as per program expected command line options. There are few getopt() specific internal variables like “optarg, optopt, opterr”

- Optarg: contains pointer to command line valid option’s argument
- Optopt: contains command line option if mandatory command line option is missing
- Opterr: set to non-zero when invalid option is provided or value of mandatory command line option is not given

Given below is basic program to understand parsing of command line options.

```
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[]) {
int opt = 0;
char *in_fname = NULL;
char *out_fname = NULL;

while ((opt = getopt(argc, argv, "i:o:")) != -1) {
    switch(opt) {
    case 'i':
    in_fname = optarg;
    printf("\nInput option value=%s", in_fname);
    break;
    case 'o':
    out_fname = optarg;
    printf("\nOutput option value=%s", out_fname);
    break;
    case '?':
    /* Case when user enters the command as
     * $ ./cmd_exe -i
     */
    if (optopt == 'i') {
    printf("\nMissing mandatory input option");
    /* Case when user enters the command as
     * # ./cmd_exe -o
     */
  } else if (optopt == 'o') {
     printf("\nMissing mandatory output option");
  } else {
     printf("\nInvalid option received");
  }
  break;
 }
 }

printf("\n");
return 0;
 }
```

Output of above program is given below with few combinations of command line options:

```
Case1:
$ ./cmdline_getopt -i /tmp/input -o /tmp/output
Input option value=/tmp/input
 Output option value=/tmp/output

Case2:
$ ./cmdline_getopt -i -o /tmp/output
```

```
Input option value=-o

Case3:
$ ./cmdline_getopt -i
 ./cmdline_getopt: option requires an argument -- 'i'
Missing mandatory input option

Case4:
$ ./cmdline_getopt -i /tmp/input -o
./cmdline_getopt: option requires an argument -- 'o'
 Input option value=/tmp/input
 Missing mandatory output option

Case5:
$ ./cmdline_getopt -k /tmp/input
 ./cmdline_getopt: invalid option -- 'k'
Invalid option received
```

In above program, ‘i’ and ‘o’ are taken as mandatory input and output command line options for program using getopt() API.

We would now have basic explanation of each case executed in above program:

- In Case1, both mandatory command line options with their arguments are provided which are properly handled in first two cases of switch condition of program.
- In Case2, value of mandatory input option is not given but we can see getopt() is not intelligent enough and considered “-o” as value of ‘I’ command line option. It is not error case for getopt(), but programmer can itself add intelligence to handle such case.
- In Case3, only command line option is specified without its value and this is mandatory option so in this case getopt() would return ‘?’ and “optopt” variable is set to ‘i’ to confirm mandatory input option’s value is missing.
- In Case4, mandatory output option’s value is missing.
- In Case5, invalid command line option is given which is not mandatory or optional command line option. In this case, getopt() returned ‘?’ and optopt is not set since it is unknown character not expected by getopt().

C Loops Explained with Examples (For Loop, Do While and While)

Loops are very basic and very useful programming facility that facilitates programmer to execute any block of code lines repeatedly and can be controlled as per conditions added by programmer. It saves writing code several times for same task.
There are three types of loops in C.

1. For loop
2. Do while loop
3. While loop

1. For Loop Examples

Basic syntax to use 'for' loop is:

```
for (variable initialization; condition to control loop; iteration of variable) {
    statement 1;
    statement 2;
    ..
    ..
}
```

In the pseudo code above :

- Variable initialization is the initialization of counter of loop.
- Condition is any logical condition that controls the number of times the loop statementsare executed.
- Iteration is the increment/decrement of counter.

It is noted that when 'for' loop execution starts, first variable initialization is done, then condition is checked before execution of statements; if and only if condition is TRUE, statements are executed; after all statements are executed, iteration of counter of loop is done either increment or decrement.

Here is a basic C program covering usage of 'for' loop in several cases:

```
#include <stdio.h>

int main () {

    int i = 0, k = 0;
    float j = 0;
    int loop_count = 5;

    printf("Case1:\n");
    for (i=0; i < loop_count; i++) {
        printf("%d\n",i);
```

```
    }

    printf("Case2:\n");
    for (j=5.5; j > 0; j--) {
        printf("%f\n",j);
    }

    printf("Case3:\n");
    for (i=2; (i < 5 && i >=2); i++) {
        printf("%d\n",i);
    }

    printf("Case4:\n");
    for (i=0; (i != 5); i++) {
        printf("%d\n",i);
    }

    printf("Case5:\n");
    /* Blank loop  */
    for (i=0; i < loop_count; i++) ;

    printf("Case6:\n");
    for (i=0, k=0; (i < 5 && k < 3); i++, k++) {
        printf("%d\n",i);
    }

    printf("Case7:\n");
    i=5;
    for (; 0; i++) {
        printf("%d\n",i);
    }

    return 0;
}
```

In the code above :

- Case1 (Normal) : Variable ‘i’ is initialized to 0; condition is to execute loop till ‘i’ is lesser than value of ‘loop_count’ variable; iteration is increment of counter variable ‘i’
- Case2 (Using float variable) : Variable ‘j’ is float and initialized to 5.5; condition is to execute loop till ‘j’ is greater than ‘0’; iteration is decrement of counter variable ‘j’.
- Case3 (Taking logical AND condition) : Variable ‘i’ is initialized to 2; condition is to execute loop when ‘i’ is greater or equal to ‘2’and lesser than ‘5’; iteration is increment of counter variable ‘i’.

- Case4 (Using logical NOT EQUAL condition) : Variable ‘i’ is initialized to 0; condition is to execute loop till ‘i’ is NOT equal to ‘5’; iteration is increment of counter variable ‘i’.
- Case5 (Blank Loop) : This example shows that loop can execute even if there is no statement in the block for execution on each iteration.
- Case6 (Multiple variables and conditions) : Variables ‘i’ and ‘k’ are initialized to 0; condition is to execute loop when ‘i’ is lesser than ‘5’and ‘k’ is lesser than ‘3’; iteration is increment of counter variables ‘i’ and ‘k’.
- Case7 (No initialization in for loop and Always FALSE condition) : Variables ‘i’ is initialized before for loop to ‘5’; condition is FALSE always as ‘0’ is provided that causes NOT to execute loop statement; iteration is increment of counter variable ‘i’.

Here is the output of the above program :

```
# ./a.out
Case1:
0
1
2
3
4
Case2:
5.500000
4.500000
3.500000
2.500000
1.500000
0.500000
Case3:
2
3
4
Case4:
0
1
2
3
4
Case5:
Case6:
0
1
2
Case7:
```

Loop can run infinitely if condition is set to TRUE always or no condition is specified. For example:

```
for (;;)
```

If loop contain only one statement then braces are optional; generally it is preferred to use braces from readability point of view. For example :

```
for (j=0;j<5;j++)
    printf("j");
```

Loops can be nested too. There can be loop inside another loop. Given below is example for nested loop to display right angle triangle of '@' symbol.

```
for (i=0; i < 5; i++)
{
    for (j=0;j<=i;j++)
    {
        printf("@");
    }
}
```

Just like For Loops, it is also important for you to understand C Pointers fundamentals.

2. Do While Loop Examples

It is another loop like 'for' loop in C. But do-while loop allows execution of statements inside block of loop for one time for sure even if condition in loop fails.

Basic syntax to use 'do-while' loop is:

```
variable initialization;
do {
statement 1;
statement 2;
..
..
iteration of variable;
} while (condition to control loop)
```

In the pseudo code above :

- Variable initialization is the initialization of counter of loop before start of 'do-while' loop.
- Condition is any logical condition that controls the number of times execution of loop statements
- Iteration is the increment/decrement of counter

Here is a basic C program covering usage of 'do-while' loop in several cases:

```
#include <stdio.h>
int main () {
int i = 0;
int loop_count = 5;

printf("Case1:\n");
do {
printf("%d\n",i);
i++;
} while (i<loop_count);

printf("Case2:\n");
i=20;
do {
printf("%d\n",i);
i++;
} while (0);

printf("Case3:\n");
i=0;
do {
printf("%d\n",i);
} while (i++<5);

printf("Case4:\n");
i=3;
do {
printf("%d\n",i);
i++;
} while (i < 5 && i >=2);
return 0;
}
```

In the code above :

- Case1 (Normal) :Variable ‘i’ is initialized to 0 before ‘do-while’ loop; iteration is increment of counter variable ‘i’; condition is to execute loop till ‘i’ is lesser than value of ‘loop_count’ variable i.e. 5.
- Case2 (Always FALSE condition) : Variables ‘i’ is initialized before ‘do-while’ loop to ‘20’; iteration is increment of counter variable ‘i’; condition is FALSE always as ‘0’ is provided that causes NOT to execute loop statements, but it is noted here in output that loop statement is executed once because do-while loop always executes its loop statements at least once even if condition fails at first iteration.
- Case3 (Iteration in condition check expression) : Variable ‘i’ is initialized to 0 before ‘do-while’ loop; here note that iteration and condition is provided in same expression. Here, observe the condition is to execute loop till ‘i’ is lesser than

‘5’, but in output 5 is also printed that is because, here iteration is being done at condition check expression, hence on each iteration ‘do-while’ loop executes statements ahead of condition check.

- Case4 (Using logical AND condition) : Variable ‘i’ is initialized before ‘do-while’ loop to ‘3’; iteration is increment of counter variable ‘i’; condition is execute loop when ‘i’ is lesser than ‘5’ AND ‘i’ is greater or equal to ‘2’.

Here is output for above program.

```
# ./a.out
 Case1:
 0
 1
 2
 3
 4
 Case2:
 20
 Case3:
 0
 1
 2
 3
 4
 5
 Case4:
 3
 4
#
```

Also, if you are interested, read about our earlier article on bitwise operators in C.

3. While Loop Examples

It is another loop like ‘do-while’ loop in C. The ‘while’ loop allows execution of statements inside block of loop only if condition in loop succeeds.

Basic syntax to use ‘while’ loop is:

```
variable initialization;
 while (condition to control loop) {
 statement 1;
 statement 2;
 ..
 ..
 iteration of variable;
 }
```

In the pseudo code above :

- Variable initialization is the initialization of counter of loop before start of ‘while’ loop
- Condition is any logical condition that controls the number of times execution of loop statements
- Iteration is the increment/decrement of counter

Basic C program covering usage of ‘while’ loop in several cases:

```
#include <stdio.h>

int main () {

    int i = 0;
    int loop_count = 5;

    printf("Case1:\n");
    while (i<loop_count) {
        printf("%d\n",i);
        i++;
    }

    printf("Case2:\n");
    i=20;
    while (0) {
        printf("%d\n",i);
        i++;
    }

    printf("Case3:\n");
    i=0;
    while (i++<5) {
        printf("%d\n",i);
    }
    printf("Case4:\n");
    i=3;
    while (i < 5 && i >=2) {
        printf("%d\n",i);
        i++;
    }

    return 0;
}
```

In the code above:

- Case1 (Normal) : Variable ‘i’ is initialized to 0 before ‘while’ loop; iteration is increment of counter variable ‘i’; condition is execute loop till ‘i’ is lesser than value of ‘loop_count’ variable i.e. 5.

- Case2 (Always FALSE condition) : Variables ‘i’ is initialized before ‘while’ loop to ‘20’; iteration is increment of counter variable ‘i’; condition is FALSE always as ‘0’ is provided that causes NOT to execute loop statements and loop statement is NOT executed. Here, it is noted that as compared to ‘do-while’ loop, statements in ‘while’ loop are NOT even executed once which executed at least once in ‘do-while’ loop because ‘while’ loop only executes loop statements only if condition succeeds.
- Case3 (Iteration in condition check expression) :Variable ‘i’ is initialized to 0 before ‘while’ loop; here note that iteration and condition is provided in same expression. Here, observe the condition is execute loop till ‘i’ is lesser than ‘5’ and loop iterates 5 times. Unlike ‘do-while’ loop, here condition is checked first then ‘while’ loop executes statements.
- Case4 (Using logical AND condition) :Variable ‘i’ is initialized before ‘while’ loop to ‘3’; iteration is increment of counter variable ‘i’; condition is execute loop when ‘i’ is lesser than ‘5’ AND ‘i’ is greater or equal to ‘2’.

Below is output for above program.

```
# ./a.out
Case1:
0
1
2
3
4
Case2:
Case3:
1
2
3
4
5
Case4:
3
4
#
```

C If and Switch Case Examples (if, if else, if else if, nested if)

Control conditions are the basic building blocks of C programming language. In this tutorial, we will cover the control conditions through some easy to understand examples.

There are two types of conditions :

- Decision making condition statement
- Selection condition statement

Let's understand these two types with the help of examples.

Decision making condition statement

Conditions like 'if', "if-else", "if-else-if", "nested if", ternary conditions etc fall under this category.

1. If Condition

This is basic most condition in C – 'if' condition. If programmer wants to execute some statements only when any condition is passed, then this single 'if' condition statement can be used. Basic syntax for 'if' condition is given below:

```
if (expression) {
    Statement 1;
    Statement 1;
    ..
    ..
}
```

Now, we should have working program on 'if' condition.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
if (argc != 2) {
     printf("Can  not  execute,  command  line  argument  expected  by
Program\n");
    exit(0);
}
return 0;
}
```

Output for above program is given below.

```
$ ./if_cond
Can not execute, command line argument expected by Program
```

In above program, programmer wanted to exit from program if two command line arguments are not passed to program. We can see if program executable is run without any argument, message is displayed on console and program exited.

2. If-Else Condition

This is two-way condition in C – 'if-else' condition. If programmer wants to execute one set of statements on success case of one condition and another set of statements in all other cases, then 'if-else' condition is used. Either 'if' case statements are executed or 'else' case statements are executed. Basic syntax for 'if-else' condition is given below:

```
if (expression1) {
    Statements;
} else {
    Statements;
}
```

Now, given below is very basic program that has been made for checking number is even or odd, it is for understanding usage of 'if-else' condition.

```
#include <stdio.h>
int main(int argc, char *argv[]) {
int num;
printf("\nEnter any number to check even or odd :");
scanf("%d", &num);
if (num%2 == 0) {
    printf("%d is EVEN\n", num);
} else {
    printf("%d is ODD\n", num);
}
return 0;
}
```

Output:

```
$ ./if-else_cond

Enter any number to check even or odd :23
23 is ODD
$ ./if-else_cond

Enter any number to check even or odd :24
24 is EVEN
```

In above program, programmer wanted user to enter number which is checked in condition whether it is divisible by 2. If condition is true, number is displayed "EVEN", otherwise number is displayed "ODD".

3. Ternary Operator

There is alternative to 'if-else' condition which is ternary operator that is different syntax but provides functionality of 'if-else' condition. Basic syntax of ternary operator is given below:

```
Condition expression ? if condition TRUE, return value1 : Otherwise,
return value2;
```

For example,

```
#include <stdio.h>
int main(int argc, char *argv[]) {
int num;
printf("\nEnter any number to check even or odd :");
scanf("%d", &num);
(num%2==0) ? printf("%d is EVEN\n", num) : printf("%d is ODD\n", num);
return 0;
}
```

Output:

```
$ ./a.out

Enter any number to check even or odd :24
24 is EVEN
$ ./a.out

Enter any number to check even or odd :23
23 is ODD
```

4. If-Else-If condition

This is multi-way condition in C – 'if-else-if' condition. If programmer wants to execute different statements in different conditions and execution of single condition out of multiple conditions at one time, then this 'if-else-if' condition statement can be used. Once any condition is matched, 'if-else-if' condition is terminated. Basic syntax for 'if-else-if' condition is given below:

```
if (expression1) {
    Statements;
} else if (expression2) {
    Statements;
} else {
    Statements;
}
```

Now, given below is very basic program that has been made for mapping user-input color with fruit, it is for understanding usage of 'if-else-if' condition.

```
#include <stdio.h>
```

```
int main(int argc, char *argv[]) {

char input_color[100] = {0};

printf("\nEnter color [red/green/yellow] to map with fruit :");
scanf("%s", input_color);

if (strncmp(input_color, "red", sizeof(input_color)) == 0) {
    printf("%s is mapped to APPLE\n", input_color);
} else if (strncmp(input_color, "green", sizeof(input_color)) == 0) {
    printf("%s is mapped to GRAPES\n", input_color);
} else if (strncmp(input_color, "yellow", sizeof(input_color)) == 0) {
    printf("%s is mapped to BANANA\n", input_color);
} else {
    printf("\nInvalid color entered :%s", input_color);
}

return 0;
}
```

Output:

```
$ ./if-else-if_cond

Enter color [red/green/yellow] to map with fruit :green
green is mapped to GRAPES
$ ./if-else-if_cond

Enter color [red/green/yellow] to map with fruit :yellow
yellow is mapped to BANANA
$ ./if-else-if_cond

Enter color [red/green/yellow] to map with fruit :testcolor

Invalid color entered :testcolor
```

In above program, programmer wanted user to enter color (out of red/green/yellow as indicated), then input color is compared first with red in ‘if condition’, then compared with ‘else-if’ conditions. Here, it is noted that once any condition is matched, ‘if-else-if’ condition terminates. Here, if no ‘if’ or ‘else if’ is matched, then at last ‘else’ condition is executed which we can see in above output when invalid color is input.

5. Nested-If conditions

This is nested if or if-else or if-else-if conditions in C. Basic syntax for nested ‘if’ or ‘if-else’ condition is given below:

```
if (expression1) {
    Statements;
    if (expression2) {
```

```
        Statements;
    } else {
        Statements;
    }
}

Given below is basic program using nested if conditions.
#include <stdio.h>

int main(int argc, char *argv[]) {

int i = 5;
int *ptr = &i;
int **double_ptr = &ptr;

if (double_ptr != NULL) {

    if (ptr != NULL) {
        printf ("Now safe to access pointer, ptr contains %d", *ptr);
    }
}
return 0;
}
```

Output:

```
$ ./a.out

Now safe to access pointer, ptr contains 5
```

In above program, nested if conditions are used. It is always safer to have NULL check on pointer before accessing it (More on C pointers here).

In the above code snippet, example is taken for double pointer. The first 'if' condition is to check double pointer (i.e. ** double_ptr) is non-NULL, then only, move ahead to access inner pointer (i.e. ptr). If double pointer is non-NULL, then only nested 'if' condition is checked whether inner pointer is NULL or not. If nested 'if' condition is OK, then it is safe to access value at pointer.

Selection condition statement

6. Switch case conditions

Switch case is clean alternative of 'if-else-if' condition. Here, several conditions are given in cases that facilitates user to select case as per input entered. Basic syntax for using switch case statement is given below.

```
switch(expression) {
        case constant expression1:
```

```
            statements1; break;
        case constant expression2:
            statements1; break;
         ..
         ..
        default : statementsN;
}
```

It is noted that any statement between switch statement and first case statement is dead code which is never executed. For understanding ‘switch’ case, basic program is created in which basic arithmetic operation on two numbers is done as per input entered by user. Several cases of arithmetic operations are handled in switch cases. Basic program using ‘switch case’ is given below.

```
#include <stdio.h>
int main(int argc, char *argv[]) {
char ch;
int num1, num2;

printf("\nBasic operation:");
printf("\nAdd [a]");
printf("\nSubtract [s]");
printf("\nMultiply [m]");
printf("\nDivide [d]");

printf("\nEnter character for operation:");
scanf("%c", &ch);

printf("\nEnter two numbers for operation:");
printf("\nEnter num1=");
scanf("%d", &num1);
printf("\nEnter num2=");
scanf("%d", &num2);

switch (ch) {
    case 'a':
        printf("\nAddition of num1 and num2=%d", (num1+num2));
        break;

    case 's':
        printf("\nSubtraction of num1 and num2=%d", (num1-num2));
        break;

    case 'm':
        printf("\nMultiplication of num1 and num2=%d", (num1*num2));
        break;

    case 'd':
        printf("\nDivision of num1 and num2=%d", (num1/num2));
        break;
    case 'x':
```

```
        printf ("\nTest switch case1");
    case 'y':
        printf ("\nTest switch case2");
    default:
        printf("\nInvalid value eneterd");
        break;
}
printf("\n");
return 0;
}
```

Output:

```
$ ./a.out

Basic operation:
Add [a]
Subtract [s]
Multiply [m]
Divide [d]
Enter character for operation:a

Enter two numbers for operation:
Enter num1=10

Enter num2=5

Addition of num1 and num2=15
$ ./a.out

Basic operation:
Add [a]
Subtract [s]
Multiply [m]
Divide [d]
Enter character for operation:d

Enter two numbers for operation:
Enter num1=10

Enter num2=5

Division of num1 and num2=2
$ ./a.out

Basic operation:
Add [a]
Subtract [s]
Multiply [m]
Divide [d]
Enter character for operation:G
```

```
Enter two numbers for operation:
Enter num1=10

Enter num2=5

Invalid value entered
```

In above program, user is given basic menu with operations allowed in program. User is asked to enter initial character of displayed operations. User is asked to enter two numbers also on which selected arithmetic operation would be performed. After all input from user, program checks input with switch cases and executes statements under matched switch case; since break statement is there so only statements under matched case are executed.

Note that if break statement is not given in cases and any case is matched, then statements of below cases would also get executed even though below cases condition is not matched. We can understand this in given below output. Here, as per code, if 'x' is entered, then case 'x' is executed and there is no break statement so all cases below case 'x' are executed without any condition check on below cases.

```
$ ./a.out

Basic operation:
Add [a]
Subtract [s]
Multiply [m]
Divide [d]
Enter character for operation:x

Enter two numbers for operation:
Enter num1=10

Enter num2=5

Test switch case1
Test switch case2
Invalid value entered
```

C program to find second most frequent character

Given a string, find the second most frequent character in it. Expected time complexity is O(n) where n is the length of the input string.

Examples:

```
Input: str = "aababababa";
Output: Second most frequent character is 'b'

Input: str = "geeksforgeeks";
Output: Second most frequent character is 'g'

Input: str = "geeksquiz";
Output: Second most frequent character is 'g'
The output can also be any other character with
count 1 like 'z', 'i'.

Input: str = "abcd";
Output: No Second most frequent character
```

A simple solution is to start from the first character, count its occurrences, then second character and so on. While counting these occurrence keep track of max and second max. Time complexity of this solution is O(n2).

We can solve this problem in O(n) time using a count array with size equal to 256 (Assuming characters are stored in ASCII format). Following is C implementation of the approach.

```
#include <stdio.h>
#define NO_OF_CHARS 256

// C function to find the second most frequent character
// in a given string 'str'
char getSecondMostFreq(char *str)
{
    // count number of occurrences of every character.
    int count[NO_OF_CHARS] = {0}, i;
    for (i=0; str[i]; i++)
        (count[str[i]])++;

    // Traverse through the count[] and find second highest element.
    int first = 0, second = 0;
    for (i = 0; i < NO_OF_CHARS; i++)
    {
        /* If current element is smaller than first then update both
          first and second */
        if (count[i] > count[first])
        {
```

```
            second = first;
            first = i;
        }

        /* If count[i] is in between first and second then update
second  */
        else if (count[i] > count[second] &&
                 count[i] != count[first])
            second = i;
    }

    return second;
}

// Driver program to test above function
int main()
{
  char str[] = "geeksforgeeks";
  char res = getSecondMostFreq(str);
  if (res != '\0')
     printf("Second most frequent char is %c", res);
  else
     printf("No second most frequent character");
  return 0;
}
```

Output:

```
Second most frequent char is g
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

C Program to Check if a Given String is Palindrome

Given a string, write a c function to check if it is palindrome or not. A string is said to be palindrome if reverse of the string is same as string. For example, “abba” is palindrome, but “abbc” is not palindrome.

Algorithm:
isPalindrome(str)

1) Find length of str. Let length be n.
2) Initialize low and high indexes as 0 and n-1 respectively.
3) Do following while low index ‘l’ is smaller than high index ‘h’.
    a. If str[l] is same as str[h], then return false.
    b. Increment l and decrement h, i.e., do l++ and h–.

4) If we reach here, it means we didn’t find a mis

Following is C implementation to check if a given string is palindrome or not.

```
#include <stdio.h>
#include <string.h>

// A function to check if a string str is palindrome
void isPalindrome(char str[])
{
    // Start from leftmost and rightmost corners of str
    int l = 0;
    int h = strlen(str) - 1;

    // Keep comparing characters while they are same
    while (h > l)
    {
        if (str[l++] != str[h--])
        {
            printf("%s is Not Palindrome\n", str);
            return;
        }
    }
    printf("%s is palindrome\n", str);
}

// Driver program to test above function
int main()
{
    isPalindrome("abba");
    isPalindrome("abbccbba");
    isPalindrome("geeks");
    return 0;
}
```

Output:

```
abba is palindrome
abbccbba is palindrome
geeks is Not Palindrome
```

C Recursion Fundamentals Explained with Examples

In C programming language, when a function calls itself over and over again, that function is known as recursive function.

The process of function calling itself repeatedly is known as recursion.

In this tutorial, we will understand the concept of recursion using practical examples.

1. C Recursion Concept

Lets start with a very basic example of recursion :

```
#include <stdio.h>

void func(void)
{
    printf("\n This is a recursive function \n");
    func();
    return;
}

int main(void)
{
    func();
    return 0;
}
```

In the code above, you can see that the function func(), in its definition calls itself. So, func() becomes a recursive function. Can you guess what will happen when the code (shown above) is executed? If we go by the code, the main() function would call func() once and then func() would continue calling itself forever. Will this be the exact behaviour?

Lets execute the code and check this. Here is the output :

```
$ ./recrsn
This is a recursive function

 This is a recursive function

 ....
 ....

 This is a recursive function

 This is a recursive function

 This is a recursive function
Segmentation fault (core dumped)
```

In the output above:

- The print “This is a recursive function” prints continuously many times.
- A set of three dots “…” is used to omit large part of actual output which was nothing but the same print.
- Towards the end of the output you cab observe “Segmentation fault” or as we popularly say, the program crashes.

Earlier, we thought that the program would continue executing forever because recursive function func() would continue calling itself forever but it did not happen so. The program crashed. Why did it crash?

Here is the reason for this crash :

- For each call to func(), a new function stack is created.
- With func() calling itself continuously, new function stacks also are also created continuously.
- At one point of time, this causes stack overflow and hence the program crashes.

On a related note, it is also important for you to get a good understanding on Buffer Over Flowand Linked Lists.

2. Practical Example of C Recursion

For complete newbies, it ok to have a question like What's the practical use of recursion? In this section, I will provide some practical examples where recursion can makes things really easy.

Suppose you have numbers from 0 to 9 and you need to calculate the sum of these numbers in the following way :

```
0 + 1 = 1
1 + 2  = 3
3 + 3 = 6
6 + 4 = 10
10 + 5 = 15
15 + 6 = 21
21 + 7  =28
28 + 8 = 36
36 + 9 = 45
```

So, you can see that we start with 0 and 1, sum them up and add the result into next number ie 2 then again we add this result to 3 and continue like this.

Now, I will show you how recursion can be used to define logic for this requirement in a C code :

```
#include <stdio.h>

int count = 1;

void func(int sum)
{
    sum  = sum + count;
```

```
    count ++;

    if(count <= 9)
    {
        func(sum);
    }
    else
    {
        printf("\nSum is [%d] \n", sum);
    }

    return;
}

int main(void)
{
    int sum = 0;
    func(sum);
    return 0;
}
```

If you try to understand what the above code does, you will observe :

- When func() was called through main(), ‘sum’ was zero.
- For every call to func(), the value of ‘sum’ is incremented with ‘count’ (which is 1 initially), which itself gets incremented with every call.
- The condition of termination of this recursion is when value of ‘count’ exceeds 9. This is exactly what we expect.
- When ‘count’ exceeds 9, at this very moment, the value of ‘sum’ is the final figure that we want and hence the solution.

Here is another example where recursion can be used to calculate factorial of a given number :

```
#include <stdio.h>

int func(int num)
{
    int res = 0;

    if(num <= 0)
    {
        printf("\n Error \n");
    }
    else if(num == 1)
    {
        return num;
    }
```

```
    else
    {
        res  = num * func(num -1);
        return res;
    }

    return -1;

}

int main(void)
{
    int num = 5 ;
    int fact  = func(num);

    if (fact > 0)
        printf("\n The factorial of [%d] is [%d]\n", num, fact);

     return 0;
}
```

Please note that I have used hard coded number ’5′ to calculate its factorial. You can enhance this example to accept input from user.

The earlier example demonstrated only how at the final call of func() sum was calculated but the reason I used example is because it demonstrates how return values can be used produced desired results. In the example above, the call sequence across different function stacks can be visualized as :

```
res  = 5 * func(5 -1); // This is func() stack 1
res  = 4 *func(4-1);   // This is func() stack 2
res  = 3 *func(4-1);   // This is func() stack 3
res  = 2 *func(2-1);   // This is func() stack 4
return 1;              // This is func() stack 5
```

Now, substitute return value of stack 5 in stack 4, the return value of stack 4 (ie res) into stack 3 and so on. Finally, in stack 1 you will get something like

```
res = 5 * 24
```

This is 120, which is the factorial of 5, as shown in the output when you execute this recursive program.

```
$ ./recrsn
 The factorial of [5] is [120]
```

12 Interesting C Interview Questions and Answers

In this article, we will discuss some interesting problems on C language that can help students to brush up their C programming skills and help them prepare their C fundamentals for interviews.

1. gets() function

Question: There is a hidden problem with the following code. Can you detect it?

```
#include<stdio.h>

int main(void)
{
    char buff[10];
    memset(buff,0,sizeof(buff));

    gets(buff);

    printf("\n The buffer entered is [%s]\n",buff);

    return 0;
}
```

Answer: The hidden problem with the code above is the use of the function gets(). This function accepts a string from stdin without checking the capacity of buffer in which it copies the value. This may well result in buffer overflow. The standard function fgets() is advisable to use in these cases.

2. strcpy() function

Question: Following is the code for very basic password protection. Can you break it without knowing the password?

```
#include<stdio.h>

int main(int argc, char *argv[])
{
    int flag = 0;
    char passwd[10];

    memset(passwd,0,sizeof(passwd));

    strcpy(passwd, argv[1]);

    if(0 == strcmp("LinuxGeek", passwd))
    {
        flag = 1;
    }

    if(flag)
    {
```

```
        printf("\n Password cracked \n");
    }
    else
    {
        printf("\n Incorrect passwd \n");

    }
    return 0;
}
```

Answer: Yes. The authentication logic in above password protector code can be compromised by exploiting the loophole of strcpy() function. This function copies the password supplied by user to the 'passwd' buffer without checking whether the length of password supplied can be accommodated by the 'passwd' buffer or not. So if a user supplies a random password of such a length that causes buffer overflow and overwrites the memory location containing the default value '0′ of the 'flag' variable then even if the password matching condition fails, the check of flag being non-zero becomes true and hence the password protection is breached.

For example :

```
$ ./psswd aaaaaaaaaaaaa

 Password cracked
```

So you can see that though the password supplied in the above example is not correct but still it breached the password security through buffer overflow.

To avoid these kind of problems the function strncpy() should be used.

Note from author : These days the compilers internally detect the possibility of stack smashing and so they store variables on stack in such a way that stack smashing becomes very difficult. In my case also, the gcc does this by default so I had to use the the compile option '-fno-stack-protector' to reproduce the above scenario.

3. Return type of main()

Question: Will the following code compile? If yes, then is there any other problem with this code?

```
#include<stdio.h>

void main(void)
{
    char *ptr = (char*)malloc(10);

    if(NULL == ptr)
```

```
    {
        printf("\n Malloc failed \n");
        return;
    }
    else
    {
        // Do some processing

        free(ptr);
    }

    return;
}
```

Answer: The code will compile error free but with a warning (by most compilers) regarding the return type of main()function. Return type of main() should be 'int' rather than 'void'. This is because the 'int' return type lets the program to return a status value. This becomes important especially when the program is being run as a part of a script which relies on the success of the program execution.

4. Memory Leak

Question: Will the following code result in memory leak?

```
#include<stdio.h>

void main(void)
{
    char *ptr = (char*)malloc(10);

    if(NULL == ptr)
    {
        printf("\n Malloc failed \n");
        return;
    }
    else
    {
        // Do some processing
    }

    return;
}
```

Answer: Well, Though the above code is not freeing up the memory allocated to 'ptr' but still this would not cause a memory leak as after the processing is done the program exits. Since the program terminates so all the memory allocated by the program is automatically freed as part of cleanup. But if the above code was all inside a while loop then this would have caused serious memory leaks.

Note : If you want to know more on memory leaks and the tool that can detect memory leaks, read our article on Valgrind.

5. The free() function

Question: The following program seg-faults (crashes) when user supplies input as 'freeze' while it works fine with input 'zebra'. Why?

```
#include<stdio.h>

int main(int argc, char *argv[])
{
    char *ptr = (char*)malloc(10);

    if(NULL == ptr)
    {
        printf("\n Malloc failed \n");
        return -1;
    }
    else if(argc == 1)
    {
        printf("\n Usage  \n");
    }
    else
    {
        memset(ptr, 0, 10);

        strncpy(ptr, argv[1], 9);

        while(*ptr != 'z')
        {
            if(*ptr == '')
                break;
            else
                ptr++;
        }

        if(*ptr == 'z')
        {
            printf("\n String contains 'z'\n");
            // Do some more processing
        }

       free(ptr);
    }

    return 0;
}
```

Answer: The problem here is that the code changes the address in 'ptr' (by incrementing the 'ptr') inside the while loop. Now when 'zebra' is supplied as input, the while loop

terminates before executing even once and so the argument passed to free() is the same address as given by malloc(). But in case of ‘freeze’ the address held by ptr is updated inside the while loop and hence incorrect address is passed to free() which causes the seg-fault or crash.

6. atexit with _exit

Question: In the code below, the atexit() function is not being called. Can you tell why?

```
#include<stdio.h>

void func(void)
{
    printf("\n Cleanup function called \n");
    return;
}

int main(void)
{
    int i = 0;

    atexit(func);

    for(;i<0xffffff;i++);

    _exit(0);
}
```

Answer: This behavior is due to the use of function _exit(). This function does not call the clean-up functions like atexit() etc. If atexit() is required to be called then exit() or ‘return’ should be used.

7. void* and C structures

Question: Can you design a function that can accept any type of argument and returns an integer? Also, is there a way in which more than one argument can be passed to it?

Answer: A function that can accept any type of argument looks like:

```
 int func(void *ptr)
```

If more than one argument needs to be passed to this function then this function could be called with a structure object where-in the structure members can be populated with the arguments that need to be passed.

8. * and ++ operators

Question: What would be the output of the following code and why?

```
#include<stdio.h>

int main(void)
{
    char *ptr = "Linux";
    printf("\n [%c] \n",*ptr++);
    printf("\n [%c] \n",*ptr);

    return 0;
}
```

Answer: The output of the above would be :

```
[L]

[i]
```

Since the priority of both '++' and '*' are same so processing of '*ptr++' takes place from right to left. Going by this logic, ptr++ is evaluated first and then *ptr. So both these operations result in 'L'. Now since a post fix '++' was applied on ptr so the next printf() would print 'i'.

9. Making changes in Code(or read-only) segment

Question: The following code seg-faults (crashes). Can you tell the reason why?

```
#include<stdio.h>

int main(void)
{
    char *ptr = "Linux";
    *ptr = 'T';

    printf("\n [%s] \n", ptr);

    return 0;
}
```

Answer: This is because, through *ptr = 'T', the code is trying to change the first byte of the string 'Linux' kept in the code (or the read-only) segment in the memory. This operation is invalid and hence causes a seg-fault or a crash.

10. Process that changes its own name

Question: Can you write a program that changes its own name when run?

Answer: Following piece of code tries to do the required :

```
#include<stdio.h>

int main(int argc, char *argv[])
```

```
{
    int i = 0;
    char buff[100];

    memset(buff,0,sizeof(buff));

    strncpy(buff, argv[0], sizeof(buff));
    memset(argv[0],0,strlen(buff));

    strncpy(argv[0], "NewName", 7);

    // Simulate a wait. Check the process
    // name at this point.
    for(;i<0xffffffff;i++);

    return 0;
}
```

11. Returning address of local variable

Question: Is there any problem with the following code? If yes, then how it can be rectified?

```
#include<stdio.h>

int* inc(int val)
{
  int a = val;
  a++;
  return &a;
}

int main(void)
{
    int a = 10;

    int *val = inc(a);

    printf("\n Incremented value is equal to [%d] \n", *val);

    return 0;
}
```

Answer: Though the above program may run perfectly fine at times but there is a serious loophole in the function ‘inc()’. This function returns the address of a local variable. Since the life time of this local variable is that of the function ‘inc()’ so after inc() is done with its processing, using the address of its local variable can cause undesired results. This can be avoided by passing the address of variable ‘a’ from main() and then inside changes can be made to the value kept at this address.

12. Processing printf() arguments

Question: What would be the output of the following code?

```
#include<stdio.h>

int main(void)
{
    int a = 10, b = 20, c = 30;

    printf("\n %d..%d..%d \n", a+b+c, (b = b*2), (c = c*2));

    return 0;
}
```

Answer: The output of the above code would be :

```
110..40..60
```

This is because the arguments to the function are processed from right to left but are printed from left to right.

How to Use C Structures, Unions and Bit Fields with Examples

Structures, Unions and Bit fields are some of the important aspects of C programming language.

While structures are widely used, unions and bit fields are comparatively less used but that does not undermine their importance.

In this tutorial we will explain the concept of Structures, Unions and Bit fields in C language using examples.

1. Structures in C

Structure provides a way to store multiple variables of similar or different types under one umbrella. This makes information more packaged and program more modular as different variables referring to different values can be accessed through a single structure object.

An example of a C structure can be :

```
struct <Name or Tag>
     {
          <member-1>;
          <member-2>;
          <member-3>;
          ...
          ...
          ...
     };
```

So we see that a structure can be defined through a keyword ‘struct’ followed by structure name. The body of structure consists of different semicolon terminated variable definitions within curly braces.

Going back to what structure really is, A structure usually does not package unrelated variables. All the variables are usually part of some broader level information that structure intends to hold.

For example, a structure can hold all the information related to an employee in an organization:

```
struct employee
```

```
    {
       char *name;
       int age;
       char *department;
       int salary;
       char *job_title;
    };
```

Now, to access structure variable, you need to define an object for that structure. For example, here is how you can define an object for the ‘employee’ structure :

```
struct employee emp_obj;
```

NOTE: The keyword ‘struct’ is mandatory while defining structure objects in C

The variable ‘emp_obj’ now becomes object of ‘employee’ structure. Individual structure members can be accessed in the following way:

```
    emp_obj.name
    emp_obj.age
    ...
    ...
    ...
```

So we see that ‘.’ is used to access individual variables

Unlike the one above, a structure object can also be of a pointer type. For example:

```
struct employee *emp_obj;
```

In this case, individual structure members can be accessed in the following way:

```
    emp_obj->name
    emp_obj->age
    ...
    ...
    ...
```

So we see that ‘->’ is used to access individual variables.

Here is a working example of C structure:

```
#include <stdio.h>

struct employee
{
   char *name;
   int age;
   char *department;
   int salary;
```

```
   char *job_title;
};

int main(void)
{
   struct employee emp_obj;
   struct employee *emp_ptr_obj;

   emp_obj.name = "theGeekStuff";
   /* Similarly Initialize other
    * variables of employee
    * structure here */

   emp_ptr_obj = &emp_obj;

   printf("\n Employee name is [%s]\n", emp_ptr_obj->name);
   return 0;
}
```

Here is the output :

```
Employee name is [theGeekStuff]
```

2. Unions in C

Unions are almost like structures in C (just explained above) but with a twist. The twist is that the memory for a union is equal to the size of it's largest member. Confused? No worries, lets understand it in more detail.

Here is how Unions are defined:

```
union char_and_ascii
{
   char ch;
   unsigned int ascii_val;
};
```

As you can see that it's more or less like how we declare structures in C. Just that the keyword 'union' is used instead of 'struct'.

So, what is the difference between a structure and a union? Well, the difference lies in the size. If the above example would have been a structure, the size of structure would have been:

sizeof(char) + sizeof(unsigned int)

ie 1 + 4 = 5 bytes.

But, in case of a union, the size is equivalent to that of the largest member type in union. So, in this case, the largest type is ‘unsigned int’ and hence the size of union becomes ’4′.

Now, having understood that, one might ask, in which scenarios union can be used. Well, there are certain scenarios where you want to use only one of the members at a time. So in that case, using a union is a wise option rather than using a structure. This will save you memory.

Here is a working example of a Union in C :

```
#include <stdio.h>

union char_and_ascii
{
    char ch;
    unsigned short ascii_val;
};

int main (void)
{
    union char_and_ascii obj;
    obj.ascii_val = 0;

    obj.ch = 'A';

    printf("\n character = [%c], ascii_value = [%u]\n", obj.ch,
obj.ascii_val);
    return 0;
}
```

Here is the output :

```
character = [A], ascii_value = [65]
```

On a different note, to get deeper understanding of C language, you should also know how C Macros / Inline Functions and C Binary Tree work.

3. Bit fields in C

There are times when the member variables of a structure represent some flags that store either 0 or 1. Here is an example:

```
struct info
{
    int isMemoryFreed;
    int isObjectAllocated;
}
```

If you observe, though a value of 0 or 1 would be stored in these variables but the memory used would be complete 8 bytes. To reduce memory consumption when it is known that only some bits would be used for a variable, the concept of bit fields can be used.

Bit fields allow efficient packaging of data in the memory. Here is how bit fields are defined:

```
struct info
{
    int isMemoryFreed : 1;
    int isObjectAllocated : 1;
}
```

The above declaration tells the compiler that only 1 bit each from the two variables would be used. After seeing this, the compiler reduces the memory size of the structure.

Here is an example that illustrates this:

```
#include <stdio.h>

struct example1
{
    int isMemoryAllocated;
    int isObjectAllocated;
};

struct example2
{
    int isMemoryAllocated : 1;
    int isObjectAllocated : 1;
};

int main(void)
{
    printf("\n sizeof example1 is [%u], sizeof example2 is [%u]\n",
sizeof(struct example1), sizeof(struct example2));

    return 0;
}
```

Here is the output :

```
sizeof example1 is [8], sizeof example2 is [4]
```

Also, if after declaring the bit field width (1 in case of above example), if you try to access other bits then compiler would not allow you to do the same.

Here is an example:

```
#include <stdio.h>

struct example2
{
    int isMemoryAllocated : 1;
    int isObjectAllocated : 1;
};

int main(void)
{
    struct example2 obj;

    obj.isMemoryAllocated = 2;

   return 0;
}
```

So, by setting the value to ’2′, we try to access more than 1 bits. Here is what compiler complains :

```
$ gcc -Wall bitf.c -o bitf
bitf.c: In function ‘main’:
bitf.c:14:5: warning: overflow in implicit constant conversion [-
Woverflow]
```

So we see that compiler effectively treats the variables size as 1 bit only.

Buffer Overflow Attack Explained with a C Program Example

Buffer overflow attacks have been there for a long time. It still exists today partly because of programmer's carelessness while writing a code. The reason I said 'partly' because sometimes a well written code can be exploited with buffer overflow attacks, as it also depends upon the dedication and intelligence level of the attacker.

The least we can do is to avoid writing bad code that gives a chance to even script kiddies to attack your program and exploit it.

In this buffer overflow tutorial, we will discuss the basics of the following:

- What is buffer overflow?
- How a buffer overflow happens?
- How a buffer overflow attack takes place?
- How to avoid buffer overrun?

We'll keep the explanation and examples simple enough for you to understand the concept completely. We'll also use C programming language to explain the buffer overflow concept.

What is Buffer Overflow?

A buffer, in terms of a program in execution, can be thought of as a region of computer's main memory that has certain boundaries in context with the program variable that references this memory.

For example:

```
char buff[10]
```

In the above example, 'buff' represents an array of 10 bytes where buff[0] is the left boundary and buff[9] is the right boundary of the buffer.

Lets take another example:

```
int arr[10]
```

In the above example, 'arr' represents an array of 10 integers. Now assuming that the size of integer is 4 bytes, the total buffer size of 'arr' is 10*4 = 40 bytes. Similar to the first example, arr[0] refers to the left boundary while arr[9] refers to the right boundary.

By now it should be clear what a buffer means. Moving on lets understand when a buffer overflows.

A buffer is said to be overflown when the data (meant to be written into memory buffer) gets written past the left or the right boundary of the buffer. This way the data gets written to a portion of memory which does not belong to the program variable that references the buffer.

Here is an example :

```
char buff[10];
buff[10] = 'a';
```

In the above example, we declared an array of size 10 bytes. Please note that index 0 to index 9 can used to refer these 10 bytes of buffer. But, in the next line, we index 10 was used to store the value ‘a’. This is the point where buffer overrun happens because data gets written beyond the right boundary of the buffer.

It is also important for you to understand how GCC compilation process works to create a C executable.

Why are buffer overflows harmful?

Some of us may think that though a buffer overflow is a bad programming practice but so is an unused variable on stack, then why there is so much hullabaloo around it? What is the harm buffer overrun can cause to the application?

Well, if in one line we have to summarize the answer to these questions then it would be :

Buffer overflows, if undetected, can cause your program to crash or produce unexpected results.

Lets understand a couple of scenarios which justify the answer mentioned above.

1. Consider a scenario where you have allocated 10 bytes on heap memory:

```
char *ptr  = (char*) malloc(10);
```

Now, if you try to do something like this :

```
ptr[10] = 'c';
```

Then this may lead to crash in most of the cases. The reason being, a pointer is not allowed to access heap memory that does not belong to it.

2. *Consider another scenario where you try to fill a buffer (on stack) beyond it's capacity :*

```
char buff[10] = {0};

strcpy(buff, "This String Will Overflow the Buffer");
```

As you can see that the strcpy() function will write the complete string in the array 'buff' but as the size of 'buff' is less than the size of string so the data will get written past the right boundary of array 'buff'. Now, depending on the compiler you are using, chances are high that this will get unnoticed during compilation and would not crash during execution. The simple reason being that stack memory belongs to program so any buffer overflow in this memory could get unnoticed.

So in these kind of scenarios, buffer over flow quietly corrupts the neighbouring memory and if the corrupted memory is being used by the program then it can cause unexpected results.

You also need to understand how you can prevent stack smashing attacks with GCC.

Buffer Overflow Attacks

Until now we discussed about what buffer overflows can do to your programs. We learned how a program could crash or give unexpected results due to buffer overflows. Horrifying isn't it ? But, that it is not the worst part.

It gets worse when an attacker comes to know about a buffer over flow in your program and he/she exploits it. Confused? Consider this example :

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char buff[15];
    int pass = 0;

    printf("\n Enter the password : \n");
    gets(buff);

    if(strcmp(buff, "thegeekstuff"))
```

```
    {
        printf ("\n Wrong Password \n");
    }
    else
    {
        printf ("\n Correct Password \n");
        pass = 1;
    }

    if(pass)
    {
       /* Now Give root or admin rights to user*/
        printf ("\n Root privileges given to the user \n");
    }

    return 0;
}
```

The program above simulates scenario where a program expects a password from user and if the password is correct then it grants root privileges to the user.

Let's the run the program with correct password ie 'thegeekstuff' :

```
$ ./bfrovrflw

 Enter the password :
thegeekstuff

 Correct Password

 Root privileges given to the user
```

This works as expected. The passwords match and root privileges are given.

But do you know that there is a possibility of buffer overflow in this program. The gets() function does not check the array bounds and can even write string of length greater than the size of the buffer to which the string is written. Now, can you even imagine what can an attacker do with this kind of a loophole?

Here is an example :

```
$ ./bfrovrflw

 Enter the password :
hhhhhhhhhhhhhhhhhhhh

 Wrong Password


 Root privileges given to the user
```

In the above example, even after entering a wrong password, the program worked as if you gave the correct password.

There is a logic behind the output above. What attacker did was, he/she supplied an input of length greater than what buffer can hold and at a particular length of input the buffer overflow so took place that it overwrote the memory of integer ‘pass’. So despite of a wrong password, the value of ‘pass’ became non zero and hence root privileges were granted to an attacker.

There are several other advanced techniques (like code injection and execution) through which buffer over flow attacks can be done but it is always important to first know about the basics of buffer, it’s overflow and why it is harmful.

To avoid buffer overflow attacks, the general advice that is given to programmers is to follow good programming practices. For example:

- Make sure that the memory auditing is done properly in the program using utilities likevalgrind memcheck
- Use fgets() instead of gets().
- Use strncmp() instead of strcmp(), strncpy() instead of strcpy() and so on.

Enumeration (or enum) in C

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

The keyword ‘enum’ is used to declare new enumeration types in C and C++. Following is an example of enum declaration.

```
enum State {Working = 1, Failed = 0};
```

Following are some interesting facts about initialization of enum.

1. Two enum names can have same value. For example, in the following C program both ‘Failed’ and ‘Freezed’ have same value 0.

```
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};

int main()
{
   printf("%d, %d, %d", Working, Failed, Freezed);
   return 0;
}
```

Output:

```
1, 0, 0
```

2. If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.

```
#include <stdio.h>
enum day {sunday, monday, tuesday, wednesday, thursday, friday,
saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Output:

```
The day number stored in d is 4
```

3. We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

```
#include <stdio.h>
enum day {sunday = 1, monday, tuesday = 5,
          wednesday, thursday = 10, friday, saturday};

int main()
{
    printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,
            wednesday, thursday, friday, saturday);
    return 0;
}
```

Output:

```
1 2 5 6 10 11 12
```

4. The value assigned to enum names must be some integeral constant, i.e., the value must be in range from minimum possible integer value to maximum possible integer value.

5. All enum constants must be unique in their scope. For example, the following program fails in compilation.

```
enum state  {working, failed};
enum result {failed, passed};

int main()  { return 0; }
```

Output:

```
Compile Error: 'failed' has a previous declaration as 'state failed'
```

Exercise:

Predict the output of following C programs

Program 1:

```
#include <stdio.h>
enum  day  {sunday  =  1,  tuesday,  wednesday,  thursday,  friday,
saturday};

int main()
{
    enum day d = thursday;
    printf("The day number stored in d is %d", d);
    return 0;
}
```

Program 2:

```
#include <stdio.h>
```

```
enum State {WORKING = 0, FAILED, FREEZED};
enum State currState = 2;

enum State FindState() {
    return currState;
}

int main() {
   (FindState()   ==   WORKING)?   printf("WORKING"):   printf("NOT
WORKING");
   return 0;
}
```

Enum vs Macro

We can also use macros to define names constants. For example we can define ‘Working’ and ‘Failed’ using following macro.

```
#define Working 0
#define Failed 1
#define Freezed 2
```

There are multiple advantages of using enum over macro when many related named constants have integral values.

a) Enums follow scope rules.
b) Enum variables are automatically assigned values.

Following is simpler

```
enum state  {Working, Failed, Freezed};
```

Structures in C

What is a structure?

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.

How to create a structure?

‘struct’ keyword is used to create a structure. Following is an example.

```
struct addrress
{
   char name[50];
```

```
   char street[100];
   char city[50];
   char state[20]
   int pin;
};
```

How to declare structure variables?

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

```
// A variable declaration with structure declaration.
struct Point
{
   int x, y;
} p1;  // The variable p1 is declared with 'Point'


// A variable declaration like basic data types
struct Point
{
   int x, y;
};

int main()
{
   struct Point p1;  // The variable p1 is declared like a normal
variable
}
```

Note: In C++, the struct keyword is optional before in declaration of variable. In C, it is mandatory.

How to initialize structure members?

Structure members cannot be initialized with declaration. For example the following C program fails in compilation.

```
struct Point
{
   int x = 0;  // COMPILER ERROR:  cannot initialize members here
   int y = 0;  // COMPILER ERROR:  cannot initialize members here
};
```

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members can be initialized using curly braces ‘{}’. For example, following is a valid initialization.

```
struct Point
{
   int x, y;
};

int main()
{
   // A valid initialization. member x gets value 0 and y
   // gets value 1.  The order of declaration is followed.
   struct Point p1 = {0, 1};
}
```

How to access structure elements?

Structure members are accessed using dot (.) operator.

```
struct Point
{
   int x, y;
};

int main()
{
   struct Point p1 = {0, 1};

   // Accesing members of point p1
   p1.x = 20;
   printf ("x = %d, y = %d", p1.x, p1.y);

   return 0;
}
```

Output:

```
20  1
```

What is designated Initialization?

Designated Initialization allows structure members to be initialized in any order. This feature has been added in C99 standard.

```
struct Point
{
   int x, y, z;
};

int main()
{
   // Examples of initializtion using designated initialization
   struct Point p1 = {.y = 0, .z = 1, .x = 2};
   struct Point p2 = {.x = 20};
```

```
   printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);
   printf ("x = %d", p2.x);
   return 0;
}
```

Output:

```
x = 2, y = 0, z = 1
x = 20
```

This feature is not available in C++ and works only in C.

What is an array of structures?

Like other primitive data types, we can create an array of structures.

```
struct Point

{

   int x, y;

};



int main()

{

   // Create an array of structures

   struct Point arr[10];



   // Access array members

   arr[0].x = 10;

   arr[0].y = 20;



   printf("%d %d", arr[0].x, arr[0].y);

   return 0;
```

```
}
```

Output:

```
10   20
```

What is a structure pointer?

Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow (->) operator.

```
struct Point
{
   int x, y;
};


int main()
{
   struct Point p1 = {1, 2};


   // p2 is a pointer to structure p1
   struct Point *p2 = &p1;


   // Accessing structure members using structure pointer
   printf("%d %d", p2->x, p2->y);
   return 0;
}
```

Output:

```
1   2
```

What is structure member alignment?
See http://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/

We will soon be discussing union and other struct related topics in C. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Union in C

Like Structures, union is a user defined data type. In union, all members share the same memory location. For example in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.

```
#include <stdio.h>


// Declaration of union is same as structures
union test
{
   int x, y;
};


int main()
{
    // A union variable t
    test t;


    t.x = 2; // t.y also gets value 2
    printf ("After making x = 2:\n x = %d, y = %d\n\n",
             t.x, t.y);
```

```
    t.y = 10;  // t.x is also updated to 10
    printf ("After making Y = 'A':\n x = %d, y = %d\n\n",
             t.x, t.y);
    return 0;
}
```

Output:

```
After making x = 2:
 x = 2, y = 2

After making Y = 'A':
 x = 10, y = 10
```

How is the size of union decided by compiler? Size of a union is taken according the size of largest member in union.

```
#include <stdio.h>


union test1
{
   int x;
   int y;
};


union test2
{
   int x;
   char y;
```

```
};


union test3
{
   int arr[10];
   char y;
};


int main()
{
    printf ("sizeof(test1) = %d, sizeof(test2) = %d,"
            "sizeof(test3) =  %d", sizeof(test1),
            sizeof(test2), sizeof(test3));
    return 0;
}
```

Output

```
sizeof(test1) = 4, sizeof(test2) = 4,sizeof(test3) =  40
```

Pointers to unions?

Like structures, we can have pointers to unions and can access members using arrow operator (->). The following example demonstrates the same.

```
union test
{
   int x;
   char y;
```

```
};


int main()
{
   union test p1;
   p1.x = 65;


   // p2 is a pointer to union p1
   union test *p2 = &p1;


   // Accessing union members using pointer
   printf("%d %c", p2->x, p2->y);
   return 0;
}
```

```
65 A
```

What are applications of union?
Unions can be useful in many situations where we want to use same memory for two ore more members. For example, suppose we want to implement a binary tree data structure where each leaf node has a double data value, while each internal node has pointers to two children, but no data. If we declare this as:

```
struct NODE {
  struct NODE *left;
  struct NODE *right;
  double data;
};
```

then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as following, then we can save space.

```
struct NODE

{

    bool is_leaf;

    union

    {

        struct

        {

            struct NODE *left;

            struct NODE *right;

        } internal;

        double data;

    } info;

};
```

The above example is taken from Computer Systems : A Programmer's Perspective (English) 2nd Edition book.

References:
http://en.wikipedia.org/wiki/Union_type
Computer Systems : A Programmer's Perspective (English) 2nd Edition

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Struct Hack

What will be the size of following structure?

```
struct employee
```

```
{
    int   emp_id;
    int   name_len;
    char   name[0];
};
```

4 + 4 + 0 = 8 bytes.

And what about size of “name[0]”. In gcc, when we create an array of zero length, it is considered as array of incomplete type that’s why gcc reports its size as “0” bytes. This technique is known as “Stuct Hack”. When we create array of zero length inside structure, it must be (and only) last member of structure. Shortly we will see how to use it. “Struct Hack” technique is used to create variable length member in a structure. In the above structure, string length of “name” is not fixed, so we can use “name” as variable length array.

Let us see below memory allocation.

```
struct employee *e = malloc(sizeof(*e) + sizeof(char) * 128);
```

is equivalent to

```
struct employee
{
    int   emp_id;
    int   name_len;
    char   name[128]; /* character array of size 128 */
};
```

And below memory allocation

```
struct employee *e = malloc(sizeof(*e) + sizeof(char) * 1024);
```

is equivalent to

```
struct employee

{

    int   emp_id;

    int   name_len;

    char  name[1024]; /* character array of size 1024 */

};
```

Note: since name is character array, in malloc instead of “sizeof(char) * 128″, we can use “128” directly. sizeof is used to avoid confusion.

Now we can use “name” same as pointer. e.g.

```
e->emp_id      = 100;
e->name_len    = strlen("Geeks For Geeks");
strncpy(e->name, "Geeks For Geeks", e->name_len);
```

When we allocate memory as given above, compiler will allocate memory to store “emp_id” and “name_len” plus contiguous memory to store “name”. When we use this technique, gcc guaranties that, “name” will get contiguous memory.
Obviously there are other ways to solve problem, one is we can use character pointer. But there is no guarantee that character pointer will get contiguous memory, and we can take advantage of this contiguous memory. For example, by using this technique, we can allocate and deallocate memory by using single malloc and free call (because memory is contagious). Other advantage of this is, suppose if we want to write data, we can write whole data by using single “write()” call. e.g.

```
write(fd, e, sizeof(*e) + name_len); /* write emp_id + name_len + name */
```

If we use character pointer, then we need 2 write calls to write data. e.g.

```
write(fd, e, sizeof(*e));          /* write emp_id
+ name_len */
write(fd, e->name, e->name_len);  /* write name */
```

Note: In C99, there is feature called “flexible array members”, which works same as “Struct Hack”

Structure Member Alignment, Padding and Data Packing

What do we mean by data alignment, structure packing and padding?

Predict the output of following program.

```
#include <stdio.h>


// Alignment requirements

// (typical 32 bit machine)


// char         1 byte

// short int    2 bytes

// int          4 bytes

// double       8 bytes


// structure A

typedef struct structa_tag

{

   char     c;
```

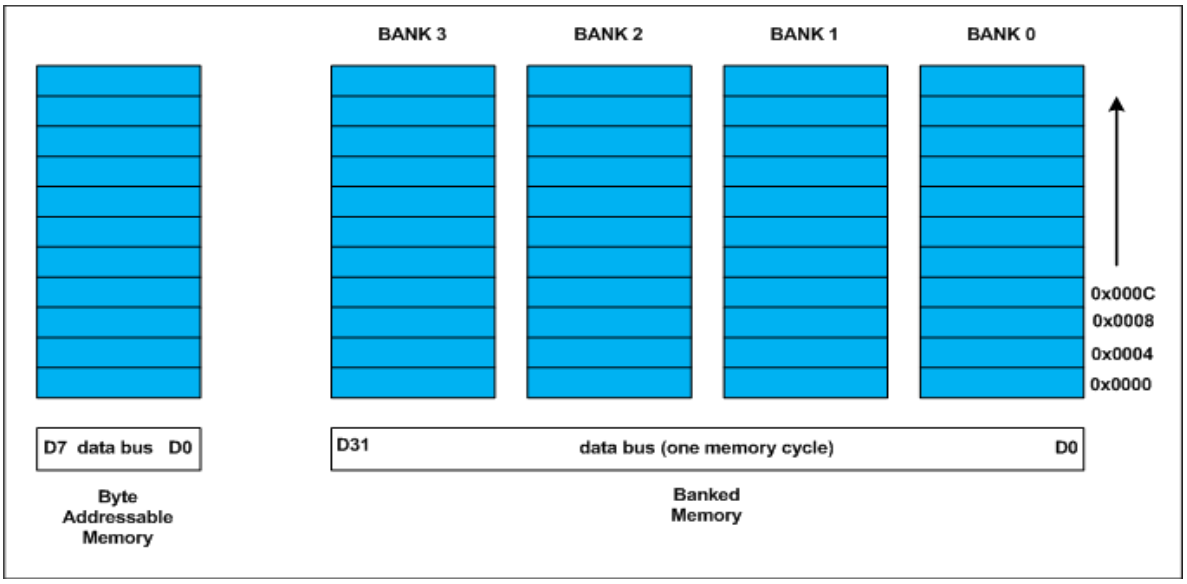
```
   short int  s;
} structa_t;

// structure B
typedef struct structb_tag
{
   short int  s;
   char     c;
   int      i;
} structb_t;

// structure C
typedef struct structc_tag
{
   char     c;
   double    d;
   int      s;
} structc_t;

// structure D
typedef struct structd_tag
{
   double    d;
   int      s;
   char     c;
```

```
} structd_t;


int main()
{
   printf("sizeof(structa_t) = %d\n", sizeof(structa_t));
   printf("sizeof(structb_t) = %d\n", sizeof(structb_t));
   printf("sizeof(structc_t) = %d\n", sizeof(structc_t));
   printf("sizeof(structd_t) = %d\n", sizeof(structd_t));


   return 0;
}
```

Before moving further, write down your answer on a paper, and read on. If you urge to see explanation, you may miss to understand any lacuna in your analogy. Also read the post by Kartik.

Data Alignment:

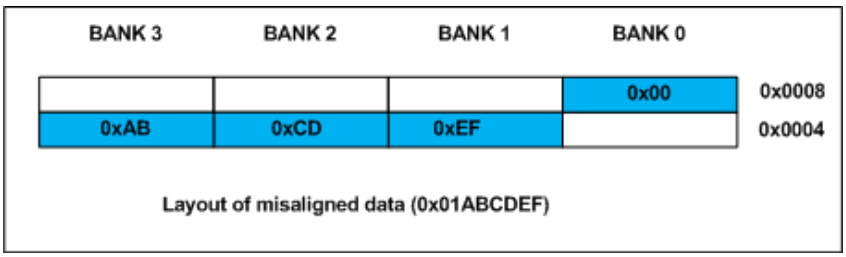
Every data type in C/C++ will have alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.



Historically memory is byte addressable and arranged sequentially. If the memory is arranged as single bank of one byte width, the processor needs to issue 4 memory read cycles to fetch an integer. It is more economical to read all 4 bytes of integer in one memory cycle. To take such advantage, the memory will be arranged as group of 4 banks as shown in the above figure.

The memory addressing still be sequential. If bank 0 occupies an address X, bank 1, bank 2 and bank 3 will be at (X + 1), (X + 2) and (X + 3) addresses. If an integer of 4 bytes is allocated on X address (X is multiple of 4), the processor needs only one memory cycle to read entire integer.

Where as, if the integer is allocated at an address other than multiple of 4, it spans across two rows of the banks as shown in the below figure. Such an integer requires two memory read cycle to fetch the data.



A variable's ***data alignment*** deals with the way the data stored in these banks. For example, the natural alignment of ***int*** on 32-bit machine is 4 bytes. When a data type is naturally aligned, the CPU fetches it in minimum read cycles.

Similarly, the natural alignment of ***short int*** is 2 bytes. It means, a ***short int*** can be stored in bank 0 – bank 1 pair or bank 2 – bank 3 pair. A ***double*** requires 8 bytes, and occupies two rows in the memory banks. Any misalignment of ***double*** will force more than two read cycles to fetch***double*** data.

Note that a **double** variable will be allocated on 8 byte boundary on 32 bit machine and requires two memory read cycles. On a 64 bit machine, based on number of banks, **double** variable will be allocated on 8 byte boundary and requires only one memory read cycle.

Structure Padding:

In C/C++ a structures are used as data pack. It doesn't provide any data encapsulation or data hiding features (C++ case is an exception due to its semantic similarity with classes).

Because of the alignment requirements of various data types, every member of structure should be naturally aligned. The members of structure allocated sequentially increasing order. Let us analyze each struct declared in the above program.

Output of Above Program:

For the sake of convenience, assume every structure type variable is allocated on 4 byte boundary (say 0x0000), i.e. the base address of structure is multiple of 4 (need not necessary always, see explanation of structc_t).

structure A

The *structa_t* first element is *char* which is one byte aligned, followed by *short int*. short int is 2 byte aligned. If the the short int element is

immediately allocated after the char element, it will start at an odd address boundary. The compiler will insert a padding byte after the char to ensure short int will have an address multiple of 2 (i.e. 2 byte aligned). The total size of structa_t will be sizeof(char) + 1 (padding) + sizeof(short), 1 + 1 + 2 = 4 bytes.

structure B

The first member of *structb_t* is short int followed by char. Since char can be on any byte boundary no padding required in between short int and char, on total they occupy 3 bytes. The next member is int. If the int is allocated immediately, it will start at an odd byte boundary. We need 1 byte padding after the char member to make the address of next int member is 4 byte aligned. On total, the *structb_t* requires 2 + 1 + 1 (padding) + 4 = 8 bytes.

structure C – Every structure will also have alignment requirements

Applying same analysis, *structc_t* needs sizeof(char) + 7 byte padding + sizeof(double) + sizeof(int) = 1 + 7 + 8 + 4 = 20 bytes. However, the sizeof(structc_t) will be 24 bytes. It is because, along with structure members, structure type variables will also have natural alignment. Let us understand it by an example. Say, we declared an array of structc_t as shown below

```
structc_t structc_array[3];
```

Assume, the base address of *structc_array* is 0x0000 for easy calculations. If the structc_t occupies 20 (0x14) bytes as we calculated, the second structc_t array element (indexed at 1) will be at 0x0000 + 0x0014 = 0x0014. It is the start address of index 1 element of array. The double member of this structc_t will be allocated on 0x0014 + 0x1 + 0x7 = 0x001C (decimal 28) which is not multiple of 8 and conflicting with the alignment requirements of double. As we mentioned on the top, the alignment requirement of double is 8 bytes.

Inorder to avoid such misalignment, compiler will introduce alignment requirement to every structure. It will be as that of the largest member of the structure. In our case alignment of structa_t is 2, structb_t is 4 and structc_t is 8. If we need nested structures, the size of largest inner structure will be the alignment of immediate larger structure.

In structc_t of the above program, there will be padding of 4 bytes after int member to make the structure size multiple of its alignment. Thus the sizeof (structc_t) is 24 bytes. It guarantees correct alignment even in arrays. You can cross check.

structure D – How to Reduce Padding?

By now, it may be clear that padding is unavoidable. There is a way to minimize padding. The programmer should declare the structure members in their increasing/decreasing order of size. An example is structd_t given in our code, whose size is 16 bytes in lieu of 24 bytes of structc_t.

What is structure packing?

Some times it is mandatory to avoid padded bytes among the members of structure. For example, reading contents of ELF file header or BMP or JPEG file header. We need to define a structure similar to that of the header layout and map it. However, care should be exercised in accessing such members. Typically reading byte by byte is an option to avoid misaligned exceptions. There will be hit on performance.

Most of the compilers provide non standard extensions to switch off the default padding like pragmas or command line switches. Consult the documentation of respective compiler for more details.

Pointer Mishaps:

There is possibility of potential error while dealing with pointer arithmetic. For example, dereferencing a generic pointer (void *) as shown below can cause misaligned exception,

```
// Deferencing a generic pointer (not safe)
// There is no guarantee that pGeneric is integer aligned
*(int *)pGeneric;
```

It is possible above type of code in programming. If the pointer *pGeneric* is not aligned as per the requirements of casted data type, there is possibility to get misaligned exception.

Infact few processors will not have the last two bits of address decoding, and there is no way to access *misaligned* address. The processor generates misaligned exception, if the programmer tries to access such address.

A note on malloc() returned pointer

The pointer returned by malloc() is *void* *. It can be converted to any data type as per the need of programmer. The implementer of malloc() should return a pointer that is aligned to maximum size of primitive data types (those defined by compiler). It is usually aligned to 8 byte boundary on 32 bit machines.

Object File Alignment, Section Alignment, Page Alignment

These are specific to operating system implementer, compiler writers and are beyond the scope of this article. Infact, I don't have much information.

General Questions:

1. Is alignment applied for stack?

Yes. The stack is also memory. The system programmer should load the stack pointer with a memory address that is properly aligned. Generally, the processor won't check stack alignment, it is the programmer's responsibility to ensure proper alignment of stack memory. Any misalignment will cause run time surprises.

For example, if the processor word length is 32 bit, stack pointer also should be aligned to be multiple of 4 bytes.

2. If *char* data is placed in a bank other bank 0, it will be placed on wrong data lines during memory read. How the processor handles *char* type?

Usually, the processor will recognize the data type based on instruction (e.g. LDRB on ARM processor). Depending on the bank it is stored, the processor shifts the byte onto least significant data lines.

3. When arguments passed on stack, are they subjected to alignment?

Yes. The compiler helps programmer in making proper alignment. For example, if a 16-bit value is pushed onto a 32-bit wide stack, the value is automatically padded with zeros out to 32 bits. Consider the following program.

```
void argument_alignment_check( char c1, char c2 )

{

   // Considering downward stack

   // (on upward stack the output will be negative)

   printf("Displacement %d\n", (int)&c2 - (int)&c1);

}
```

The output will be 4 on a 32 bit machine. It is because each character occupies 4 bytes due to alignment requirements.

4. What will happen if we try to access a misaligned data?

It depends on processor architecture. If the access is misaligned, the processor automatically issues sufficient memory read cycles and packs the data properly onto the data bus. The penalty is on performance. Where as few processors will not have last two address lines, which means there is no-way to access odd byte boundary. Every data access must be aligned (4 bytes) properly. A misaligned access is critical exception on

such processors. If the exception is ignored, read data will be incorrect and hence the results.

5. Is there any way to query alignment requirements of a data type.

Yes. Compilers provide non standard extensions for such needs. For example, __alignof() in Visual Studio helps in getting the alignment requirements of data type. Read MSDN for details.

6. When memory reading is efficient in reading 4 bytes at a time on 32 bit machine, why should a**double** type be aligned on 8 byte boundary?

It is important to note that most of the processors will have math co-processor, called Floating Point Unit (FPU). Any floating point operation in the code will be translated into FPU instructions. The main processor is nothing to do with floating point execution. All this will be done behind the scenes.

As per standard, double type will occupy 8 bytes. And, every floating point operation performed in FPU will be of 64 bit length. Even float types will be promoted to 64 bit prior to execution.

The 64 bit length of FPU registers forces double type to be allocated on 8 byte boundary. I am assuming (I don't have concrete information) in case of FPU operations, data fetch might be different, I mean the data bus, since it goes to FPU. Hence, the address decoding will be different for double types (which is expected to be on 8 byte boundary). It means, *the address decoding circuits of floating point unit will not have last 3 pins*.

Answers:

```
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
```

Update: 1-May-2013

It is observed that on latest processors we are getting size of struct_c as 16 bytes. I yet to read relevant documentation. I will update once I got proper information (written to few experts in hardware).

On older processors (AMD Athlon X2) using same set of tools (GCC 4.7) I got struct_c size as 24 bytes. The size depends on how memory banking organized at the hardware level.

Question structure

Anonymous structs/unions is not part of the C standard, but rather a not very widespread GNU extension.

In your particular example some compilers (mainly GCC) will allow you to access manager and worker unique variables via e.g. `company[i].shares` or `company[i].department`, but `company[i].age` is ambiguous and the compiler will not know which one is meant. Your approach is similar to trying to define

```
union {
     int num;
     float num;
} union_number;
```

which is not even valid C.

there are two ways to solve this.

a) moving the shared attributes outside the struct (the evil GNU way, please don't do that, I know for a fact that icc does not even compile this)

```
union employee
{
    char key;

    struct person {
        short int age;

        union {
            struct manager
            {
                float shares;
                short int level;
            };

            struct worker
            {
                short int skill;
                short int department;
            };
```

```
        }
    };

} company[10];
```

b) or the more clean standardized way to name your structs:

```
union employee
{
    char key;

    struct manager
    {
        short int age;
        float shares;
        short int level;
    } manager;

    struct worker
    {
        short int age;
        short int skill;
        short int department;
    } worker;

} company[10];
```

in this case you will be able to access the struct elements
via `company[i].manager.age`, `company[i].worker.skill` and so on.
Please pay attention that at runtime there is no way to test whether your union contains a key, a manager or a worker. That must be known in advance.

Another thing: I am not sure if this is intended, but in your declaration you cannot save a key together with a manager or a worker. Your union contains only **one** of key, manager or worker

Operations on struct variables in C

In C, the only operation that can be applied to *struct* variables is assignment. Any other operation (e.g. equality check) is not allowed on *struct* variables.
For example, program 1 works without any error and program 2 fails in compilation.

Program 1

```
#include <stdio.h>

struct Point {
  int x;
  int y;
};

int main()
{
  struct Point p1 = {10, 20};
  struct Point p2 = p1; // works: contents of p1 are copied to p1
  printf(" p2.x = %d, p2.y = %d", p2.x, p2.y);
  getchar();
  return 0;
}
```

Program 2

```
#include <stdio.h>

struct Point {
  int x;
  int y;
};

int main()
{
  struct Point p1 = {10, 20};
  struct Point p2 = p1; // works: contents of p1 are copied to p1
  if (p1 == p2)  // compiler error: cannot do equality check for
                 // whole structures
  {
    printf("p1 and p2 are same ");
  }
  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Bit Fields in C

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is withing a small range.

For example, consider the following declaration of date without use of bit fields.

```
#include <stdio.h>

// A simple representation of date
struct date
{
   unsigned int d;
   unsigned int m;
   unsigned int y;
};

int main()
{
   printf("Size of date is %d bytes\n", sizeof(struct date));
   struct date dt = {31, 12, 2014};
   printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
}
```

Output:

```
Size of date is 12 bytes
Date is 31/12/2014
```

The above representation of ‘date’ takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

```
#include <stdio.h>

// A space optimized representation of date
struct date
{
   // d has value between 1 and 31, so 5 bits
   // are sufficient
   unsigned int d: 5;

   // m has value between 1 and 12, so 12 bits
   // are sufficient
   unsigned int m: 4;

   unsigned int y;
};

int main()
{
   printf("Size of date is %d bytes\n", sizeof(struct date));
```

```
   struct date dt = {31, 12, 2014};
   printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
   return 0;
}
```

Output:

```
Size of date is 8 bytes
Date is 31/12/2014
```

Following are some interesting facts about bit fields in C.

1) A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

```
#include <stdio.h>

// A structure without forced alignment
struct test1
{
   unsigned int x: 5;
   unsigned int y: 8;
};

// A structure with forced alignment
struct test2
{
   unsigned int x: 5;
   unsigned int: 0;
   unsigned int y: 8;
};

int main()
{
   printf("Size of test1 is %d bytes\n", sizeof(struct test1));
   printf("Size of test2 is %d bytes\n", sizeof(struct test2));
   return 0;
}
```

Output:

```
Size of test1 is 4 bytes
Size of test2 is 8 bytes
```

2) We cannot have pointers to bit field members as they may not start at a byte boundary.

```
#include <stdio.h>
struct test
{
   unsigned int x: 5;
   unsigned int y: 5;
   unsigned int z;
};
```

```
int main()
{
   test t;

   // Uncommenting the following line will make
   // the program compile and run
   printf("Address of t.x is %p", &t.x);

   // The below line works fine as z is not a
   // bit field member
   printf("Address of t.z is %p", &t.z);
   return 0;
}
```

Output:

```
 error: attempt to take address of bit-field structure member 'test::x'
```

3) It is implementation defined to assign an out-of-range value to a bit field member.

```
#include <stdio.h>
struct test
{
   unsigned int x: 2;
   unsigned int y: 2;
   unsigned int z: 2;
};
int main()
{
   test t;
   t.x = 5;
   printf("%d", t.x);
   return 0;
}
```

Output:

```
Implementation-Dependent
```

4) In C++, we can have static members in a structure/class, but bit fields cannot be static.

```
// The below C++ program compiles and runs fine
struct test1 {
   static unsigned int x;
};
int main() {  }


// But below C++ program fails in compilation as bit fields
// cannot be static
struct test1 {
   static unsigned int x: 5;
};
int main() {  }
```

```
// error: static member 'x' cannot be a bit-field
```

5) Array of bit fields is not allowed. For example, the below program fails in compilation.

```
struct test
{
  unsigned int x[10]: 5;
};

int main()
{

}
```

Output:

```
error: bit-field 'x' has invalid type
```

Exercise:

Predict the output of following programs. Assume that unsigned int takes 4 bytes and long int takes 8 bytes.

1)

```
#include <stdio.h>
struct test
{
   unsigned int x;
   unsigned int y: 33;
   unsigned int z;
};
int main()
{
   printf("%d", sizeof(struct test));
   return 0;
}
```

2)

```
#include <stdio.h>
struct test
{
   unsigned int x;
   long int y: 33;
   unsigned int z;
};
int main()
{
   struct test t;
   unsigned int *ptr1 = &t.x;
   unsigned int *ptr2 = &t.z;
   printf("%d", ptr2 - ptr1);
```

```
   return 0;
}
```

3)

```
#include <stdio.h>
union test
{
  unsigned int x: 3;
  unsigned int y: 3;
  int z;
};

int main()
{
   union test t;
   t.x = 5;
   t.y = 4;
   t.z = 1;
   printf("t.x = %d, t.y = %d, t.z = %d",
           t.x, t.y, t.z);
   return 0;
}
```

4) Use bit fields in C to figure out a way whether a machine is little endian or big endian.

Count set bits in an integer

Write an efficient program to count number of 1s in binary representation of an integer.

1. Simple Method Loop through all bits in an integer, check if a bit is set and if it is then increment the set bit count. See below program.

```
/* Function to get no of set bits in binary representation of passed
binary no. */
int countSetBits(unsigned int n)
{
  unsigned int count = 0;
  while(n)
  {
    count += n & 1;
    n >>= 1;
  }
  return count;
}

/* Program to test function countSetBits */
int main()
{
```

```
    int i = 9;
    printf("%d", countSetBits(i));
    getchar();
    return 0;
}
```

Time Complexity: (-)(logn) (Theta of logn)

2. Brian Kernighan's Algorithm:

Subtraction of 1 from a number toggles all the bits (from right to left) till the rightmost set bit(including the righmost set bit). So if we subtract a number by 1 and do bitwise & with itself (n & (n-1)), we unset the righmost set bit. If we do n & (n-1) in a loop and count the no of times loop executes we get the set bit count. Beauty of the this solution is number of times it loops is equal to the number of set bits in a given integer.

```
   1  Initialize count: = 0
   2  If integer n is not zero
      (a) Do bitwise & with (n-1) and assign the value back to n
          n: = n&(n-1)
      (b) Increment count by 1
      (c) go to step 2
   3  Else return count
```

Implementation of Brian Kernighan's Algorithm:

```
#include<stdio.h>

/* Function to get no of set bits in binary
   representation of passed binary no. */
int countSetBits(int n)
{
    unsigned int count = 0;
    while (n)
    {
      n &= (n-1) ;
      count++;
    }
    return count;
}

/* Program to test function countSetBits */
int main()
{
    int i = 9;
    printf("%d", countSetBits(i));
    getchar();
    return 0;
```

```
}
```

Example for Brian Kernighan's Algorithm:

```
   n =  9 (1001)
   count = 0

   Since 9 > 0, subtract by 1 and do bitwise & with (9-1)
   n = 9&8  (1001 & 1000)
   n = 8
   count  = 1

   Since 8 > 0, subtract by 1 and do bitwise & with (8-1)
   n = 8&7  (1000 & 0111)
   n = 0
   count = 2

   Since n = 0, return count which is 2 now.
```

Time Complexity: O(logn)

http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable

Swap two nibbles in a byte

A nibble is a four-bit aggregation, or half an octet. There are two nibbles in a byte. Given a byte, swap the two nibbles in it. For example 100 is be represented as 01100100 in a byte (or 8 bits). The two nibbles are (0110) and (0100). If we swap the two nibbles, we get 01000110 which is 70 in decimal.

To swap the nibbles, we can use bitwise &, bitwise '<<' and '>>' operators. A byte can be represented using a unsigned char in C as size of char is 1 byte in a typical C compiler. Following is C program to swap the two nibbles in a byte.

```
#include <stdio.h>

unsigned char swapNibbles(unsigned char x)
{
    return ( (x & 0x0F)<<4 | (x & 0xF0)>>4 );
}

int main()
{
    unsigned char x = 100;
    printf("%u", swapNibbles(x));
```

```
    return 0;
}
```

Output:

```
70
```

Explanation:
100 is 01100100 in binary. The operation can be split mainly in two parts.
1) The expression “x & 0x0F” gives us last 4 bits of x. For x = 100, the result is 00000100. Using bitwise ‘<<' operator, we shift the last four bits to the left 4 times and make the new last four bits as 0. The result after shift is 01000000.

2) The expression “x & 0xF0” gives us first four bits of x. For x = 100, the result is 01100000. Using bitwise ‘>>’ operator, we shift the digit to the right 4 times and make the first four bits as 0. The result after shift is 00000110.

At the end we use the bitwise OR ‘|’ operation of the two expressions explained above. The OR operator places first nibble to the end and last nibble to first. For x = 100, the value of (01000000) OR (00000110) gives the result 01000110 which is equal to 70 in decimal.

Setting a bit

Use the bitwise OR operator (|) to set a bit.

```
number |= 1 << x;
```

That will set bit x.

Clearing a bit

Use the bitwise AND operator (&) to clear a bit.

```
number &= ~(1 << x);
```

That will clear bit x. You must invert the bit string with the bitwise NOT operator (~), then AND it.

Toggling a bit

The XOR operator (^) can be used to toggle a bit.

```
number ^= 1 << x;
```

That will toggle bit x.

Checking a bit

You didn't ask for this but I might as well add it.

To check a bit, shift the number x to the right, then bitwise AND it:

```
bit = (number >> x) & 1;
```

That will put the value of bit x into the variable bit.

Changing the nth bit to x

Setting the nth bit to either 1 or 0 can be achieved with the following:

```
number ^= (-x ^ number) & (1 << n);
```

Bit n will be set if x is 1, and cleared if x is 0.

Find position of the only set bit

Given a number having only one ‘1’ and all other ’0’s in its binary representation, find position of the only set bit.

The idea is to start from rightmost bit and one by one check value of every bit. Following is detailed algorithm.

1) If number is power of two then and then only its binary representation contains only one ‘1’. That’s why check whether given number is power of 2 or not. If given number is not power of 2, then print error message and exit.

2) Initialize two variables; i = 1 (for looping) and pos = 1 (to find position of set bit)

3) Inside loop, do bitwise AND of i and number ‘N’. If value of this operation is true, then “pos” bit is set, so break the loop and return position. Otherwise, increment “pos” by 1 and left shift i by 1 and repeat the procedure.

```
// C program to find position of only set bit in a given number
```

```
#include <stdio.h>

// A utility function to check whether n is power of 2 or not.
int isPowerOfTwo(unsigned n)
{  return n && (! (n & (n-1)) ); }

// Returns position of the only set bit in 'n'
int findPosition(unsigned n)
{
    if (!isPowerOfTwo(n))
        return -1;

    unsigned i = 1, pos = 1;

    // Iterate through bits of n till we find a set bit
    // i&n will be non-zero only when 'i' and 'n' have a set bit
    // at same position
    while (!(i & n))
    {
        // Unset current bit and set the next bit in 'i'
        i = i << 1;

        // increment position
        ++pos;
    }

    return pos;
}

// Driver program to test above function
int main(void)
{
    int n = 16;
    int pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    return 0;
}
```

Output:

```
n = 16, Position 5
n = 12, Invalid number
```

```
n = 128, Position 8
```

Following is another method for this problem. The idea is to one by one right shift the set bit of given number ‘n’ until ‘n’ becomes 0. Count how many times we shifted to make ‘n’ zero. The final count is position of the set bit.

```
// C program to find position of only set bit in a given number
#include <stdio.h>

// A utility function to check whether n is power of 2 or not
int isPowerOfTwo(unsigned n)
{  return n && (! (n & (n-1)) ); }

// Returns position of the only set bit in 'n'
int findPosition(unsigned n)
{
    if (!isPowerOfTwo(n))
        return -1;

    unsigned count = 0;

    // One by one move the only set bit to right till it reaches end
    while (n)
    {
        n = n >> 1;

        // increment count of shifts
        ++count;
    }

    return count;
}

// Driver program to test above function
int main(void)
{
    int n = 0;
    int pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    return 0;
}
```

Output:

```
n = 0, Invalid number
n = 12, Invalid number
n = 128, Position 8
```

We can also use log base 2 to find the position.

```
#include <stdio.h>

unsigned int Log2n(unsigned int n)
{
   return (n > 1)? 1 + Log2n(n/2): 0;
}

int isPowerOfTwo(unsigned n)
{
    return n && (! (n & (n-1)) );
}

int findPosition(unsigned n)
{
    if (!isPowerOfTwo(n))
        return -1;
    return Log2n(n) + 1;
}

// Driver program to test above function
int main(void)
{
    int n = 0;
    int pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
                 printf("n = %d, Position %d \n", n, pos);

    return 0;
}
```

Output:

```
n = 0, Invalid number
n = 12, Invalid number
n = 128, Position 8
```

Swap all odd and even bits

Given an unsigned integer, swap all odd bits with even bits. For example, if the given number is 23 (00010111), it should be converted to 43 (00101011). Every even position bit is swapped with adjacent bit on right side (even position bits are highlighted in binary representation of 23), and every odd position bit is swapped with adjacent on left side.

If we take a closer look at the example, we can observe that we basically need to right shift (>>) all even bits (In the above example, even bits of 23 are highlighted) by 1 so that they become odd bits (highlighted in 43), and left shift (<<) all odd bits by 1 so that they become even bits. The following solution is based on this observation. The solution assumes that input number is stored using 32 bits.

Let the input number be x

1. Get all even bits of x by doing bitwise and of x with 0xAAAAAAAA. The number 0xAAAAAAAA is a 32 bit number with all even bits set as 1 and all odd bits as 0.
2. Get all odd bits of x by doing bitwise and of x with 0x55555555. The number 0x55555555 is a 32 bit number with all odd bits set as 1 and all even bits as 0.
3. Right shift all even bits.
4. Left shift all odd bits.
5. Combine new even and odd bits and return.

```
// C program to swap even and odd bits of a given number
#include <stdio.h>

unsigned int swapBits(unsigned int x)
{
    // Get all even bits of x
    unsigned int even_bits = x & 0xAAAAAAAA;

    // Get all odd bits of x
    unsigned int odd_bits  = x & 0x55555555;

    even_bits >>= 1;  // Right shift even bits
    odd_bits <<= 1;   // Left shift odd bits

    return (even_bits | odd_bits); // Combine even and odd bits
}

// Driver program to test above function
int main()
```

```
{
    unsigned int x = 23; // 00010111

    // Output is 43 (00101011)
    printf("%u ", swapBits(x));

    return 0;
}
```

Output:

```
 43
```

Add two bit strings

Given two bit sequences as strings, write a function to return the addition of the two sequences. Bit strings can be of different lengths also. For example, if string 1 is “1100011” and second string 2 is “10”, then the function should return “1100101”.

Since sizes of two strings may be different, we first make the size of smaller string equal to that of bigger string by adding leading 0s. After making sizes same, we one by one add bits from rightmost bit to leftmost bit. In every iteration, we need to sum 3 bits: 2 bits of 2 given strings and carry. The sum bit will be 1 if, either all of the 3 bits are set or one of them is set. So we can do XOR of all bits to find the sum bit. How to find carry – carry will be 1 if any of the two bits is set. So we can find carry by taking OR of all pairs. Following is step by step algorithm.

1. Make them equal sized by adding 0s at the begining of smaller string.
2. Perform bit addition

…..Boolean expression for adding 3 bits a, b, c

…..Sum = a XOR b XOR c

…..Carry = (a AND b) OR (b AND c) OR (c AND a)

Following is C++ implementation of the above algorithm.

```
#include <iostream>
using namespace std;

//adds the two bit strings and return the result
string addBitStrings( string first, string second );

// Helper method: given two unequal sized bit strings, converts them to
// same length by aadding leading 0s in the smaller string. Returns the
// the new length
int makeEqualLength(string &str1, string &str2)
{
    int len1 = str1.size();
    int len2 = str2.size();
    if (len1 < len2)
    {
        for (int i = 0 ; i < len2 - len1 ; i++)
            str1 = '0' + str1;
        return len2;
    }
    else if (len1 > len2)
    {
        for (int i = 0 ; i < len1 - len2 ; i++)
            str2 = '0' + str2;
    }
```

```
    return len1; // If len1 >= len2
}

// The main function that adds two bit sequences and returns the addition
string addBitStrings( string first, string second )
{
    string result;  // To store the sum bits

    // make the lengths same before adding
    int length = makeEqualLength(first, second);

    int carry = 0;  // Initialize carry

    // Add all bits one by one
    for (int i = length-1 ; i >= 0 ; i--)
    {
        int firstBit = first.at(i) - '0';
        int secondBit = second.at(i) - '0';

        // boolean expression for sum of 3 bits
        int sum = (firstBit ^ secondBit ^ carry)+'0';

        result = (char)sum + result;

        // boolean expression for 3-bit addition
        carry = (firstBit & secondBit) | (secondBit & carry) | (firstBit &
carry);
    }

    // if overflow, then add a leading 1
    if (carry)
        result = '1' + result;

    return result;
}

// Driver program to test above functions
int main()
{
    string str1 = "1100011";
    string str2 = "10";

    cout << "Sum is " << addBitStrings(str1, str2);
    return 0;
}
```

Output:

```
Sum is 1100101
```

How to swap two numbers without using a temporary variable?

Given two variables, x and y, swap two variables without using a third variable.

Method 1 (Using Arithmetic Operators)
The idea is to get sum in one of the two given numbers. The numbers can then be swapped using the sum and subtraction from sum.

```
#include <stdio.h>
int main()
{
  int x = 10, y = 5;

  // Code to swap 'x' and 'y'
  x = x + y;  // x now becomes 15
  y = x - y;  // y becomes 10
  x = x - y;  // x becomes 5

  printf("After Swapping: x = %d, y = %d", x, y);

  return 0;
}
```

Output:

```
After Swapping: x = 5, y = 10
```

Multiplication and division can also be used for swapping.

```
#include <stdio.h>
int main()
{
  int x = 10, y = 5;

  // Code to swap 'x' and 'y'
  x = x * y;  // x now becomes 50
  y = x / y;  // y becomes 10
  x = x / y;  // x becomes 5

  printf("After Swapping: x = %d, y = %d", x, y);

  return 0;
}
```

Output:

```
After Swapping: x = 5, y = 10
```

Method 2 (Using Bitwise XOR)

The bitwise XOR operator can be used to swap two variables. The XOR of two numbers x and y returns a number which has all the bits as 1 wherever bits of x and y differ. For example XOR of 10 (In Binary 1010) and 5 (In Binary 0101) is 1111 and XOR of 7 (0111) and 5 (0101) is (0010).

```
#include <stdio.h>
int main()
{
  int x = 10, y = 5;

  // Code to swap 'x' (1010) and 'y' (0101)
  x = x ^ y;  // x now becomes 15 (1111)
  y = x ^ y;  // y becomes 10 (1010)
  x = x ^ y;  // x becomes 5 (0101)

  printf("After Swapping: x = %d, y = %d", x, y);

  return 0;
}
```

Output:

```
After Swapping: x = 5, y = 10
```

Problems with above methods

1) The multiplication and division based approach doesn’ work if one of the numbers is 0 as the product becomes 0 irrespective of the other number.

2) Both Arithmetic solutions may cause arithmetic overflow. If x and y are too large, addition and multiplication may go out of integer range.

3) When we use pointers to variable and make a function swap, all of the above methods fail when both pointers point to the same variable. Let’s take a look what will happen in this case if both are pointing to the same variable.

// Bitwise XOR based method

x = x ^ x; // x becomes 0

x = x ^ x; // x remains 0

x = x ^ x; // x remains 0

// Arithmetic based method

x = x + x; // x becomes 2x

x = x – x; // x becomes 0

x = x – x; // x remains 0

Let us see the following program.

```
#include <stdio.h>
void swap(int *xp, int *yp)
{
    *xp = *xp ^ *yp;
    *yp = *xp ^ *yp;
    *xp = *xp ^ *yp;
}

int main()
{
  int x = 10;
  swap(&x, &x);
  printf("After swap(&x, &x): x = %d", x);
  return 0;
}
```

Output:

```
After swap(&x, &x): x = 0
```

Swapping a variable with itself may needed in many standard algorithms. For example see thisimplementation of QuickSort where we may swap a variable with itself. The above problem can be avoided by putting a condition before the swapping.

```
#include <stdio.h>
void swap(int *xp, int *yp)
{
    if (xp == yp) // Check if the two addresses are same
      return;
    *xp = *xp + *yp;
    *yp = *xp - *yp;
    *xp = *xp - *yp;
}
int main()
{
  int x = 10;
  swap(&x, &x);
  printf("After swap(&x, &x): x = %d", x);
  return 0;
}
```

Output:

```
After swap(&x, &x): x = 10
```

Add two bit strings

Given two bit sequences as strings, write a function to return the addition of the two sequences. Bit strings can be of different lengths also. For example, if string 1 is “1100011” and second string 2 is “10”, then the function should return “1100101”.

Since sizes of two strings may be different, we first make the size of smaller string equal to that of bigger string by adding leading 0s. After making sizes same, we one by one add bits from rightmost bit to leftmost bit. In every iteration, we need to sum 3 bits: 2 bits of 2 given strings and carry. The sum bit will be 1 if, either all of the 3 bits are set or one of them is set. So we can do XOR of all bits to find the sum bit. How to find carry – carry will be 1 if any of the two bits is set. So we can find carry by taking OR of all pairs. Following is step by step algorithm.

1. Make them equal sized by adding 0s at the begining of smaller string.
2. Perform bit addition

…..Boolean expression for adding 3 bits a, b, c

…..Sum = a XOR b XOR c

…..Carry = (a AND b) OR (b AND c) OR (c AND a)

Following is C++ implementation of the above algorithm.

```
#include <iostream>
using namespace std;

//adds the two bit strings and return the result
string addBitStrings( string first, string second );

// Helper method: given two unequal sized bit strings, converts them to
// same length by aadding leading 0s in the smaller string. Returns the
// the new length
int makeEqualLength(string &str1, string &str2)
{
    int len1 = str1.size();
    int len2 = str2.size();
    if (len1 < len2)
    {
        for (int i = 0 ; i < len2 - len1 ; i++)
            str1 = '0' + str1;
        return len2;
    }
    else if (len1 > len2)
    {
        for (int i = 0 ; i < len1 - len2 ; i++)
```

```
            str2 = '0' + str2;
    }
    return len1; // If len1 >= len2
}

// The main function that adds two bit sequences and returns the addition
string addBitStrings( string first, string second )
{
    string result;  // To store the sum bits

    // make the lengths same before adding
    int length = makeEqualLength(first, second);

    int carry = 0;  // Initialize carry

    // Add all bits one by one
    for (int i = length-1 ; i >= 0 ; i--)
    {
        int firstBit = first.at(i) - '0';
        int secondBit = second.at(i) - '0';

        // boolean expression for sum of 3 bits
        int sum = (firstBit ^ secondBit ^ carry)+'0';

        result = (char)sum + result;

        // boolean expression for 3-bit addition
        carry = (firstBit & secondBit) | (secondBit & carry) | (firstBit &
carry);
    }

    // if overflow, then add a leading 1
    if (carry)
        result = '1' + result;

    return result;
}

// Driver program to test above functions
int main()
{
    string str1 = "1100011";
    string str2 = "10";

    cout << "Sum is " << addBitStrings(str1, str2);
    return 0;
}
```

Output:

```
Sum is 1100101
```

Check if binary representation of a number is palindrome

Given an integer ‘x’, write a C function that returns true if binary representation of x is palindrome else return false.

For example a numbers with binary representation as 10..01 is palindrome and number with binary representation as 10..00 is not palindrome.

The idea is similar to checking a string is palindrome or not. We start from leftmost and rightmost bits and compare bits one by one. If we find a mismatch, then return false.

Algorithm:

```
isPalindrome(x)
1) Find number of bits in x using sizeof() operator.
2) Initialize left and right positions as 1 and n respectively.
3) Do following while left 'l' is smaller than right 'r'.
.....a) If bit at position 'l' is not same as bit at position 'r',
then return false.
.....b) Increment 'l' and decrement 'r', i.e., do l++ and r--.
4) If we reach here, it means we didn't find a mismatching bit.
```

To find the bit at a given position, we can use the idea similar to this post. The expression “x & (1 << (k-1))” gives us non-zero value if bit at k’th position from right is set and gives a zero value if if k’th bit is not set.

Following is C++ implementation of the above algorithm.

```
#include<iostream>
using namespace std;

// This function returns true if k'th bit in x is set (or 1).
// For example if x (0010) is 2 and k is 2, then it returns true
bool isKthBitSet(unsigned int x, unsigned int k)
{
    return (x & (1 << (k-1)))? true: false;
}

// This function returns true if binary representation of x is
// palindrome. For example (1000...001) is paldindrome
bool isPalindrome(unsigned int x)
{
    int l = 1; // Initialize left position
    int r = sizeof(unsigned int)*8; // initialize right position

    // One by one compare bits
    while (l < r)
    {
```

```
        if (isKthBitSet(x, l) != isKthBitSet(x, r))
            return false;
        l++;     r--;
    }
    return true;
}

// Driver program to test above function
int main()
{
    unsigned int x = 1<<15 + 1<<16;
    cout << isPalindrome(x) << endl;
    x = 1<<31 + 1;
    cout << isPalindrome(x) << endl;
    return 0;
}
```

Output:

```
1
1
```

Get a bit from a given position

```
#include <stdio.h>
int getbit(int n, int pos);
int main()
{
        int n,a,pos;
        printf("Enter the number and bit pos\n");
        scanf("%d%d",&n,&pos);
        a = getbit(n,pos);
        printf("The bit at %d  position is %d\n",pos,a);
        return 0;
}
int getbit(int n, int pos)
{
        return(n >> pos) & 0x01 ;
}
```

Write an Efficient C Program to Reverse Bits of a Number

Method1 – Simple

Loop through all the bits of an integer. If a bit at ith position is set in the i/p no. then set the bit at (NO_OF_BITS – 1) – i in o/p. Where NO_OF_BITS is number of bits present in the given number.

```
/* Function to reverse bits of num */
unsigned int reverseBits(unsigned int num)
{
    unsigned int  NO_OF_BITS = sizeof(num) * 8;
    unsigned int reverse_num = 0, i, temp;

    for (i = 0; i < NO_OF_BITS; i++)
    {
        temp = (num & (1 << i));
        if(temp)
            reverse_num |= (1 << ((NO_OF_BITS - 1) - i));
    }

    return reverse_num;
}

/* Driver function to test above function */
int main()
{
    unsigned int x = 2;
    printf("%u", reverseBits(x));
    getchar();
}
```

Above program can be optimized by removing the use of variable temp. See below the modified code.

```
unsigned int reverseBits(unsigned int num)
{
    unsigned int  NO_OF_BITS = sizeof(num) * 8;
    unsigned int reverse_num = 0;
    int i;
    for (i = 0; i < NO_OF_BITS; i++)
    {
        if((num & (1 << i)))
           reverse_num |= 1 << ((NO_OF_BITS - 1) - i);
   }
    return reverse_num;
}
```

Time Complexity: O(log n)

Space Complexity: O(1)

Method 2 – Standard

The idea is to keep putting set bits of the num in reverse_num until num becomes zero. After num becomes zero, shift the remaining bits of reverse_num.

Let num is stored using 8 bits and num be 00000110. After the loop you will get reverse_num as 00000011. Now you need to left shift reverse_num 5 more times and you get the exact reverse 01100000.

```
unsigned int reverseBits(unsigned int num)
```

```
{
    unsigned int count = sizeof(num) * 8 - 1;
    unsigned int reverse_num = num;

    num >>= 1;
    while(num)
    {
       reverse_num <<= 1;
       reverse_num |= num & 1;
       num >>= 1;
       count--;
    }
    reverse_num <<= count;
    return reverse_num;
}

int main()
{
    unsigned int x = 1;
    printf("%u", reverseBits(x));
    getchar();
}
```

Time Complexity: O(log n)

Space Complexity: O(1)

Method 3 – Lookup Table:

We can reverse the bits of a number in O(1) if we know the size of the number. We can implement it using look up table. Go through the below link for details. You will find some more interesting bit related stuff there.

Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.

The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, in a system if we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays

1. Dynamic size
2. Ease of insertion/deletion

Drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2) Extra memory space for a pointer is required with each element of the list.

Representation in C:

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL. Each node in a list consists of at least two parts:

1) Data
2) Pointer to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

```
struct node
{
  int data;
  struct node *next;
};
```

First Simple Linked List in C

Let us create a simple linked list with 3 nodes.

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
  int data;
  struct node *next;
};

// Program to create a simple linked list with 3 nodes
int main()
{
  struct node* head = NULL;
  struct node* second = NULL;
  struct node* third = NULL;

  // allocate 3 nodes in the heap
  head = (struct node*)malloc(sizeof(struct node));
  second = (struct node*)malloc(sizeof(struct node));
  third = (struct node*)malloc(sizeof(struct node));

  /* Three blocks have been allocated dynamically.
     We have pointers to these three blocks as first, second and
third
       head           second           third
        |               |               |
        |               |               |
    +---+-----+     +----+----+     +----+----+
    | #  | #  |     | #  | #  |     |  # |  # |
    +---+-----+     +----+----+     +----+----+

   # represents any random value.
   Data is random because we haven't assigned anything yet  */

  head->data = 1; //assign data in first node
  head->next = second; // Link first node with the second node

  /* data has been assigned to data part of first block (block
```

```
    pointed by head).  And next pointer of first block points to
    second.  So they both are linked.

       head          second         third
        |              |              |
        |              |              |
    +---+---+     +----+----+     +-----+----+
    | 1  | o----->| #  | #  |     |  #  | #  |
    +---+---+     +----+----+     +-----+----+
  */

  second->data = 2; //assign data to second node
  second->next = third;

  /* data has been assigned to data part of second block (block pointed
by
     second). And next pointer of the second block points to third
block.
    So all three blocks are linked.

       head         second         third
        |             |             |
        |             |             |
    +---+---+     +---+---+     +----+----+
    | 1  | o----->| 2 | o-----> |  # |  # |
    +---+---+     +---+---+     +----+----+      */

  third->data = 3; //assign data to third node
  third->next = NULL;

  /* data has been assigned to data part of third block (block pointed
    by third). And next pointer of the third block is made NULL to
indicate
    that the linked list is terminated here.

     We have the linked list ready.

           head
             |
             |
        +---+---+     +---+---+       +----+------+
        | 1  | o----->|  2  | o-----> |  3 | NULL |
        +---+---+     +---+---+       +----+------+


    Note that only head is sufficient to represent the whole list.  We
can
    traverse the complete list by following next pointers.    */

  getchar();
  return 0;
}
```

Linked List Traversal

In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general purpose function printList() that prints any given list.

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
  int data;
  struct node *next;
};

// This function prints contents of linked list starting from
the given node
void printList(struct node *n)
{
  while (n != NULL)
  {
     printf(" %d ", n->data);
     n = n->next;
  }
}

int main()
{
  struct node* head = NULL;
  struct node* second = NULL;
  struct node* third = NULL;

  // allocate 3 nodes in the heap
  head  = (struct node*)malloc(sizeof(struct node));
  second = (struct node*)malloc(sizeof(struct node));
  third  = (struct node*)malloc(sizeof(struct node));

  head->data = 1; //assign data in first node
  head->next = second; // Link first node with the second node

  second->data = 2; //assign data to second node
  second->next = third;

  third->data = 3; //assign data to third node
  third->next = NULL;

  printList(head);

  getchar();
  return 0;
}
```

Output:

```
 1  2  3
```

Linked List Insertion

A node can be added in three ways

1) At the front of the linked list
2) After a given node.
3) At the end of the linked list.

1. Add a node at the front: (A 4 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List. For example if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push (). The push () must receive a pointer to the head pointer, because push must change the head pointer to point to the new node

Following are the 4 steps to add node at the front.

```
/* Given a reference (pointer to pointer) to the head of a list
and an int,
   inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct
node));

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref)    = new_node;
}
```

Time complexity of push () is O (1) as it does constant amount of work.

2. Add a node after a given node: (5 steps process)

We are given pointer to a node, and the new node is inserted after the given node.

```
/* Given a node prev_node, insert a new node after the given prev_node
*/
void insertAfter(struct node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
      printf("the given previous node cannot be NULL");
      return;
    }

    /* 2. allocate new node */
    struct  node*  new_node  =(struct  node*)  malloc(sizeof(struct
node));

    /* 3. put in the data  */
    new_node->data  = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}
```

Time complexity of insertAfter() is O(1) as it does constant amount of work.

3. Add a node at the end: (6 steps process)

The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.

Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

Following are the 6 steps to add node at the end.

```
/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end  */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref;  /* used in step 5*/

    /* 2. put in the data  */
```

```
    new_node->data  = new_data;

    /* 3. This new node is going to be the last node, so make next of it as
NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
       *head_ref = new_node;
       return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}
```

Time complexity of append is O(n) where n is the number of nodes in linked list. Since there is a loop from head to end, the function does O(n) work.

Following is a complete program that uses all of the above methods to create a linked list.

```
// A complete working C program to demonstrate all insertion methods
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
  int data;
  struct node *next;
};

/* Given a reference (pointer to pointer) to the head of a list and an int,
   inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref)    = new_node;
```

```
}

/* Given a node prev_node, insert a new node after the given prev_node */
void insertAfter(struct node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
      printf("the given previous node cannot be NULL");
      return;
    }

    /* 2. allocate new node */
    struct node* new_node =(struct node*) malloc(sizeof(struct node));

    /* 3. put in the data  */
    new_node->data  = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}

/* Given a reference (pointer to pointer) to the head
   of a list and an int, appends a new node at the end  */
void append(struct node** head_ref, int new_data)
{
    /* 1. allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    struct node *last = *head_ref;  /* used in step 5*/

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. This new node is going to be the last node, so make next of it as
NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make the new node as head */
    if (*head_ref == NULL)
    {
       *head_ref = new_node;
       return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}
```

```
// This function prints contents of linked list starting from the given node
void printList(struct node *node)
{
  while (node != NULL)
  {
     printf(" %d ", node->data);
     node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  // Insert 6.  So linked list becomes 6->NULL
  append(&head, 6);

  // Insert 7 at the beginning. So linked list becomes 7->6->NULL
  push(&head, 7);

  // Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
  push(&head, 1);

  // Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
  append(&head, 4);

  // Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
  insertAfter(head->next, 8);

  printf("\n Created Linked list is: ");
  printList(head);

  getchar();
  return 0;
}
```

Output:

```
 Created Linked list is:  1  7  8  6  4
```

Linked List - Deleting a node

Let us formulate the problem statement to understand the deletion process. Given a ‘key’, delete the first occurrence of this key in linked list.

To delete a node from linked list, we need to do following steps.

1. Find previous node of the node to be deleted.
2. Changed next of previous node.
3. Free memory for the node to be deleted.

Since every node of linked list is dynamically allocated using malloc () in C, we need to call free () for freeing memory allocated for the node to be deleted.

```
// A complete working C program to demonstrate deletion in singly
// linked list
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
};

/* Given a reference (pointer to pointer) to the head of a list
   and an int, inserts a new node on the front of the list. */
void push(struct node** head_ref, int new_data)
{
    struct  node*  new_node  =  (struct  node*)  malloc(sizeof(struct
node));
    new_node->data  = new_data;
    new_node->next = (*head_ref);
    (*head_ref)    = new_node;
}

/* Given a reference (pointer to pointer) to the head of a list
   and a key, deletes the first occurrence of key in linked list */
void deleteNode(struct node **head_ref, int key)
{
    // Store head node
    struct node* temp = *head_ref, *prev;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;   // Changed head
        free(temp);               // free old head
        return;
    }

    // Search for the key to be deleted, keep track of the
    // previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == NULL) return;

    // Unlink the node from linked list
    prev->next = temp->next;
```

```
    free(temp);  // Free memory
}

// This function prints contents of linked list starting from
// the given node
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf(" %d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 7);
    push(&head, 1);
    push(&head, 3);
    push(&head, 2);

    puts("Created Linked List: ");
    printList(head);
    deleteNode(&head, 1);
    puts("\nLinked List after Deletion of 1: ");
    printList(head);
    return 0;
}
```

Output:

```
Created Linked List:
 2  3  1  7
Linked List after Deletion of 1:
 2  3  7
```

A Programmer's approach of looking at Array vs. Linked List

In general, array is considered a data structure for which size is fixed at the compile time and array memory is allocated either from Data section (e.g. global array) or Stack section (e.g. local array).

Similarly, linked list is considered a data structure for which size is not fixed and memory is allocated from Heap section (e.g. using malloc () etc.) as and when needed. In this sense, array is taken as a static data structure (residing in Data or Stack section) while linked list is taken as a dynamic data structure (residing in Heap section). Memory representation of array and linked list can be visualized as follows:

An array of 4 elements (integer type) which have been initialized with 1, 2, 3 and 4. Suppose, these elements are allocated at memory addresses 0x100, 0x104, 0x08 and 0x10B respectively.

[(1)] [(2)] [(3)] [(4)]

0x100 0x104 0x108 0x10B

A linked list with 4 nodes where each node has integer as data and these data are initialized with 1, 2, 3 and 4. Suppose, these nodes are allocated via malloc () and memory allocated for them is 0x200, 0x308, 0x404 and 0x20B respectively.

[(1), 0x308] [(2), 0x404] [(3), 0x20B] [(4), NULL]

0x200 0x308 0x404 0x20B

Anyone with even little understanding of array and linked-list might not be interested in the above explanation. I mean, it is well know that the array elements are allocated memory in sequence i.e. contiguous memory while nodes of a linked list are non-contiguous in memory. Though it sounds trivial yet this the most important difference between array and linked list. It should be noted that due to this contiguous versus non-contiguous memory, array and linked list are different. In fact, this difference is what makes array vs. linked list! In the following sections, we will try to explore on this very idea further.

Since elements of array are contiguous in memory, we can access any element randomly using index e.g. int Arr [3] will access directly fourth element of the array. (For newbies, array indexing start from 0 and that's why fourth element is indexed with 3). Also, due to contiguous memory for successive elements in array, no extra information is needed to be stored in individual elements i.e. no overhead of metadata in arrays. Contrary to this, linked list nodes are non-contiguous in memory. It means that we need some mechanism to traverse or access linked list nodes. To achieve this, each node stores the location of next node and this forms the basis of the link from one node to next node. Therefore, it's called Linked list. Though storing the location of next node is overhead in linked list but it's required. Typically, we see linked list node declaration as follows:

```
struct llNode
{
  int dataInt;

  /* nextNode is the pointer to next node in linked list*/
```

```
  struct llNode * nextNode;
};
```

So array elements are contiguous in memory and therefore not requiring any metadata. And linked list nodes are non-contiguous in memory thereby requiring metadata in the form of location of next node. Apart from this difference, we can see that array could have several unused elements because memory has already been allocated. But linked list will have only the required no. of data items. All the above information about array and linked list has been mentioned in several textbooks though in different ways.

What if we need to allocate array memory from Heap section (i.e. at run time) and linked list memory from Data/Stack section? First of all, is it possible? Before that, one might ask why someone would need to do this. Now, I hope that the remaining article would make you rethink about the idea of array vs. linked-list

Now consider the case when we need to store certain data in array (because array has the property of random access due to contiguous memory) but we don't know the total size apriori. One possibility is to allocate memory of this array from Heap at run time. For example, as follows:

/*At run-time, suppose we know the required size for integer array (e.g. input size from user). Say, the array size is stored in variable arrSize. Allocate this array from Heap as follows*/

```
int * dynArr = (int *)malloc(sizeof(int)*arrSize);
```

Though the memory of this array is allocated from Heap, the elements can still be accessed via index mechanism e.g. dynArr[i]. Basically, based on the programming problem, we have combined one benefit of array (i.e. random access of elements) and one benefit of linked list (i.e. delaying the memory allocation till run time and allocating memory from Heap). Another advantage of having this type of dynamic array is that, this method of allocating array from Heap at run time could reduce code-size (of course, it depends on certain other factors e.g. program format etc.)

Now consider the case when we need to store data in a linked list (because no. of nodes in linked list would be equal to actual data items stored i.e. no extra space like array) but we aren't allowed to get this memory from Heap again and again for each node. This might look hypothetical situation to some folks but it's not very uncommon requirement in embedded systems. Basically, in several embedded programs, allocating memory via malloc () etc. isn't allowed due to multiple reasons. One obvious reason is

performance i.e. allocating memory via malloc () is costly in terms of time complexity because your embedded program is required to be deterministic most of the times. Another reason could be module specific memory management i.e. it's possible that each module in embedded system manages its own memory. In short, if we need to perform our own memory management, instead of relying on system provided APIs of malloc () and free (), we might choose the linked list which is simulated using array. I hope that you got some idea why we might need to simulate linked list using array. Now, let us first see how this can be done. Suppose, type of a node in linked list (i.e. underlying array) is declared as follows:

```
struct sllNode
{
  int dataInt;

 /*Here, note that nextIndex stores the location of next node in
  linked list*/
  int nextIndex;
};

struct sllNode arrayLL[5];
```

If we initialize this linked list (which is actually an array), it would look as follows in memory:

[(0),-1] [(0),-1] [(0),-1] [(0),-1] [(0),-1]

0x500 0x508 0x510 0x518 0x520

The important thing to notice is that all the nodes of the linked list are contiguous in memory (each one occupying 8 bytes) and nextIndex of each node is set to -1. This (i.e. -1) is done to denote that the each node of the linked list is empty as of now. This linked list is denoted by head index 0.

Now, if this linked list is updated with four elements of data part 4, 3, 2 and 1 successively, it would look as follows in memory. This linked list can be viewed as 0x500 -> 0x508 -> 0x510 -> 0x518.

[(1),1] [(2),2] [(3),3] [(4),-2] [(0),-1]

0x500 0x508 0x510 0x518 0x520

The important thing to notice is nextIndex of last node (i.e. fourth node) is set to -2. This (i.e. -2) is done to denote the end of linked list. Also, head node of the linked list

is index 0. This concept of simulating linked list using array would look more interesting if we delete say second node from the above linked list. In that case, the linked list will look as follows in memory:

[(1),2] [(0),-1] [(3),3] [(4),-2] [(0),-1]

0x500 0x508 0x510 0x518 0x520

The resultant linked list is 0x500 -> 0x510 -> 0x518. Here, it should be noted that even though we have deleted second node from our linked list, the memory for this node is still there because underlying array is still there. But the nextIndex of first node now points to third node (for which index is 2).

Hopefully, the above examples would have given some idea that for the simulated linked list, we need to write our own API similar to malloc() and free() which would basically be used to insert and delete a node. Now this is what's called own memory management. Let us see how this can be done in algorithmic manner.

There are multiple ways to do so. If we take the simplistic approach of creating linked list using array, we can use the following logic. For inserting a node, traverse the underlying array and find a node whose nextIndex is -1. It means that this node is empty. Use this node as a new node. Update the data part in this new node and set the nextIndex of this node to current head node (i.e. head index) of the linked list. Finally, make the index of this new node as head index of the linked list. To visualize it, let us take an example. Suppose the linked list is as follows where head Index is 0 i.e. linked list is 0x500 -> 0x508 -> 0x518 -> 0x520

[(1),1] [(2),3] [(0),-1] [(4),4] [(5),-2]

0x500 0x508 0x510 0x518 0x520

After inserting a new node with data 8, the linked list would look as follows with head index as 2.

[(1),1] [(2),3] [(8),0] [(4),4] [(5),-2]

0x500 0x508 0x510 0x518 0x520

So the linked list nodes would be at addresses 0x510 -> 0x500 -> 0x508 -> 0x518 -> 0x520

For deleting a node, we need to set the nextIndex of the node as -1 so that the node is marked as empty node. But, before doing so, we need to make sure that the nextIndex of the previous node is updated correctly to index of next node of this node to be deleted. We can see that we have done own memory management for creating a linked list out of the array memory. But, this is one way of inserting and deleting nodes in this linked list. It can be easily noticed that finding an empty node is not so efficient in terms of time complexity. Basically, we're searching the complete array linearly to find an empty node.

Let us see if we can optimize it further. Basically we can maintain a linked list of empty nodes as well in the same array. In that case, the linked list would be denoted by two indexes – one index would be for linked list which has the actual data values i.e. nodes which have been inserted so far and other index would for linked list of empty nodes. By doing so, whenever, we need to insert a new node in existing linked list, we can quickly find an empty node. Let us take an example:

[(4),2] [(0),3] [(5),5] [(0),-1] [(0),1] [(9),-1]

0x500 0x508 0x510 0x518 0x520 0x528

The above linked list which is represented using two indexes (0 and 5) has two linked lists: one for actual values and another for empty nodes. The linked list with actual values has nodes at address 0x500 -> 0x510 -> 0x528 while the linked list with empty nodes has nodes at addresses 0x520 -> 0x508 -> 0x518. It can be seen that finding an empty node (i.e. writing own API similar to malloc()) should be relatively faster now because we can quickly find a free node. In real world embedded programs, a fixed chunk of memory (normally called memory pool) is allocated using malloc() only once by a module. And then the management of this memory pool (which is basically an array) is done by that module itself using techniques mentioned earlier. Sometimes, there are multiple memory pools each one having different size of node. Of course, there are several other aspects of own memory management but we'll leave it here itself. But it's worth mentioning that there are several methods by which the insertion (which requires our own memory allocation) and deletion (which requires our own memory freeing) can be improved further.

If we look carefully, it can be noticed that the Heap section of memory is basically a big array of bytes which is being managed by the underlying operating system (OS).

And OS is providing this memory management service to programmers via malloc(), free() etc.

The important take-aways from this article can be summed as follows:

A) Array means contiguous memory. It can exist in any memory section be it Data or Stack or Heap.
B) Linked List means non-contiguous linked memory. It can exist in any memory section be it Heap or Data or Stack.
C) As a programmer, looking at a data structure from memory perspective could provide us better insight in choosing a particular data structure or even designing a new data structure. For example, we might create an array of linked lists etc.

Question

1. How to write C functions that modify head pointer of a Linked List?

Consider simple representation (without any dummy node) of Linked List. Functions that operate on such Linked lists can be divided in two categories:

1) Functions that do not modify the head pointer: Examples of such functions include, printing a linked list, updating data members of nodes like adding given a value to all nodes, or some other operation which access/update data of nodes It is generally easy to decide prototype of functions of this category. We can always pass head pointer as an argument and traverse/update the list. For example, the following function that adds x to data members of all nodes.

```
void addXtoList(struct node *node, int x)
{
    while(node != NULL)
    {
        node->data = node->data + x;
        node = node->next;
    }
}
```

2) Functions that modify the head pointer: Examples include, inserting a node at the beginning (head pointer is always modified in this function), inserting a node at the end (head pointer is modified only when the first node is being inserted), and deleting a given node (head pointer is modified when the deleted node is first node). There may be different ways to update the head pointer in these functions. Let us discuss these ways using following simple problem:

“Given a linked list, write a function deleteFirst() that deletes the first node of a given linked list.

For example, if the list is 1->2->3->4, then it should be modified to 2->3->4″

Algorithm to solve the problem is a simple 3 step process:

i. Store the head pointer
ii. change the head pointer to point to next node
iii. Delete the previous head node.

Following are different ways to update head pointer in deleteFirst() so that the list is updated everywhere.

2.1) make head pointer global: We can make the head pointer global so that it can be accessed and updated in our function. Following is C code that uses global head pointer.

```
// global head pointer
struct node *head = NULL;

// function to delete first node: uses approach 2.1
// See http://ideone.com/ClfQB for complete program and output
void deleteFirst()
{
    if(head != NULL)
    {
       // store the old value of head pointer
       struct node *temp = head;

       // Change head pointer to point to next node
       head = head->next;

       // delete memory allocated for the previous head node
       free(temp);
    }
}
```

See this for complete running program that uses above function.

This is not a recommended way as it has many problems like following:
a) head is globally accessible, so it can be modified anywhere in your project and may lead to unpredictable results.

b) If there are multiple linked lists, then multiple global head pointers with different names are needed.

See this to know all reasons why should we avoid global variables in our projects.

2.2) Return head pointer: We can write deleteFirst() in such a way that it returns the modified head pointer. Whoever is using this function, have to use the returned value to update the head node.

```
// function to delete first node: uses approach 2.2
// See http://ideone.com/P5oLe for complete program and output
struct node *deleteFirst(struct node *head)
{
    if(head != NULL)
    {
       // store the old value of head pointer
       struct node *temp = head;

       // Change head pointer to point to next node
       head = head->next;

       // delete memory allocated for the previous head node
       free(temp);
    }

    return head;
}
```

See this for complete program and output.

This approach is much better than the previous 1. There is only one issue with this, if user misses to assign the returned value to head, then things become messy. C/C++ compilers allows to call a function without assigning the returned value.

```
head = deleteFirst(head);  // proper use of deleteFirst()
deleteFirst(head);   // improper use of deleteFirst(), allowed by
compiler
```

2.3) Use Double Pointer: This approach follows the simple C rule: if you want to modify local variable of one function inside another function, pass pointer to that variable. So we can pass pointer to the head pointer to modify the head pointer in our deleteFirst() function.

```
// function to delete first node: uses approach 2.3
// See http://ideone.com/9GwTb for complete program and output
void deleteFirst(struct node **head_ref)
{
    if(*head_ref != NULL)
    {
       // store the old value of pointer to head pointer
       struct node *temp = *head_ref;

       // Change head pointer to point to next node
       *head_ref = (*head_ref)->next;

       // delete memory allocated for the previous head node
```

```
        free(temp);
    }
}
```

2. Write a GetNth() function that takes a linked list and an integer index and returns the data value stored in the node at that index position.

Algorithm:

```
1. Initialize count = 0
2. Loop through the link list
     a. if count is equal to the passed index then return current
         node
     b. Increment count
     c. change current to point to next of the current.
```

Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
    of a list and an int, push a new node on the front
    of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Takes head pointer of the linked list and index
```

```
    as arguments and return data at index*/
int GetNth(struct node* head, int index)
{
    struct node* current = head;
    int count = 0; /* the index of the node we're currently
                  looking at */
    while (current != NULL)
    {
       if (count == index)
          return(current->data);
       count++;
       current = current->next;
    }

    /* if we get to this line, the caller was asking
       for a non-existent element so we assert fail */
    assert(0);
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
     1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    /* Check the count function */
    printf("Element at index 3 is %d", GetNth(head, 3));
    getchar();
}
```

Time Complexity: O(n)

3. Given only a pointer to a node to be deleted in a singly linked list, how do you delete it?

A simple solution is to traverse the linked list until you find the node you want to delete. But this solution requires pointer to the head node which contradicts the problem statement.

Fast solution is to copy the data from the next node to the node to be deleted and delete the next node. Something like this following.

```
    struct node *temp  = node_ptr->next;
    node_ptr->data  = temp->data;
    node_ptr->next  = temp->next;
    free(temp);
```

Program:

```
#include<stdio.h>
#include<assert.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
    of a list and an int, push a new node on the front
    of the list. */
void push(struct node** head_ref, int new_data)
{
   /* allocate node */
   struct node* new_node =
             (struct node*) malloc(sizeof(struct node));

   /* put in the data  */
   new_node->data  = new_data;

   /* link the old list off the new node */
   new_node->next = (*head_ref);

   /* move the head to point to the new node */
   (*head_ref)    = new_node;
}

void printList(struct node *head)
{
   struct node *temp = head;
   while(temp != NULL)
   {
      printf("%d  ", temp->data);
      temp = temp->next;
   }
}

void deleteNode(struct node *node_ptr)
{
   struct node *temp = node_ptr->next;
   node_ptr->data    = temp->data;
   node_ptr->next    = temp->next;
   free(temp);
}
```

```
/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
    1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    printf("\n Before deleting \n");
    printList(head);

    /* I m deleting the head itself.
        You can check for more cases */
    deleteNode(head);

    printf("\n After deleting \n");
    printList(head);
    getchar();
    return 0;
}
```

This solution doesn't work if the node to be deleted is the last node of the list. To make this solution work we can mark the end node as a dummy node. But the programs/functions that are using this function should also be modified.

4. Write a C function to print the middle of a given linked list

Method 1:

Traverse the whole linked list and count the no. of nodes. Now traverse the list again till count/2 and return the node at count/2.

Method 2:

Traverse linked list using two pointers. Move one pointer by one and other pointer by two. When the fast pointer reaches end slow pointer will reach middle of the linked list.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};
```

```
/* Function to get the middle of the linked list*/
void printMiddle(struct node *head)
{
    struct node *slow_ptr = head;
    struct node *fast_ptr = head;

    if (head!=NULL)
    {
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;
            slow_ptr = slow_ptr->next;
        }
        printf("The middle element is [%d]\n\n", slow_ptr->data);
    }
}

void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%d->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    for (i=5; i>0; i--)
    {
        push(&head, i);
        printList(head);
```

```
        printMiddle(head);
    }

    return 0;
}
```

Output:

```
5->NULL
The middle element is [5]

4->5->NULL
The middle element is [5]

3->4->5->NULL
The middle element is [4]

2->3->4->5->NULL
The middle element is [4]

1->2->3->4->5->NULL
The middle element is [3]
```

Method 3:

Initialize mid element as head and initialize a counter as 0. Traverse the list from head, while traversing increment the counter and change mid to mid->next whenever the counter is odd. So the mid will move only half of the total length of the list.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to get the middle of the linked list*/
void printMiddle(struct node *head)
{
    int count = 0;
    struct node *mid = head;

    while (head != NULL)
    {
        /* update mid, when 'count' is odd number */
        if (count & 1) //(count%2 == 1)
            mid = mid->next;

        ++count;
        head = head->next;
    }
```

```
    /* if empty list is provided */
    if (mid != NULL)
        printf("The middle element is [%d]\n\n", mid->data);
}


void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%d->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    for (i=5; i>0; i--)
    {
        push(&head, i);
        printList(head);
        printMiddle(head);
    }

    return 0;
}
```

Output:

```
5->NULL
```

```
The middle element is [5]

4->5->NULL
The middle element is [5]

3->4->5->NULL
The middle element is [4]

2->3->4->5->NULL
The middle element is [4]

1->2->3->4->5->NULL
The middle element is [3]
```

5. Nth node from the end of a Linked List

Given a Linked List and a number n, write a function that returns the value at the nth node from end of the Linked List.

Method 1 (Use length of linked list)

1) Calculate the length of Linked List. Let the length be len.
2) Print the (len – n + 1)th node from the begining of the Linked List.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
  int data;
  struct node* next;
};

/* Function to get the nth node from the last of a linked list*/
void printNthFromLast(struct node* head, int n)
{
    int len = 0, i;
    struct node *temp = head;

    // 1) count the number of nodes in Linked List
    while (temp != NULL)
    {
        temp = temp->next;
        len++;
    }

    // check if value of n is not more than length of the linked list
    if (len < n)
      return;

    temp = head;
```

```
    // 2) get the (n-len+1)th node from the begining
    for (i = 1; i < len-n+1; i++)
       temp = temp->next;

    printf ("%d", temp->data);

    return;
}

void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  // create linked 35->15->4->20
  push(&head, 20);
  push(&head, 4);
  push(&head, 15);
  push(&head, 35);

  printNthFromLast(head, 5);
  getchar();
  return 0;
}
```

Other Method

```
LinkedListNode nthToLast(LinkedListNode head, int n) {
  if (head == null || n < 1) {
  return null;
}
  LinkedListNode p1 = head;
  LinkedListNode p2 = head;
  for (int j = 0; j < n - 1; ++j) { // skip n-1 steps ahead
  if (p2 == null) {
       return null; // not found since list size < n
   }
  p2 = p2.next;
```

```
  }
  while (p2.next != null) {
  p1 = p1.next;
  p2 = p2.next;
 }
   return p1;
 }
```

If the elements are 8->10->5->7->2->1->5->4->10->10 then the result is 7th to last node is 7

Algorithm works by first creating references to two nodes in your linked list that are N nodes apart. Thus, in your example, if N is 7, then it will set p1 to 8 and p2 to 4.

It will then advance each node reference to the next node in the list until p2 reaches the last element in the list. Again, in your example, this will be when p1 is 5 and p2 is 10. At this point, p1 is referring to the Nth to the last element in the list (by the property that they are N nodes apart).

Following is a recursive C code for the same method.

```
void printNthFromLast(struct node* head, int n)
{

/*this has to be static so that it holds it's value from
  call to call because of the fact that a non-static variable
  will not hold it's value since the variable will just be local
  to the function if it's not static and will not be preserved
  across call stack:  */
  static int i = 0;

//base case when we reach the end of linked list:
    if(head == NULL)
       return;
//this is where head pointer is advanced (head->next)...
    printNthFromLast(head->next, n);

//increment i and check to see if equals n
   //if it does equal n then print out the element's data
    if(++i == n)
       printf("%d", head->data);
}
```

Time Complexity: O(n) where n is the length of linked list.

Method 2 (Use two pointers)

Maintain two pointers – reference pointer and main pointer. Initialize both reference and main pointers to head. First move reference pointer to n nodes from head. Now

move both pointers one by one until reference pointer reaches end. Now main pointer will point to nth node from the end. Return main pointer.

Implementation:

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
  int data;
  struct node* next;
};

/* Function to get the nth node from the last of a linked list*/
void printNthFromLast(struct node *head, int n)
{
  struct node *main_ptr = head;
  struct node *ref_ptr = head;

  int count = 0;
  if(head != NULL)
  {
     while( count < n )
     {
        if(ref_ptr == NULL)
        {
           printf("%d is greater than the no. of "
                    "nodes in list", n);
           return;
        }
        ref_ptr = ref_ptr->next;
        count++;
     } /* End of while*/

     while(ref_ptr != NULL)
     {
        main_ptr = main_ptr->next;
        ref_ptr  = ref_ptr->next;
     }
     printf("Node no. %d from last is %d ",
               n, main_ptr->data);
  }
}

void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;
```

```
  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;
  push(&head, 20);
  push(&head, 4);
  push(&head, 15);

  printNthFromLast(head, 3);
  getchar();
}
```

Time Complexity: O(n) where n is the length of linked list.

6. Write a function to delete a Linked List

Algorithm: Iterate through the linked list and delete all the nodes one by one. Main point here is not to access next of the current pointer if current pointer is deleted.

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to delete the entire linked list */
void deleteList(struct node** head_ref)
{
   /* deref head_ref to get the real head */
   struct node* current = *head_ref;
   struct node* next;

   while (current != NULL)
   {
       next = current->next;
       free(current);
       current = next;
   }

   /* deref head_ref to affect the real head back
```

```
      in the caller. */
   *head_ref = NULL;
}

/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Use push() to construct below list
     1->12->1->4->1  */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    printf("\n Deleting linked list");
    deleteList(&head);

    printf("\n Linked list deleted");
    getchar();
}
```

Time Complexity: O(n)
Space Complexity: O(1)

7. Write a function that counts the number of times a given int occurs in a Linked List

Algorithm:

```
1. Initialize count as zero.
2. Loop through each element of linked list:
```

```
     a) If element data is equal to the passed number then
        increment the count.
3. Return count.
```

Implementation:

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Counts the no. of occurences of a node
   (search_for) in a linked list (head)*/
int count(struct node* head, int search_for)
{
    struct node* current = head;
    int count = 0;
    while (current != NULL)
    {
        if (current->data == search_for)
           count++;
        current = current->next;
    }
    return count;
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
```

```
    /* Use push() to construct below list
     1->2->1->3->1  */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    printf("count of 1 is %d", count(head, 1));
    getchar();
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)

8. Write a function to reverse a linked list

i. Iterative Method:

Iterate trough the linked list. In loop, change next to prev, prev to current and current to next.

Implementation of Iterative Method

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to reverse the linked list */
static void reverse(struct node** head_ref)
{
    struct node* prev   = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
```

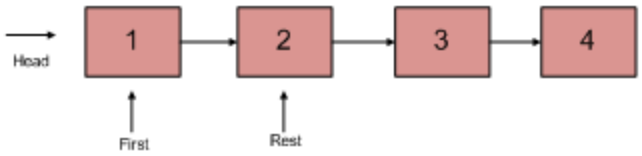
```
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d  ", temp->data);
        temp = temp->next;
    }
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

     push(&head, 20);
     push(&head, 4);
     push(&head, 15);
     push(&head, 85);

     printList(head);
     reverse(&head);
     printf("\n Reversed Linked list \n");
     printList(head);
     getchar();
}
```

Time Complexity: O(n) Space Complexity: O(1)

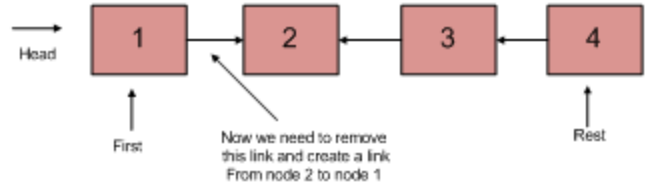
ii. Recursive Method:

```
   1) Divide the list in two parts - first node and rest of the linked
list.
   2) Call reverse for the rest of the linked list.
   3) Link rest to first.
   4) Fix head pointer
```

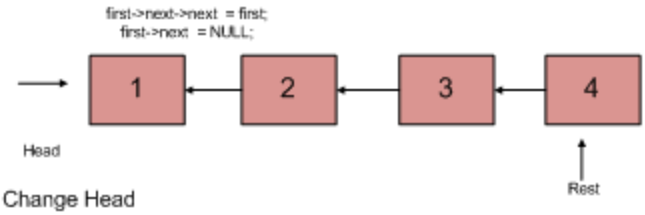
Divide the List in two parts



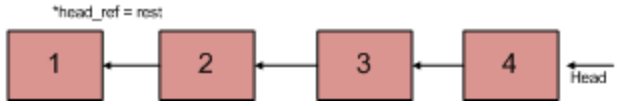
Reverse Rest



Link Rest to First



Change Head




```
void recursiveReverse(struct node** head_ref)
{
    struct node* first;
    struct node* rest;

    /* empty list */
    if (*head_ref == NULL)
       return;

    /* suppose first = {1, 2, 3}, rest = {2, 3} */
    first = *head_ref;
    rest  = first->next;

    /* List has only one node */
    if (rest == NULL)
       return;

    /* reverse the rest list and put the first element at the end
*/
    recursiveReverse(&rest);
    first->next->next  = first;

    /* tricky step -- see the diagram */
    first->next  = NULL;
```

```
    /* fix the head pointer */
    *head_ref = rest;
}
```

9. Write a recursive function to print reverse of a Linked List

Algorithm

```
printReverse(head)
  1. call print reverse for hed->next
  2. print head->data
```

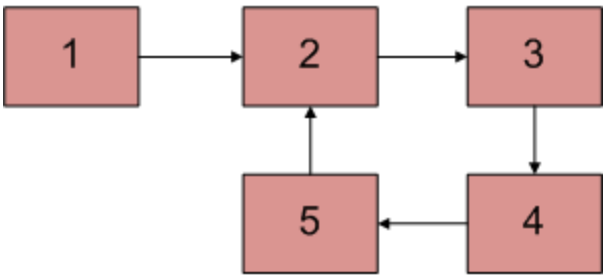
Implementation:

```
/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to reverse the linked list */
void printReverse(struct node* head)
{
  // Base case
  if(head == NULL)
    return;

  // print the list after head node
  printReverse(head->next);

  // After everything else is printed, print head
  printf("%d  ", head->data);
}

int main()
{

  struct node* head = NULL;
  printReverse(head);
}
```

Time Complexity: O(n)

10. Write a C function to detect loop in a linked list

Below diagram shows a linked list with a loop



Following are different ways of doing this

Use Hashing:
Traverse the list one by one and keep putting the node addresses in a Hash Table. At any point, if NULL is reached then return false and if next of current node points to any of the previously stored nodes in Hash then return true.

Mark Visited Nodes:
This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the linked list and keep marking visited nodes. If you see a visited node again then there is a loop. This solution works in O(n) but requires additional information with each node.

A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Just store the addresses of visited nodes in a hash and if you see an address that already exists in hash then there is a loop.

Floyd's Cycle-Finding Algorithm:
This is the fastest method. Traverse linked list using two pointers. Move one pointer by one and other pointer by two. If these pointers meet at some node then there is a loop. If pointers do not meet then linked list doesn't have loop.

Once you know a node within the loop, there's an O(n) guaranteed method to find the start of the loop.

Let's return to the original position after you've found an element somewhere in the loop, but you're not sure where the start of the loop is.

```
                                AB (this is where A and B
                                |               first met).
                                v
```

```
head -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8
                  ^                        |
                  |                        |
                  +------------------------+
```

This is the process to follow:

1. First, advance B and set the loopsize to 1.
2. Second, while A and B are not equal, continue to advance B, increasing the loopsize each time. That gives the size of the loop, six in this case. If the loopsize ends up as 1, you knowthat you must already be at the start of the loop, so simply return A as the start, and skip the rest of the steps below.
3. Third, simply set both A and B to the first element then advance B exactly loopsize times (to7 in this case). This gives two pointers that are different by the size of the loop.
4. Lastly, while A and B are not equal, you advance them together. Since they remain exactly loopsize elements apart from each other at all times, A will enter the loop at exactly the same time as B returns to the start of the loop. You can see that with the following walkthrough:
    a. `loopsize` is evaluated as `6`
    b. set both `A` and `B` to `1`
    c. advance `B` by `loopsize` elements to `7`
    d. `1` and `7` aren't equal so advance both
    e. `2` and `8` aren't equal so advance both
    f. `3` and `3` *are* equal so that is your loop start

Now, since each those operations are O(n) and performed sequentially, the whole thing is O(n).

Implementation of Floyd's Cycle-Finding Algorithm:

```
/* Link list node */
struct node
{
  int data;
  struct node* next;
};

int detectloop(struct node *list)
{
  struct node  *slow_p = list, *fast_p = list;

  while(slow_p && fast_p &&
          fast_p->next )
```

```
  {
    slow_p = slow_p->next;
    fast_p  = fast_p->next->next;
    if (slow_p == fast_p)
    {
       printf("Found Loop");
       return 1;
    }
  }
  return 0;
}

int main()
{
  /* Start with the empty list */
  struct node* head = NULL;


  /* Create a loop for testing */
  head->next->next->next->next = head;
  detectloop(head);

  getchar();
}
```

Time Complexity: O(n) Auxiliary Space: O(1)

11. Function to check if a singly linked list is palindrome

Given a singly linked list of characters, write a function that returns true if the given list is palindrome, else false.

METHOD 1 (Use a Stack)

A simple solution is to use a stack of list nodes. This mainly involves three steps.

1. Traverse the given list from head to tail and push every visited node to stack.
2. Traverse the list again. For every visited node, pop a node from stack and compare data of popped node with currently visited node.
3. If all nodes matched, then return true, else false.

Time complexity of above method is O(n), but it requires O(n) extra space. Following methods solve this with constant extra space.

METHOD 2 (By reversing the list)

This method takes O(n) time and O(1) extra space.

1. Get the middle of the linked list.
2. Reverse the second half of the linked list.
3. Check if the first half and second half are identical.
4. Construct the original linked list by reversing the second half again and attaching it back to the first half

To divide the list in two halves, method 2 of this post is used.

When number of nodes is even, the first and second half contains exactly half nodes. The challenging thing in this method is to handle the case when number of nodes is odd. We don't want the middle node as part of any of the lists as we are going to compare them for equality. For odd case, we use a separate variable 'midnode'.

```
/* Program to check if a linked list is palindrome */
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Link list node */
struct node
{
    char data;
    struct node* next;
};

void reverse(struct node**);
bool compareLists(struct node*, struct node *);

/* Function to check if given linked list is
  palindrome or not */
bool isPalindrome(struct node *head)
{
    struct node *slow_ptr = head, *fast_ptr = head;
    struct node *second_half, *prev_of_slow_ptr = head;
    struct node *midnode = NULL;  // To handle odd size list
    bool res = true; // initialize result

    if (head!=NULL && head->next!=NULL)
    {
        /* Get the middle of the list. Move slow_ptr by 1
```

```
          and fast_ptrr by 2, slow_ptr will have the middle
          node */
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;

            /*We need previous of the slow_ptr for
             linked lists  with odd elements */
            prev_of_slow_ptr = slow_ptr;
            slow_ptr = slow_ptr->next;
        }


        /* fast_ptr would become NULL when there are even elements
           in list.And not NULL for odd elements. We need to skip the
           middle node for odd case and store it somewhere so that
           we can restore the original list*/
        if (fast_ptr != NULL)
        {
            midnode = slow_ptr;
            slow_ptr = slow_ptr->next;
        }

        // Now reverse the second half and compare it with first half
        second_half = slow_ptr;
        prev_of_slow_ptr->next = NULL; // NULL terminate first half
        reverse(&second_half);  // Reverse the second half
        res = compareLists(head, second_half); // compare

        /* Construct the original list back */
         reverse(&second_half); // Reverse the second half again
         if (midnode != NULL)  // If there was a mid node (odd size
case) which

          // was not part of either first half or second half.
         {
            prev_of_slow_ptr->next = midnode;
            midnode->next = second_half;
         }
         else  prev_of_slow_ptr->next = second_half;
    }
    return res;
}

/* Function to reverse the linked list  Note that this
    function may change the head */
void reverse(struct node** head_ref)
{
    struct node* prev   = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
```

```
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to check if two input lists have same data*/
bool compareLists(struct node* head1, struct node *head2)
{
    struct node* temp1 = head1;
    struct node* temp2 = head2;

    while (temp1 && temp2)
    {
        if (temp1->data == temp2->data)
        {
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        else return 0;
    }

    /* Both are empty reurn 1*/
    if (temp1 == NULL && temp2 == NULL)
        return 1;

    /* Will reach here when one is NULL
      and other is not */
    return 0;
}

/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref)    = new_node;
}
```

```
// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%c->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}


/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    char str[] = "abacaba";
    int i;

    for (i = 0; str[i] != '\0'; i++)
    {
       push(&head, str[i]);
       printList(head);
       isPalindrome(head)? printf("Is Palindrome\n\n"):
                           printf("Not Palindrome\n\n");
    }

    return 0;
}
```

Output:

```
a->NULL
Palindrome

b->a->NULL
Not Palindrome

a->b->a->NULL
Is Palindrome

c->a->b->a->NULL
Not Palindrome

a->c->a->b->a->NULL
Not Palindrome

b->a->c->a->b->a->NULL
Not Palindrome
```

```
a->b->a->c->a->b->a->NULL
Is Palindrome
```

Time Complexity O(n) Auxiliary Space: O(1)

METHOD 3 (Using Recursion)

Use two pointers left and right. Move right and left using recursion and check for following in each recursive call.

1) Sub-list is palindrome.

2) Value at current left and right are matching.

If both above conditions are true then return true.

The idea is to use function call stack as container. Recursively traverse till the end of list. When we return from last NULL, we will be at last node. The last node to be compared with first node of list.

In order to access first node of list, we need list head to be available in the last call of recursion. Hence we pass head also to the recursive function. If they both match we need to compare (2, n-2) nodes. Again when recursion falls back to (n-2)nd node, we need reference to 2nd node from head. We advance the head pointer in previous call, to refer to next node in the list.

However, the trick is in identifying double pointer. Passing single pointer is as good as pass-by-value, and we will pass the same pointer again and again. We need to pass the address of head pointer for reflecting the changes in parent recursive calls.

```
// Recursive program to check if a given linked list is palindrome
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* Link list node */
struct node
{
    char data;
    struct node* next;
};

// Initial parameters to this function are &head and head
bool isPalindromeUtil(struct node **left, struct  node *right)
{
   /* stop recursion when right becomes NULL */
   if (right == NULL)
      return true;
```

```
   /* If sub-list is not palindrome then no need to
       check for current left and right, return false */
   bool isp = isPalindromeUtil(left, right->next);
   if (isp == false)
      return false;

   /* Check values at current left and right */
   bool isp1 = (right->data == (*left)->data);

   /* Move left to next node */
   *left = (*left)->next;

   return isp1;
}

// A wrapper over isPalindromeUtil()
bool isPalindrome(struct node *head)
{
   isPalindromeUtil(&head, head);
}

/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref)    = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%c->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    char str[] = "abacaba";
```

```
    int i;

    for (i = 0; str[i] != '\0'; i++)
    {
       push(&head, str[i]);
       printList(head);
       isPalindrome(head)? printf("Is Palindrome\n\n"):
                           printf("Not Palindrome\n\n");
    }

    return 0;
}
```

Output:

```
a->NULL
Not Palindrome

b->a->NULL
Not Palindrome

a->b->a->NULL
Is Palindrome

c->a->b->a->NULL
Not Palindrome

a->c->a->b->a->NULL
Not Palindrome

b->a->c->a->b->a->NULL
Not Palindrome

a->b->a->c->a->b->a->NULL
Is Palindrome
```

Time Complexity: O(n) Auxiliary Space: O(n) if Function Call Stack size is considered, otherwise O(1).

12. Given a linked list which is sorted, how will you insert in sorted way

Algorithm:

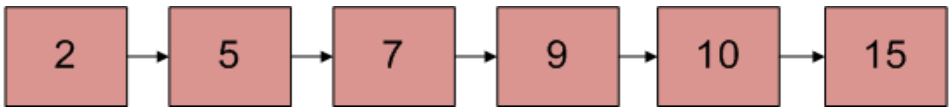
Let input linked list is sorted in increasing order.

```
1. If Linked list is empty then make the node as head and return it.
2. If value of the node to be inserted is smaller than value of head
   node then insert the node at start and make it head.
3. In a loop, find the appropriate node after which the input node
   (let 9) is to be inserted. To find the appropriate node start from
   head, keep moving until you reach a node GN (10 in the below
```

```
   diagram) who's value is greater than the input node. The node just
   before GN is the appropriate node (7).
4. Insert the node (9) after the appropriate node (7) found in step
   3.
```

Initial Linked List

Linked List after insertion of 9



Implementation:

```
/* Program to insert in a sorted list */
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function to insert a new_node in a list. Note that this
  function expects a pointer to head_ref as this can modify the
  head of the input linked list (similar to push())*/
void sortedInsert(struct node** head_ref, struct node* new_node)
{
    struct node* current;
    /* Special case for the head end */
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        current = *head_ref;
        while (current->next!=NULL &&
               current->next->data < new_node->data)
        {
            current = current->next;
        }
        new_node->next = current->next;
```

```
        current->next = new_node;
    }
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST sortedInsert */

/* A utility function to create a new node */
struct node *newNode(int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;
    new_node->next =  NULL;

    return new_node;
}

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d  ", temp->data);
        temp = temp->next;
    }
}

/* Drier program to test count function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    struct node *new_node = newNode(5);
    sortedInsert(&head, new_node);
    new_node = newNode(10);
    sortedInsert(&head, new_node);
    new_node = newNode(7);
    sortedInsert(&head, new_node);
    new_node = newNode(3);
    sortedInsert(&head, new_node);
    new_node = newNode(1);
    sortedInsert(&head, new_node);
    new_node = newNode(9);
    sortedInsert(&head, new_node);
    printf("\n Created Linked List\n");
    printList(head);

    getchar();
    return 0;
}
```

Shorter Implementation using double pointers

```
void sortedInsert(struct node** head_ref, struct node* new_node)
{
  if (head_ref == NULL)
  {
        return;
  }

      /* Locate the node before the point of insertion */
      struct node** current = head_ref;
      while (*current !=NULL && (*current)->data < data)
      {
        current = &((*current)->next);
      }

      new_node->next = *current;
      *current = new_node;
}
```

The code uses double pointer to keep track of the next pointer of the previous node (after which new node is being inserted).
Note that below line in code changes current to have address of next pointer in a node.

```
current = &((*current)->next);
```

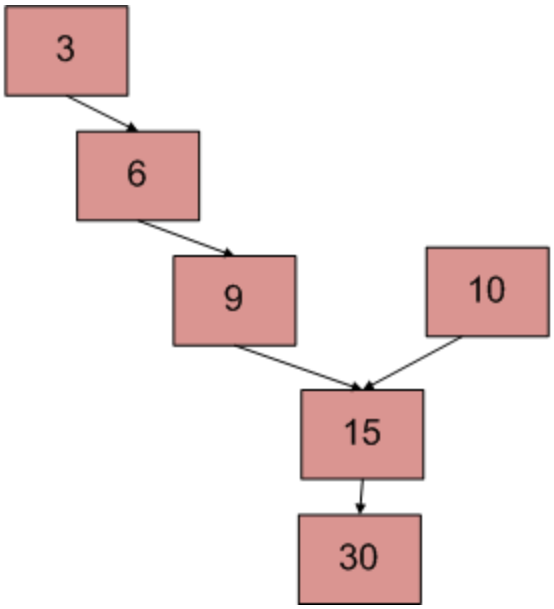
Also, note below comments.

```
new_node->next = *current; /* Copies the value-at-address current to
                              new_node's next pointer*/

*current = new_node;  /* Fix next pointer of the node
                         (using it's address) after which new_node is being inserted */
```

Time Complexity: O(n)

13. Write a function to get the intersection point of two Linked Lists.

There are two singly linked lists in a system. By some programming error the end node of one of the linked list got linked into the second list, forming a inverted Y shaped list. Write a program to get the point where two linked list merge.



Above diagram shows an example with two linked list having 15 as intersection point.

Method 1(Simply use two loops)

Use 2 nested for loops. Outer loop will be for each node of the 1st list and inner loop will be for 2nd list. In the inner loop, check if any of nodes of 2nd list is same as the current node of first linked list. Time complexity of this method will be O(mn) where m and n are the number of nodes in two lists.

Method 2 (Mark Visited Nodes)

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the first linked list and keep marking visited nodes. Now traverse second linked list, If you see a visited node again then there is an intersection point, return the intersecting node. This solution works in O(m+n) but requires additional information with each node. A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Traverse the first linked list and store the addresses of visited nodes in a hash. Now traverse the second linked list and if you see an address that already exists in hash then return the intersecting node.

Method 3(Using difference of node counts)

1) Get count of the nodes in first list, let count be c1.
2) Get count of the nodes in second list, let count be c2.
3) Get the difference of counts d = abs(c1 – c2)

4) Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes.
5) Then we can traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
  int data;
  struct node* next;
};

/* Function to get the counts of node in a linked list */
int getCount(struct node* head);

/* function to get the intersection point of two linked
   lists head1 and head2 where head1 has d more nodes than
   head2 */
int _getIntesectionNode(int d, struct node* head1, struct node* head2);

/* function to get the intersection point of two linked
   lists head1 and head2 */
int getIntesectionNode(struct node* head1, struct node* head2)
{
  int c1 = getCount(head1);
  int c2 = getCount(head2);
  int d;

  if(c1 > c2)
  {
    d = c1 - c2;
    return _getIntesectionNode(d, head1, head2);
  }
  else
  {
    d = c2 - c1;
    return _getIntesectionNode(d, head2, head1);
  }
}

/* function to get the intersection point of two linked
   lists head1 and head2 where head1 has d more nodes than
   head2 */
int _getIntesectionNode(int d, struct node* head1, struct node* head2)
{
  int i;
  struct node* current1 = head1;
  struct node* current2 = head2;

  for(i = 0; i < d; i++)
  {
```

```
    if(current1 == NULL)
    {  return -1; }
    current1 = current1->next;
  }

  while(current1 !=  NULL && current2 != NULL)
  {
    if(current1 == current2)
      return current1->data;
    current1= current1->next;
    current2= current2->next;
  }

  return -1;
}

/* Takes head pointer of the linked list and
   returns the count of nodes in the list */
int getCount(struct node* head)
{
  struct node* current = head;
  int count = 0;

  while (current != NULL)
  {
    count++;
    current = current->next;
  }

  return count;
}

/* IGNORE THE BELOW LINES OF CODE. THESE LINES
   ARE JUST TO QUICKLY TEST THE ABOVE FUNCTION */
int main()
{
  /*
    Create two linked lists

    1st 3->6->9->15->30
    2nd 10->15->30

    15 is the intersection point
  */

  struct node* newNode;
  struct node* head1 =
            (struct node*) malloc(sizeof(struct node));
  head1->data  = 10;

  struct node* head2 =
            (struct node*) malloc(sizeof(struct node));
  head2->data  = 3;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 6;
```

```
  head2->next = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 9;
  head2->next->next = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 15;
  head1->next = newNode;
  head2->next->next->next  = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 30;
  head1->next->next= newNode;

  head1->next->next->next = NULL;

  printf("\n The node of intersection is %d \n",
          getIntesectionNode(head1, head2));

  getchar();
}
```

Time Complexity: O(m+n) Auxiliary Space: O(1)

Method 4(Make circle in first list)

1. Traverse the first linked list(count the elements) and make a circular linked list. (Remember last node so that we can break the circle later on).
2. Now view the problem as find the loop in the second linked list. So the problem is solved.
3. Since we already know the length of the loop(size of first linked list) we can traverse those many number of nodes in second list, and then start another pointer from the beginning of second list. we have to traverse until they are equal, and that is the required intersection point.
4. remove the circle from the linked list.

Time Complexity: O(m+n) Auxiliary Space: O(1)

Method 5 (Reverse the first list and make equations)

```
1) Let X be the length of the first linked list until intersection point.
   Let Y be the length of the second linked list until the intersection
point.
   Let Z be the length of the linked list from intersection point to End
of
   the linked list including the intersection node.
   We Have
            X + Z = C1;
            Y + Z = C2;
2) Reverse first linked list.
```

```
3) Traverse Second linked list. Let C3 be the length of second list - 1.
     Now we have
        X + Y = C3
     We have 3 linear equations. By solving them, we get
       X = (C1 + C3 - C2)/2;
       Y = (C2 + C3 - C1)/2;
       Z = (C1 + C2 - C3)/2;
      WE GOT THE INTERSECTION POINT.
4)  Reverse first linked list.
```

Advantage: No Comparison of pointers.
Disadvantage: Modifying linked list(Reversing list).

Time complexity: O(m+n) Auxiliary Space: O(1)

Method 6 (Traverse both lists and compare addresses of last nodes)

This method is only to detect if there is an intersection point or not.

```
1) Traverse the list 1, store the last node address
2) Traverse the list 2, store the last node address.
3) If nodes stored in 1 and 2 are same then they are intersecting.
```

Time complexity of this method is O(m+n) and used Auxiliary space is O(1)

14. Remove duplicates from a sorted linked list

Write a removeDuplicates () function which takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates () should convert the list to 11->21->43->60.

Algorithm:
Traverse the list from the head (or start) node. While traversing, compare each node with its next node. If data of next node is same as current node then delete the next node. Before we delete a node, we need to store next pointer of the node

Implementation:
Functions other than removeDuplicates() are just to create a linked linked list and test removeDuplicates().

```
/*Program to remove duplicates from a sorted linked list */
#include<stdio.h>
#include<stlib.h>
```

```
/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* The function removes duplicates from a sorted list */
void removeDuplicates(struct node* head)
{
  /* Pointer to traverse the linked list */
  struct node* current = head;

  /* Pointer to store the next pointer of a node to be deleted*/
  struct node* next_next;

  /* do nothing if the list is empty */
  if(current == NULL)
     return;

  /* Traverse the list till last node */
  while(current->next != NULL)
  {
   /* Compare current node with next node */
    if(current->data == current->next->data)
    {
       /*The sequence of steps is important*/
      next_next = current->next->next;
      free(current->next);
      current->next = next_next;
    }
    else /* This is tricky: only advance if no deletion */
    {
      current = current->next;
    }
  }
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}
```

```
/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;

  /* Let us create a sorted linked list to test the functions
   Created linked list will be 11->11->11->13->13->20 */
  push(&head, 20);
  push(&head, 13);
  push(&head, 13);
  push(&head, 11);
  push(&head, 11);
  push(&head, 11);

  printf("\n Linked list before duplicate removal  ");
  printList(head);

  /* Remove duplicates from linked list */
  removeDuplicates(head);

  printf("\n Linked list after duplicate removal ");
  printList(head);

  getchar();
}
```

Time Complexity: O(n) where n is number of nodes in the given linked list.

15. Remove duplicates from an unsorted linked list

Write a removeDuplicates() function which takes a list and deletes any duplicate nodes from the list. The list is not sorted.

For example if the linked list is 12->11->12->21->41->43->21 then removeDuplicates() should convert the list to 12->11->21->41->43.

METHOD 1 (Using two loops)

This is the simple way where two loops are used. Outer loop is used to pick the elements one by one and inner loop compares the picked element with rest of the elements.

```
/* Program to remove duplicates in an unsorted array */
```

```
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
 int data;
 struct node *next;
};

/* Function to remove duplicates from a unsorted linked list */
void removeDuplicates(struct node *start)
{
  struct node *ptr1, *ptr2, *dup;
  ptr1 = start;

  /* Pick elements one by one */
  while(ptr1 != NULL && ptr1->next != NULL)
  {
     ptr2 = ptr1;

     /* Compare the picked element with rest of the elements */
     while(ptr2->next != NULL)
     {
       /* If duplicate then delete it */
       if(ptr1->data == ptr2->next->data)
       {
          /* sequence of steps is important here */
          dup = ptr2->next;
          ptr2->next = ptr2->next->next;
          free(dup);
       }
       else /* This is tricky */
       {
          ptr2 = ptr2->next;
       }
     }
     ptr1 = ptr1->next;
  }
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data);

/* Function to print nodes in a given linked list */
void printList(struct node *node);

/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   10->12->11->11->12->11->10*/
```

```
  push(&start, 10);
  push(&start, 11);
  push(&start, 12);
  push(&start, 11);
  push(&start, 11);
  push(&start, 12);
  push(&start, 10);

  printf("\n Linked list before removing duplicates ");
  printList(start);

  removeDuplicates(start);

  printf("\n Linked list after removing duplicates ");
  printList(start);

  getchar();
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}
```

Time Complexity: O(n^2)

METHOD 2 (Use Sorting)

In general, Merge Sort is the best suited sorting algorithm for sorting linked lists efficiently.

1) Sort the elements using Merge Sort. We will soon be writing a post about sorting a linked list. O(nLogn)

2) Remove duplicates in linear time using the algorithm for removing duplicates in sorted Linked List. O(n)

Please note that this method doesn't preserve the original order of elements.

Time Complexity: O(nLogn)

METHOD 3 (Use Hashing)

We traverse the link list from head to end. For every newly encountered element, we check whether it is in the hash table: if yes, we remove it; otherwise we put it in the hash table.

Time Complexity: O(n) on average (assuming that hash table access time is O(1) on average).

16. Pairwise swap elements of a given linked list

Given a singly linked list, write a function to swap elements pairwise. For example, if the linked list is 1->2->3->4->5 then the function should change it to 2->1->4->3->5, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5.

METHOD 1 (Iterative)

Start from the head node and traverse the list. While traversing swap data of each node with its next node's data.

```
/* Program to pairwise swap elements in a given linked list */
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
  int data;
  struct node *next;
};

/*Function to swap two integers at addresses a and b */
void swap(int *a, int *b);

/* Function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node *head)
{
  struct node *temp = head;

   /* Traverse further only if there are at-least two nodes left */
  while (temp != NULL && temp->next != NULL)
```

```
  {
    /* Swap data of node with its next node's data */
    swap(&temp->data, &temp->next->data);

    /* Move temp by 2 for the next pair */
    temp = temp->next->next;
  }
}

/* UTILITY FUNCTIONS */
/* Function to swap two integers */
void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while (node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   1->2->3->4->5 */
  push(&start, 5);
  push(&start, 4);
  push(&start, 3);
```

```
  push(&start, 2);
  push(&start, 1);

  printf("\n Linked list before calling  pairWiseSwap() ");
  printList(start);

  pairWiseSwap(start);

  printf("\n Linked list after calling  pairWiseSwap() ");
  printList(start);

  getchar();
  return 0;
}
```

Time complexity: O(n)

METHOD 2 (Recursive)

If there are 2 or more than 2 nodes in Linked List then swap the first two nodes and recursively call for rest of the list.

```
/* Recursive function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node *head)
{
  /* There must be at-least two nodes in the list */
  if(head != NULL && head->next != NULL)
  {
    /* Swap the node's data with data of next node */
    swap(&head->data, &head->next->data);

    /* Call pairWiseSwap() for rest of the list */
    pairWiseSwap(head->next->next);
  }
}
```

Time complexity: O(n)

The solution provided there swaps data of nodes. If data contains many fields, there will be many swap operations. See this for an implementation that changes links rather than swapping data.

17. Move last element to front of a given Linked List

Write a C function that moves last element to front in a given Singly Linked List. For example, if the given Linked List is 1->2->3->4->5, then the function should change the list to 5->1->2->3->4.

Algorithm:
Traverse the list till last node. Use two pointers: one to store the address of last node and other for address of second last node. After the end of loop do following operations.

1. Make second last as last (secLast->next = NULL).
2. Set next of last as head (last->next = *head_ref).
3. Make last as head (*head_ref = last)

```
/* Program to move last element to front in a given linked list */
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
 int data;
 struct node *next;
};

/* We are using a double pointer head_ref here because we change
   head of the linked list inside this function.*/
void moveToFront(struct node **head_ref)
{
  /* If linked list is empty, or it contains only one node,
    then nothing needs to be done, simply return */
  if(*head_ref == NULL || (*head_ref)->next == NULL)
    return;

  /* Initialize second last and last pointers */
  struct node *secLast = NULL;
  struct node *last = *head_ref;

  /*After this loop secLast contains address of second last
  node and last contains address of last node in Linked List */
  while(last->next != NULL)
  {
    secLast = last;
    last = last->next;
  }

  /* Set the next of second last as NULL */
  secLast->next = NULL;

  /* Set next of last as head node */
  last->next = *head_ref;

  /* Change the head pointer to point to last node now */
  *head_ref = last;
}

/* UTILITY FUNCTIONS */
/* Function to add a node at the begining of Linked List */
```

```
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}


/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   1->2->3->4->5 */
  push(&start, 5);
  push(&start, 4);
  push(&start, 3);
  push(&start, 2);
  push(&start, 1);

  printf("\n Linked list before moving last to front ");
  printList(start);

  moveToFront(&start);

  printf("\n Linked list after removing last to front ");
  printList(start);

  getchar();
}
```

Time Complexity: O(n) where n is the number of nodes in the given Linked List.

18. Intersection of two Sorted Linked Lists

Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed.

For example, let the first linked list be 1->2->3->4->6 and second linked list be 2->4->6->8, then your function should create and return a third list as 2->4->6.

Method 1 (Using Dummy Node)

The strategy here uses a temporary dummy node as the start of the result list. The pointer tail always points to the last node in the result list, so appending new nodes is easy. The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either 'a' or 'b', and adding it to tail. When we are done, the result is in dummy.next.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

void push(struct node** head_ref, int new_data);

/*This solution uses the temporary dummy to build up the result list */
struct node* sortedIntersect(struct node* a, struct node* b)
{
  struct node dummy;
  struct node* tail = &dummy;
  dummy.next = NULL;

  /* Once one or the other list runs out -- we're done */
  while (a != NULL && b != NULL)
  {
    if (a->data == b->data)
    {
       push((&tail->next), a->data);
       tail = tail->next;
       a = a->next;
       b = b->next;
    }
    else if (a->data < b->data) /* advance the smaller list */
       a = a->next;
    else
       b = b->next;
```

```
  }
  return(dummy.next);
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while (node != NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty lists */
  struct node* a = NULL;
  struct node* b = NULL;
  struct node *intersect = NULL;

  /* Let us create the first sorted linked list to test the functions
   Created linked list will be 1->2->3->4->5->6 */
  push(&a, 6);
  push(&a, 5);
  push(&a, 4);
  push(&a, 3);
  push(&a, 2);
  push(&a, 1);

  /* Let us create the second sorted linked list
   Created linked list will be 2->4->6->8 */
  push(&b, 8);
  push(&b, 6);
  push(&b, 4);
  push(&b, 2);
```

```
  /* Find the intersection two linked lists */
  intersect = sortedIntersect(a, b);

  printf("\n Linked list containing common items of a & b \n ");
  printList(intersect);

  getchar();
}
```

Time Complexity: O(m+n) where m and n are number of nodes in first and second linked lists respectively.

Method 2 (Using Local References)

This solution is structurally very similar to the above, but it avoids using a dummy node Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** "reference" strategy can be used

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

void push(struct node** head_ref, int new_data);

/* This solution uses the local reference */
struct node* sortedIntersect(struct node* a, struct node* b)
{
  struct node* result = NULL;
  struct node** lastPtrRef = &result;

  /* Advance comparing the first nodes in both lists.
    When one or the other list runs out, we're done. */
  while (a!=NULL && b!=NULL)
  {
    if (a->data == b->data)
    {
      /* found a node for the intersection */
      push(lastPtrRef, a->data);
      lastPtrRef = &((*lastPtrRef)->next);
      a = a->next;
      b = b->next;
    }
    else if (a->data < b->data)
      a=a->next;       /* advance the smaller list */
    else
```

```
      b=b->next;
  }
  return(result);
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty lists */
  struct node* a = NULL;
  struct node* b = NULL;
  struct node *intersect = NULL;

  /* Let us create the first sorted linked list to test the functions
   Created linked list will be 1->2->3->4->5->6 */
  push(&a, 6);
  push(&a, 5);
  push(&a, 4);
  push(&a, 3);
  push(&a, 2);
  push(&a, 1);

  /* Let us create the second sorted linked list
   Created linked list will be 2->4->6->8 */
  push(&b, 8);
  push(&b, 6);
  push(&b, 4);
  push(&b, 2);
```

```
  /* Find the intersection two linked lists */
  intersect = sortedIntersect(a, b);

  printf("\n Linked list containing common items of a & b \n ");
  printList(intersect);

  getchar();
}
```

Time Complexity: O(m+n) where m and n are number of nodes in first and second linked lists respectively.

Method 3 (Recursive)

Below is the recursive implementation of sortedIntersect().

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

struct node *sortedIntersect(struct node *a, struct node *b)
{
    /* base case */
    if (a == NULL || b == NULL)
        return NULL;

    /* If both lists are non-empty */

    /* advance the smaller list and call recursively */
    if (a->data < b->data)
        return sortedIntersect(a->next, b);

    if (a->data > b->data)
        return sortedIntersect(a, b->next);

    // Below lines are executed only when a->data == b->data
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->data = a->data;

    /* advance both lists and call recursively */
    temp->next = sortedIntersect(a->next, b->next);
    return temp;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
```

```
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty lists */
    struct node* a = NULL;
    struct node* b = NULL;
    struct node *intersect = NULL;

    /* Let us create the first sorted linked list to test the functions
     Created linked list will be 1->2->3->4->5->6 */
    push(&a, 6);
    push(&a, 5);
    push(&a, 4);
    push(&a, 3);
    push(&a, 2);
    push(&a, 1);

    /* Let us create the second sorted linked list
     Created linked list will be 2->4->6->8 */
    push(&b, 8);
    push(&b, 6);
    push(&b, 4);
    push(&b, 2);

    /* Find the intersection two linked lists */
    intersect = sortedIntersect(a, b);

    printf("\n Linked list containing common items of a & b \n ");
    printList(intersect);

    return 0;
}
```

Time Complexity: O(m+n) where m and n are number of nodes in first and second linked lists respectively.

19. Delete alternate nodes of a Linked List

Given a Singly Linked List, starting from the second node delete all alternate nodes of it. For example, if the given linked list is 1->2->3->4->5 then your function should convert it to 1->3->5, and if the given linked list is 1->2->3->4 then convert it to 1->3.

Method 1 (Iterative)

Keep track of previous of the node to be deleted. First change the next link of previous node and then free the memory allocated for the node.

```
#include<stdio.h>
#include<stdlib.h>

/* A linked list node */
struct node
{
  int data;
  struct node *next;
};

/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct node *head)
{
  if (head == NULL)
    return;

   /* Initialize prev and node to be deleted */
  struct node *prev = head;
  struct node *node = head->next;

  while (prev != NULL && node != NULL)
  {
     /* Change next link of previous node */
     prev->next = node->next;

     /* Free memory */
     free(node);

     /* Update prev and node */
     prev = prev->next;
     if(prev != NULL)
        node = prev->next;
  }
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
```

```
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Drier program to test above functions */
int main()
{
  int arr[100];

  /* Start with the empty list */
  struct node* head = NULL;

  /* Using push() to construct below list
    1->2->3->4->5  */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);

  printf("\n List before calling deleteAlt() ");
  printList(head);

  deleteAlt(head);

  printf("\n List after calling deleteAlt() ");
  printList(head);

  getchar();
  return 0;
}
```

Time Complexity: O(n) where n is the number of nodes in the given Linked List.

Method 2 (Recursive)

Recursive code uses the same approach as method 1. The recursive coed is simple and short, but causes O(n) recursive function calls for a linked list of size n.

```
/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct node *head)
{
  if (head == NULL)
    return;

  struct node *node = head->next;

  if (node == NULL)
    return;

  /* Change the next link of head */
  head->next = node->next;

  /* free memory allocated for node */
  free(node);

  /* Recursively call for the new next of head */
  deleteAlt(head->next);
}
```

Time Complexity: O(n)

20. Alternating split of a given Singly Linked List

Write a function AlternatingSplit() that takes one list and divides up its nodes to make two smaller lists ‘a’ and ‘b’. The sublists should be made from alternating elements in the original list. So if the original list is 0->1->0->1->0->1 then one sublist should be 0->0->0 and the other should be 1->1->1.

Method 1(Simple)

The simplest approach iterates over the source list and pull nodes off the source and alternately put them at the front (or beginning) of ‘a’ and b’. The only strange part is that the nodes will be in the reverse order that they occurred in the source list.

Method 2

Inserts the node at the end by keeping track of last node in sublists.

```
/*Program to alternatively split a linked list into two halves */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct node
{
```

```
    int data;
    struct node* next;
};

/* pull off the front node of the source and put it in dest */
void MoveNode(struct node** destRef, struct node** sourceRef) ;

/* Given the source list, split its nodes into two shorter lists.
  If we number the elements 0, 1, 2, ... then all the even elements
  should go in the first list, and all the odd elements in the second.
  The elements in the new lists may be in any order. */
void AlternatingSplit(struct node* source, struct node** aRef,
                              struct node** bRef)
{
  /* split the nodes of source to these 'a' and 'b' lists */
  struct node* a = NULL;
  struct node* b = NULL;

  struct node* current = source;
  while (current != NULL)
  {
    MoveNode(&a, &current); /* Move a node to list 'a' */
    if (current != NULL)
    {
       MoveNode(&b, &current); /* Move a node to list 'b' */
    }
  }
  *aRef = a;
  *bRef = b;
}

/* Take the node from the front of the source, and move it to the front of
the dest.
   It is an error to call this with the source list empty.

   Before calling MoveNode():
   source == {1, 2, 3}
   dest == {1, 2, 3}

   Affter calling MoveNode():
   source == {2, 3}
   dest == {1, 1, 2, 3}
*/
void MoveNode(struct node** destRef, struct node** sourceRef)
{
  /* the front source node  */
  struct node* newNode = *sourceRef;
  assert(newNode != NULL);

  /* Advance the source pointer */
  *sourceRef = newNode->next;

  /* Link the old dest off the new node */
  newNode->next = *destRef;

  /* Move dest to point to the new node */
```

```
  *destRef = newNode;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* head = NULL;
  struct node* a = NULL;
  struct node* b = NULL;

  /* Let us create a sorted linked list to test the functions
   Created linked list will be 0->1->2->3->4->5 */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);
  push(&head, 0);

  printf("\n Original linked List:  ");
  printList(head);

  /* Remove duplicates from linked list */
  AlternatingSplit(head, &a, &b);

  printf("\n Resultant Linked List 'a' ");
  printList(a);
```

```
  printf("\n Resultant Linked List 'b' ");
  printList(b);

  getchar();
  return 0;
}
```

Time Complexity: O(n) where n is number of node in the given linked list.

Method 2(Using Dummy Nodes)

Here is an alternative approach which builds the sub-lists in the same order as the source list. The code uses a temporary dummy header nodes for the ‘a’ and ‘b’ lists as they are being built. Each sublist has a “tail” pointer which points to its current last node — that way new nodes can be appended to the end of each list easily. The dummy nodes give the tail pointers something to point to initially. The dummy nodes are efficient in this case because they are temporary and allocated in the stack. Alternately, local “reference pointers” (which always points to the last pointer in the list instead of to the last node) could be used to avoid Dummy nodes.

```
void AlternatingSplit(struct node* source, struct node** aRef,
                              struct node** bRef)
{
  struct node aDummy;
  struct node* aTail = &aDummy; /* points to the last node in 'a' */
  struct node bDummy;
  struct node* bTail = &bDummy; /* points to the last node in 'b' */
  struct node* current = source;
  aDummy.next = NULL;
  bDummy.next = NULL;
  while (current != NULL)
  {
    MoveNode(&(aTail->next), &current); /* add at 'a' tail */
    aTail = aTail->next; /* advance the 'a' tail */
    if (current != NULL)
    {
      MoveNode(&(bTail->next), &current);
      bTail = bTail->next;
    }
  }
  *aRef = aDummy.next;
  *bRef = bDummy.next;
}
```

Time Complexity: O(n) where n is number of node in the given linked list.

21. Merge two sorted linked lists

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order.

SortedMerge() should return the new list. The new list should be made by splicing together the nodes of the first two lists.

For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20.

There are many cases to deal with: either ‘a’ or ‘b’ may be empty, during processing either ‘a’ or ‘b’ may run out first, and finally there’s the problem of starting the result list empty, and building it up while going through ‘a’ and ‘b’.

Method 1 (Using Dummy Nodes)

The strategy here uses a temporary dummy node as the start of the result list. The pointer Tail always points to the last node in the result list, so appending new nodes is easy. The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either ‘a’ or ‘b’, and adding it to tail. When we are done, the result is in dummy.next.

```
/*Program to alternatively split a linked list into two halves */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* pull off the front node of the source and put it in dest */
void MoveNode(struct node** destRef, struct node** sourceRef);

/* Takes two lists sorted in increasing order, and splices their nodes together
to make one big sorted list which is returned.  */
struct node* SortedMerge(struct node* a, struct node* b)
{
   /* a dummy first node to hang the result on */
   struct node dummy;

   /* tail points to the last result node  */
   struct node* tail = &dummy;

   /* so tail->next is the place to add new nodes
     to the result. */
   dummy.next = NULL;
   while(1)
```

```
   {
      if(a == NULL)
      {
         /* if either list runs out, use the other list */
         tail->next = b;
         break;
      }
      else if (b == NULL)
      {
         tail->next = a;
         break;
      }
      if (a->data <= b->data)
      {
         MoveNode(&(tail->next), &a);
      }
      else
     {
        MoveNode(&(tail->next), &b);
     }
     tail = tail->next;
  }
  return(dummy.next);
}


/* UTILITY FUNCTIONS */
/*MoveNode() function takes the node from the front of the source, and move it
to the front of the dest.
   It is an error to call this with the source list empty.

   Before calling MoveNode():
   source == {1, 2, 3}
   dest == {1, 2, 3}

   Affter calling MoveNode():
   source == {2, 3}
   dest == {1, 1, 2, 3}
*/
void MoveNode(struct node** destRef, struct node** sourceRef)
{
  /* the front source node  */
  struct node* newNode = *sourceRef;
  assert(newNode != NULL);

  /* Advance the source pointer */
  *sourceRef = newNode->next;

  /* Link the old dest off the new node */
  newNode->next = *destRef;

  /* Move dest to point to the new node */
  *destRef = newNode;
}
```

```
/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* res = NULL;
  struct node* a = NULL;
  struct node* b = NULL;

  /* Let us create two sorted linked lists to test the functions
   Created lists shall be a: 5->10->15,  b: 2->3->20 */
  push(&a, 15);
  push(&a, 10);
  push(&a, 5);

  push(&b, 20);
  push(&b, 3);
  push(&b, 2);

  /* Remove duplicates from linked list */
  res = SortedMerge(a, b);

  printf("\n Merged Linked List is: \n");
  printList(res);

  getchar();
  return 0;
}
```

Method 2 (Using Local References)

This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** "reference" strategy can be used (see Section 1 for details).

```
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* point to the last result pointer */
  struct node** lastPtrRef = &result;

  while(1)
  {
    if (a == NULL)
    {
      *lastPtrRef = b;
       break;
    }
    else if (b==NULL)
    {
       *lastPtrRef = a;
       break;
    }
    if(a->data <= b->data)
    {
      MoveNode(lastPtrRef, &a);
    }
    else
    {
      MoveNode(lastPtrRef, &b);
    }

    /* tricky: advance to point to the next ".next" field */
    lastPtrRef = &((*lastPtrRef)->next);
  }
  return(result);
}
```

Method 3 (Using Recursion)

Merge is one of those nice recursive problems where the recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists.

```
struct node* SortedMerge(struct node* a, struct node* b)
{
```

```
  struct node* result = NULL;

  /* Base cases */
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
  return(result);
}
```

22. Identical Linked Lists

Two Linked Lists are identical when they have same data and arrangement of data is also same. For example Linked lists a (1->2->3) and b(1->2->3) are identical. . Write a function to check if the given two linked lists are identical.

Method 1 (Iterative)

To identify if two lists are identical, we need to traverse both lists simultaneously, and while traversing we need to compare data.

```
#include<stdio.h>
#include<stdlib.h>

/* Structure for a linked list node */
struct node
{
  int data;
  struct node *next;
};

/* returns 1 if linked lists a and b are identical, otherwise 0 */
bool areIdentical(struct node *a, struct node *b)
{
  while(1)
  {
    /* base case */
    if(a == NULL && b == NULL)
    {  return 1; }
    if(a == NULL && b != NULL)
    {  return 0; }
    if(a != NULL && b == NULL)
    {  return 0; }
```

```
    if(a->data != b->data)
    {  return 0; }

    /* If we reach here, then a and b are not NULL and their
       data is same, so move to next nodes in both lists */
    a = a->next;
    b = b->next;
  }
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, push a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Druver program to test above function */
int main()
{
  struct node *a = NULL;
  struct node *b = NULL;

  /* The constructed linked lists are :
   a: 3->2->1
   b: 3->2->1 */
  push(&a, 1);
  push(&a, 2);
  push(&a, 3);

  push(&b, 1);
  push(&b, 2);
  push(&b, 3);

  if(areIdentical(a, b) == 1)
    printf(" Linked Lists are identical ");
  else
    printf(" Linked Lists are not identical ");

  getchar();
  return 0;
}
```

Method 2 (Recursive)

Recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists

```
bool areIdentical(struct node *a, struct node *b)
{
  if (a == NULL && b == NULL)
  {  return 1;  }
  if (a == NULL && b != NULL)
  {  return 0;  }
  if (a != NULL && b == NULL)
  {  return 0;  }
  if (a->data != b->data)
  {  return 0;  }

  /* If we reach here, then a and b are not NULL and their
       data is same, so move to next nodes in both lists */
  return areIdentical(a->next, b->next);
}
```

Time Complexity: O(n) for both iterative and recursive versions. n is the length of the smaller list among a and b.

23. Merge Sort for Linked Lists

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

MergeSort(headRef)

```
1) If head is NULL or there is only one element in the Linked List
    then return.
2) Else divide the linked list into two halves.
      FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
      MergeSort(a);
      MergeSort(b);
```

```
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
     *headRef = SortedMerge(a, b);
```

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
  struct node* head = *headRef;
  struct node* a;
  struct node* b;

  /* Base case -- length 0 or 1 */
  if ((head == NULL) || (head->next == NULL))
  {
    return;
  }

  /* Split head into 'a' and 'b' sublists */
  FrontBackSplit(head, &a, &b);

  /* Recursively sort the sublists */
  MergeSort(&a);
  MergeSort(&b);

  /* answer = merge the two sorted lists together */
  *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
   function */
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* Base cases */
```

```
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
  return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
     and return the two lists using the reference parameters.
     If the length is odd, the extra node should go in the front
list.
     Uses the fast/slow pointer strategy.  */
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef)
{
  struct node* fast;
  struct node* slow;
  if (source==NULL || source->next==NULL)
  {
    /* length < 2 cases */
    *frontRef = source;
    *backRef = NULL;
  }
  else
  {
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL)
    {
      fast = fast->next;
      if (fast != NULL)
      {
        slow = slow->next;
        fast = fast->next;
      }
    }
```

```
    /* 'slow' is before the midpoint in the list, so split it in
two
      at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
  }
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Function to insert a node at the beginging of the linked list
*/
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

  /* put in the data  */
  new_node->data  = new_data;

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* res = NULL;
  struct node* a = NULL;

  /* Let us create a unsorted linked lists to test the functions
   Created lists shall be a: 2->3->20->5->10->15 */
  push(&a, 15);
  push(&a, 10);
  push(&a, 5);
  push(&a, 20);
  push(&a, 3);
  push(&a, 2);
```

```
  /* Sort the above created Linked List */
  MergeSort(&a);

  printf("\n Sorted Linked List is: \n");
  printList(a);

  getchar();
  return 0;
}
```

Time Complexity: O(nLogn)

24. Reverse a Linked List in groups of given size

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

```
Example:
Inputs:  1->2->3->4->5->6->7->8->NULL and k = 3
Output:  3->2->1->6->5->4->8->7->NULL.

Inputs:   1->2->3->4->5->6->7->80->NULL and k = 5
Output:  5->4->3->2->1->8->7->6->NULL.
```

Algorithm: reverse(head, k)

1) Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be next and pointer to the previous node be prev. See this post for reversing a linked list.
2) head->next = reverse(next, k) /* Recursively call for rest of the list and link the two sub-lists */
3) return prev /* prev becomes the new head of the list */

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Reverses the linked list in groups of size k and returns the
   pointer to the new head node. */
struct node *reverse (struct node *head, int k)
{
    struct node* current = head;
    struct node* next = NULL;
    struct node* prev = NULL;
    int count = 0;
```

```
    /*reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
       next  = current->next;
       current->next = prev;
       prev = current;
       current = next;
       count++;
    }

    /* next is now a pointer to (k+1)th node
       Recursively call for the list starting from current.
       And make rest of the list as next of first node */
    if(next !=  NULL)
    {  head->next = reverse(next, k); }

    /* prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

     /* Created Linked list is 1->2->3->4->5->6->7->8 */
```

```
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("\n Given linked list \n");
    printList(head);
    head = reverse(head, 3);

    printf("\n Reversed Linked list \n");
    printList(head);

    getchar();
    return(0);
}
```

Time Complexity: O(n) where n is the number of nodes in the given list.

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

25.Reverse alternate K nodes in a Singly Linked List

Given a linked list, write a function to reverse every alternate k nodes (where k is an input to the function) in an efficient way. Give the complexity of your algorithm.

```
Example:
Inputs:   1->2->3->4->5->6->7->8->9->NULL and k = 3
Output:   3->2->1->4->5->6->9->8->7->NULL.
```

Method 1 (Process 2k nodes and recursively call for rest of the list)

This method is basically an extension of the method discussed in this post.

```
kAltReverse(struct node *head, int k)
  1)  Reverse first k nodes.
  2)  In the modified list head points to the kth node.  So change next
      of head to (k+1)th node
  3)  Move the current pointer to skip next k nodes.
  4)  Call the kAltReverse() recursively for rest of the n - 2k nodes.
  5)  Return new head of the list.
```

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
```

```
    struct node* next;
};

/* Reverses alternate k nodes and
   returns the pointer to the new head node */
struct node *kAltReverse(struct node *head, int k)
{
    struct node* current = head;
    struct node* next;
    struct node* prev = NULL;
    int count = 0;

    /*1) reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
       next  = current->next;
       current->next = prev;
       prev = current;
       current = next;
       count++;
    }

    /* 2) Now head points to the kth node.  So change next
       of head to (k+1)th node*/
    if(head != NULL)
      head->next = current;

    /* 3) We do not want to reverse next k nodes. So move the current
        pointer to skip next k nodes */
    count = 0;
    while(count < k-1 && current != NULL )
    {
      current = current->next;
      count++;
    }

    /* 4) Recursively call for the list starting from current->next.
       And make rest of the list as next of first node */
    if(current !=  NULL)
       current->next = kAltReverse(current->next, k);

    /* 5) prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;
```

```
    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    int count = 0;
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
        count++;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    // create a list 1->2->3->4->5...... ->20
    for(int i = 20; i > 0; i--)
      push(&head, i);

     printf("\n Given linked list \n");
     printList(head);
     head = kAltReverse(head, 3);

     printf("\n Modified Linked list \n");
     printList(head);

     getchar();
     return(0);
}
```

```
Output:
Given linked list
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
Modified Linked list
3 2 1 4 5 6 9 8 7 10 11 12 15 14 13 16 17 18 20 19
```

Time Complexity: O(n)

Method 2 (Process k nodes and recursively call for rest of the list)

The method 1 reverses the first k node and then moves the pointer to k nodes ahead. So method 1 uses two while loops and processes 2k nodes in one recursive call. This method processes only k nodes in a recursive call. It uses a third bool parameter b which decides whether to reverse the k elements or simply move the pointer.

```
_kAltReverse(struct node *head, int k, bool b)
  1)  If b is true, then reverse first k nodes.
  2)  If b is false, then move the pointer k nodes ahead.
  3)  Call the kAltReverse() recursively for rest of the n - k nodes and
link
       rest of the modified list with end of first k nodes.
Return new head of the list.
```

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Helper function for kAltReverse() */
struct node * _kAltReverse(struct node *node, int k, bool b);

/* Alternatively reverses the given linked list in groups of
   given size k. */
struct node *kAltReverse(struct node *head, int k)
{
  return _kAltReverse(head, k, true);
}

/*  Helper function for kAltReverse().  It reverses k nodes of the list
only if
    the third parameter b is passed as true, otherwise moves the pointer
k
    nodes ahead and recursively calls iteself  */
struct node * _kAltReverse(struct node *node, int k, bool b)
{
   if(node == NULL)
       return NULL;

   int count = 1;
   struct node *prev = NULL;
   struct node  *current = node;
   struct node *next;

   /* The loop serves two purposes
      1) If b is true, then it reverses the k nodes
      2) If b is false, then it moves the current pointer */
   while(current != NULL && count <= k)
   {
       next = current->next;

       /* Reverse the nodes only if b is true*/
       if(b == true)
          current->next = prev;
```

```
        prev = current;
        current = next;
        count++;
    }

    /* 3) If b is true, then node is the kth node.
        So attach rest of the list after node.
      4) After attaching, return the new head */
    if(b == true)
    {
         node->next = _kAltReverse(current,k,!b);
         return prev;
    }

    /* If b is not true, then attach rest of the list after prev.
      So attach rest of the list after prev */
    else
    {
         prev->next = _kAltReverse(current, k, !b);
         return node;
    }
}


/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    int count = 0;
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
        count++;
    }
}

/* Drier program to test above function*/
int main(void)
```

```
{
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    // create a list 1->2->3->4->5...... ->20
    for(i = 20; i > 0; i--)
      push(&head, i);

    printf("\n Given linked list \n");
    printList(head);
    head = kAltReverse(head, 3);

    printf("\n Modified Linked list \n");
    printList(head);

    getchar();
    return(0);
}
```

Output:

```
Given linked list
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
Modified Linked list
3 2 1 4 5 6 9 8 7 10 11 12 15 14 13 16 17 18 20 19
```

Time Complexity: O(n)

26. Delete nodes which have a greater value on right side

Given a singly linked list, remove all the nodes which have a greater value on right side.

Examples:

a) The list 12->15->10->11->5->6->2->3->NULL should be changed to 15->11->6->3->NULL. Note that 12, 10, 5 and 2 have been deleted because there is a greater value on the right side.

When we examine 12, we see that after 12 there is one node with value greater than 12 (i.e. 15), so we delete 12. When we examine 15, we find no node after 15 that has value greater than 15 so we keep this node.

When we go like this, we get 15->6->3

b) The list 10->20->30->40->50->60->NULL should be changed to 60->NULL. Note that 10, 20, 30, 40 and 50 have been deleted because they all have a greater value on the right side.

c) The list 60->50->40->30->20->10->NULL should not be changed.

Method 1 (Simple)

Use two loops. In the outer loop, pick nodes of the linked list one by one. In the inner loop, check if there exist a node whose value is greater than the picked node. If there exists a node whose value is greater, then delete the picked node. Time Complexity: O(n^2)

Method 2 (Use Reverse)

Reverse the list.

Traverse the reversed list. Keep max till now. If next node < max, then delete the next node, otherwise max = next node.

Reverse the list again to retain the original order.

Time Complexity: O(n)

```
#include <stdio.h>
#include <stdlib.h>

/* structure of a linked list node */
struct node
{
     int data;
     struct node *next;
};

/* prototype for utility functions */
void reverseList(struct node **headref);
void _delLesserNodes(struct node *head);

/* Deletes nodes which have a node with greater value node
  on left side */
void delLesserNodes(struct node **head_ref)
{
    /* 1) Reverse the linked list */
    reverseList(head_ref);

    /* 2) In the reversed list, delete nodes which have a node
       with greater value node on left side. Note that head
       node is never deleted because it is the leftmost node.*/
    _delLesserNodes(*head_ref);

    /* 3) Reverse the linked list again to retain the
       original order */
    reverseList(head_ref);
}

/* Deletes nodes which have greater value node(s) on left side */
void _delLesserNodes(struct node *head)
{
```

```
    struct node *current = head;

    /* Initialize max */
    struct node *maxnode = head;
    struct node *temp;

    while (current != NULL && current->next != NULL)
    {
        /* If current is smaller than max, then delete current */
        if(current->next->data < maxnode->data)
        {
            temp = current->next;
            current->next = temp->next;
            free(temp);
        }

        /* If current is greater than max, then update max and
           move current */
        else
        {
            current = current->next;
            maxnode = current;
        }

    }
}

/* Utility function to insert a node at the begining */
void push(struct node **head_ref, int new_data)
{
    struct node *new_node =
              (struct node *)malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

/* Utility function to reverse a linked list */
void reverseList(struct node **headref)
{
    struct node *current = *headref;
    struct node *prev = NULL;
    struct node *next;
    while(current != NULL)
    {
         next = current->next;
         current->next = prev;
         prev = current;
         current = next;
    }
    *headref = prev;
}

/* Utility function to print a linked list */
void printList(struct node *head)
{
```

```
     while(head!=NULL)
     {
        printf("%d ",head->data);
        head=head->next;
     }
     printf("\n");
}

/* Driver program to test above functions */
int main()
{
    struct node *head = NULL;

    /* Create following linked list
      12->15->10->11->5->6->2->3 */
    push(&head,3);
    push(&head,2);
    push(&head,6);
    push(&head,5);
    push(&head,11);
    push(&head,10);
    push(&head,15);
    push(&head,12);

    printf("Given Linked List: ");
    printList(head);

    delLesserNodes(&head);

    printf("\nModified Linked List: ");
    printList(head);

    getchar();
    return 0;
}
```

Output:

```
Given Linked List: 12 15 10 11 5 6 2 3
Modified Linked List: 15 11 6 3
```

27. Segregate even and odd nodes in a Linked List

Given a Linked List of integers, write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers same.

Examples:
Input: 17->15->8->12->10->5->4->1->7->6->NULL

Output: 8->12->10->4->6->17->15->5->1->7->NULL

Input: 8->12->10->5->4->1->6->NULL

Output: 8->12->10->4->6->5->1->NULL

// If all numbers are even then do not change the list

Input: 8->12->10->NULL

Output: 8->12->10->NULL

// If all numbers are odd then do not change the list

Input: 1->3->5->7->NULL

Output: 1->3->5->7->NULL

Method 1

The idea is to get pointer to the last node of list. And then traverse the list starting from the head node and move the odd valued nodes from their current position to end of the list.

Algorithm:

1) Get pointer to the last node.
2) Move all the odd nodes to the end.
   a) Consider all odd nodes before the first even node and move them to end.
   b) Change the head pointer to point to the first even node.
   c) Consider all odd nodes after the first even node and move them to the end.

```
#include <stdio.h>
#include <stdlib.h>

/* a node of the singly linked list */
struct node
{
    int data;
    struct node *next;
};

void segregateEvenOdd(struct node **head_ref)
{
    struct node *end = *head_ref;

    struct node *prev = NULL;
    struct node *curr = *head_ref;

    /* Get pointer to the last node */
    while (end->next != NULL)
        end = end->next;
```

```
    struct node *new_end = end;

    /* Consider all odd nodes before the first even node
       and move then after end */
    while (curr->data %2 != 0 && curr != end)
    {
        new_end->next = curr;
        curr = curr->next;
        new_end->next->next = NULL;
        new_end = new_end->next;
    }

    // 10->8->17->17->15
    /* Do following steps only if there is any even node */
    if (curr->data%2 == 0)
    {
        /* Change the head pointer to point to first even node */
        *head_ref = curr;

        /* now current points to the first even node */
        while (curr != end)
        {
            if ( (curr->data)%2 == 0 )
            {
                prev = curr;
                curr = curr->next;
            }
            else
            {
                /* break the link between prev and current */
                prev->next = curr->next;

                /* Make next of curr as NULL  */
                curr->next = NULL;

                /* Move curr to end */
                new_end->next = curr;

                /* make curr as new end of list */
                new_end = curr;

                /* Update current pointer to next of the moved node */
                curr = prev->next;
            }
        }
    }

    /* We must have prev set before executing lines following this
       statement */
    else prev = curr;

    /* If there are more than 1 odd nodes and end of original list is
      odd then move this node to end to maintain same order of odd
      numbers in modified list */
    if (new_end!=end && (end->data)%2 != 0)
    {
```

```
        prev->next = end->next;
        end->next = NULL;
        new_end->next = end;
    }
    return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List
*/
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Drier program to test above functions*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    /* Let us create a sample linked list as following
      0->2->4->6->8->10->11 */

    push(&head, 11);
    push(&head, 10);
    push(&head, 8);
    push(&head, 6);
    push(&head, 4);
    push(&head, 2);
    push(&head, 0);

    printf("\n Original Linked list ");
    printList(head);
```

```
    segregateEvenOdd(&head);

    printf("\n Modified Linked list ");
    printList(head);

    return 0;
}
```

Output:

```
 Original Linked list 0 2 4 6 8 10 11
 Modified Linked list 0 2 4 6 8 10 11
```

Time complexity: O(n)

Method 2

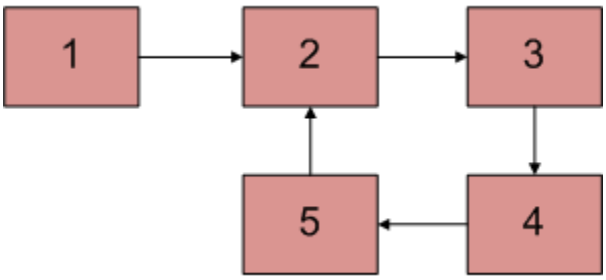
The idea is to split the linked list into two: one containing all even nodes and other containing all odd nodes. And finally attach the odd node linked list after the even node linked list.

To split the Linked List, traverse the original Linked List and move all odd nodes to a separate Linked List of all odd nodes. At the end of loop, the original list will have all the even nodes and the odd node list will have all the odd nodes. To keep the ordering of all nodes same, we must insert all the odd nodes at the end of the odd node list. And to do that in constant time, we must keep track of last pointer in the odd node list.

Time complexity: O(n)

28. Detect and Remove Loop in a Linked List

Write a function detectAndRemoveLoop() that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. detectAndRemoveLoop() must change the below list to 1->2->3->4->5->NULL.



Write a C function to detect loop in a linked list

Before trying to remove the loop, we must detect it. Techniques discussed in the above post can be used to detect loop. To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone.

We can easily use Hashing or Visited node techniques (discussed in the aobve mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.

We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

Method 1 (Check one by one)

We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say ptr2. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node  *slow_p = list, *fast_p = list;
```

```
    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

/* Function to remove loop.
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
   struct node *ptr1;
   struct node *ptr2;

   /* Set a pointer to the beging of the Linked List and
      move it one by one to find the first node which is
      part of the Linked List */
   ptr1 = head;
   while(1)
   {
     /* Now start a pointer from loop_node and check if it ever
       reaches ptr2 */
     ptr2 = loop_node;
     while(ptr2->next != loop_node && ptr2->next != ptr1)
     {
         ptr2 = ptr2->next;
     }

     /* If ptr2 reahced ptr1 then there is a loop. So break the
        loop */
     if(ptr2->next == ptr1)
        break;

     /* If ptr2 did't reach ptr1 then try the next node after ptr1 */
     else
       ptr1 = ptr1->next;
   }

   /* After the end of loop ptr2 is the last node of the loop. So
     make next of ptr2 as NULL */
   ptr2->next = NULL;
}
```

```
/* UTILITY FUNCTIONS */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, pushes a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 10);
    push(&head, 4);
    push(&head, 15);
    push(&head, 20);
    push(&head, 50);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    getchar();
    return 0;
}
```

Method 2 (Efficient Solution)

This method is also dependent on Floyd's Cycle detection algorithm.

1. Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.
2. Count the number of nodes in loop. Let the count be k.
3. Fix one pointer to the head and another to kth node from head.
4. Move both pointers at the same pace, they will meet at loop starting node.
5. Get pointer to the last node of loop and make next of it as NULL.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node  *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

/* Function to remove loop.
```

```
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1 = loop_node;
    struct node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for(i = 0; i < k; i++)
      ptr2 = ptr2->next;

    /*  Move both pointers at the same pace,
      they will meet at loop starting node */
    while(ptr2 != ptr1)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    ptr2 = ptr2->next;
    while(ptr2->next != ptr1)
       ptr2 = ptr2->next;

    /* Set the next node of the loop ending node
      to fix the loop */
    ptr2->next = NULL;
}

/* UTILITY FUNCTIONS */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, pushes a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);
```

```
    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 10);
    push(&head, 4);
    push(&head, 15);
    push(&head, 20);
    push(&head, 50);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    getchar();
    return 0;
}
```

29. Add two numbers represented by linked lists | Set 1

Given two numbers represented by two lists, write a function that returns sum list. The sum list is list representation of addition of two input numbers.

Example 1

```
Input:
  First List: 5->6->3  // represents number 365
  Second List: 8->4->2 //  represents number 248
Output
  Resultant list: 3->1->6  // represents number 613
```

Example 2

```
Input:
```

```
  First List: 7->5->9->4->6  // represents number 64957
  Second List: 8->4 //  represents number 48
Output
  Resultant list: 5->0->0->5->6  // represents number 65005
```

Solution

Traverse both lists. One by one pick nodes of both lists and add the values. If sum is more than 10 then make carry as 1 and reduce sum. If one list has more elements than the other then consider remaining values of this list as 0. Following is C implementation of this approach.

```
#include<stdio.h>
#include<stdlib.h>

/* Linked list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to create a new node with given data */
struct node *newNode(int data)
{
    struct node *new_node = (struct node *) malloc(sizeof(struct node));
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = newNode(new_data);

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Adds contents of two linked lists and return the head node of resultant
list */
struct node* addTwoLists (struct node* first, struct node* second)
{
    struct node* res = NULL; // res is head node of the resultant list
    struct node *temp, *prev = NULL;
    int carry = 0, sum;

    while (first != NULL || second != NULL) //while both lists exist
    {
        // Calculate value of next digit in resultant list.
```

```
        // The next digit is sum of following things
        // (i)  Carry
        // (ii) Next digit of first list (if there is a next digit)
        // (ii) Next digit of second list (if there is a next digit)
        sum = carry + (first? first->data: 0) + (second? second->data: 0);

        // update carry for next calulation
        carry = (sum >= 10)? 1 : 0;

        // update sum if it is greater than 10
        sum = sum % 10;

        // Create a new node with sum as data
        temp = newNode(sum);

        // if this is the first node then set it as head of the resultant
list
        if(res == NULL)
            res = temp;
        else // If this is not the first node then connect it to the rest.
            prev->next = temp;

        // Set prev for next insertion
        prev  = temp;

        // Move first and second pointers to next nodes
        if (first) first = first->next;
        if (second) second = second->next;
    }

    if (carry > 0)
      temp->next = newNode(carry);

    // return head of the resultant list
    return res;
}

// A utility function to print a linked list
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Drier program to test above function */
int main(void)
{
    struct node* res = NULL;
    struct node* first = NULL;
    struct node* second = NULL;

    // create first list 7->5->9->4->6
```

```
    push(&first, 6);
    push(&first, 4);
    push(&first, 9);
    push(&first, 5);
    push(&first, 7);
    printf("First List is ");
    printList(first);

    // create second list 8->4
    push(&second, 4);
    push(&second, 8);
    printf("Second List is ");
    printList(second);

    // Add the two lists and see result
    res = addTwoLists(first, second);
    printf("Resultant list is ");
    printList(res);

   return 0;
}
```

Output:

```
First List is 7 5 9 4 6
Second List is 8 4
Resultant list is 5 0 0 5 6
```

Time Complexity: O(m + n) where m and n are number of nodes in first and second lists respectively.

30. Delete a given node in Linked List under given constraints

Given a Singly Linked List, write a function to delete a given node. Your function must follow following constraints:
1) It must accept pointer to the start node as first parameter and node to be deleted as second parameter i.e., pointer to head node is not global.
2) It should not return pointer to the head node.
3) It should not accept pointer to pointer to head node.

You may assume that the Linked List never becomes empty.

Let the function name be deleteNode(). In a straightforward implementation, the function needs to modify head pointer when the node

to be deleted is first node. As discussed in previous post, when a function modifies the head pointer, the function must use one of the given approaches, we can't use any of those approaches here.

Solution

We explicitly handle the case when node to be deleted is first node, we copy the data of next node to head and delete the next node. The cases when deleted node is not the head node can be handled normally by finding the previous node and changing next of previous node. Following is C implementation.

```
#include <stdio.h>
#include <stdlib.h>

/* structure of a linked list node */
struct node
{
    int data;
    struct node *next;
};

void deleteNode(struct node *head, struct node *n)
{
    // When node to be deleted is head node
    if(head == n)
    {
        if(head->next == NULL)
        {
            printf("There is only one node. The list can't be made empty
");
            return;
        }

        /* Copy the data of next node to head */
        head->data = head->next->data;

        // store address of next node
        n = head->next;

        // Remove the link of next node
        head->next = head->next->next;

        // free memory
        free(n);

        return;
    }
```

```
    // When not first node, follow the normal deletion process

    // find the previous node
    struct node *prev = head;
    while(prev->next != NULL && prev->next != n)
        prev = prev->next;

    // Check if node really exists in Linked List
    if(prev->next == NULL)
    {
        printf("\n Given node is not present in Linked List");
        return;
    }

    // Remove node from Linked List
    prev->next = prev->next->next;

    // Free memory
    free(n);

    return;
}

/* Utility function to insert a node at the begining */
void push(struct node **head_ref, int new_data)
{
    struct node *new_node =
        (struct node *)malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

/* Utility function to print a linked list */
void printList(struct node *head)
{
    while(head!=NULL)
    {
        printf("%d ",head->data);
        head=head->next;
    }
    printf("\n");
}

/* Driver program to test above functions */
int main()
{
    struct node *head = NULL;

    /* Create following linked list
```

```
      12->15->10->11->5->6->2->3 */
    push(&head,3);
    push(&head,2);
    push(&head,6);
    push(&head,5);
    push(&head,11);
    push(&head,10);
    push(&head,15);
    push(&head,12);

    printf("Given Linked List: ");
    printList(head);

    /* Let us delete the node with value 10 */
    printf("\nDeleting node %d: ", head->next->next->data);
    deleteNode(head, head->next->next);

    printf("\nModified Linked List: ");
    printList(head);

    /* Let us delete the the first node */
    printf("\nDeleting first node ");
    deleteNode(head, head);

    printf("\nModified Linked List: ");
    printList(head);

    getchar();
    return 0;
}
```

Output:

```
Given Linked List: 12 15 10 11 5 6 2 3

Deleting node 10:
Modified Linked List: 12 15 11 5 6 2 3

Deleting first node
Modified Linked List: 15 11 5 6 2 3
```

31. Union and Intersection of two Linked Lists

Given two Linked Lists, create union and intersection lists that contain union and intersection of the elements present in the given lists. Order of elments in output lists doesn't matter.

Example:

```
Input:
   List1: 10->15->4->20
   lsit2:  8->4->2->10
Output:
   Intersection List: 4->10
   Union List: 2->8->20->4->15->10
```

Method 1 (Simple)
Following are simple algorithms to get union and intersection lists respectively.

Intersection (list1, list2)
Initialize result list as NULL. Traverse list1 and look for its each element in list2, if the element is present in list2, then add the element to result.

Union *(list1, list2):*
Initialize result list as NULL. Traverse list1 and add all of its elements to the result.
Traverse list2. If an element of list2 is already present in result then do not insert it to result, otherwise insert.

This method assumes that there are no duplicates in the given lists.

Thanks to Shekhu for suggesting this method. Following is C implementation of this method.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* A utility function to insert a node at the begining of a linked
list*/
void push (struct node** head_ref, int new_data);

/* A utilty function to chec if given data is present in a list */
bool isPresent (struct node *head, int data);

/* Function to get union of two linked lists head1 and head2 */
struct node *getUnion (struct node *head1, struct node *head2)
{
```

```
    struct node *result = NULL;
    struct node *t1 = head1, *t2 = head2;

    // Insert all elements of list1 to the result list
    while (t1 != NULL)
    {
        push(&result, t1->data);
        t1 = t1->next;
    }

    // Insert those elements of list2 which are not present in result
list
    while (t2 != NULL)
    {
        if (!isPresent(result, t2->data))
            push(&result, t2->data);
        t2 = t2->next;
    }

    return result;
}

/* Function to get intersection of two linked lists head1 and head2 */
struct node *getIntersection (struct node *head1, struct node *head2)
{
    struct node *result = NULL;
    struct node *t1 = head1;

    // Traverse list1 and search each element of it in list2. If the
element
    // is present in list 2, then insert the element to result
    while (t1 != NULL)
    {
        if (isPresent(head2, t1->data))
            push (&result, t1->data);
        t1 = t1->next;
    }

    return result;
}

/* A utility function to insert a node at the begining of a linked
list*/
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
```

```
    (*head_ref) = new_node;
}

/* A utility function to print a linked list*/
void printList (struct node *node)
{
    while (node != NULL)
    {
        printf ("%d ", node->data);
        node = node->next;
    }
}

/* A utilty function that returns true if data is present in linked
list
else return false */
bool isPresent (struct node *head, int data)
{
    struct node *t = head;
    while (t != NULL)
    {
        if (t->data == data)
            return 1;
        t = t->next;
    }
    return 0;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head1 = NULL;
    struct node* head2 = NULL;
    struct node* intersecn = NULL;
    struct node* unin = NULL;

    /*create a linked lits 10->15->5->20 */
    push (&head1, 20);
    push (&head1, 4);
    push (&head1, 15);
    push (&head1, 10);

    /*create a linked lits 8->4->2->10 */
    push (&head2, 10);
    push (&head2, 2);
    push (&head2, 4);
    push (&head2, 8);

    intersecn = getIntersection (head1, head2);
    unin = getUnion (head1, head2);

    printf ("\n First list is \n");
    printList (head1);

    printf ("\n Second list is \n");
```

```
    printList (head2);

    printf ("\n Intersection list is \n");
    printList (intersecn);

    printf ("\n Union list is \n");
    printList (unin);

    return 0;
}
```

Output:

```
 First list is
10 15 4 20
 Second list is
8 4 2 10
 Intersection list is
4 10
 Union list is
2 8 20 4 15 10
```

Time Complexity: O(mn) for both union and intersection operations. Here m is the number of elements in first list and n is the number of elements in second list.

Method 2 (Use Merge Sort)
In this method, algorithms for Union and Intersection are very similar. First we sort the given lists, then we traverse the sorted lists to get union and intersection.
Following are the steps to be followed to get union and intersection lists.

1) Sort the first Linked List using merge sort. This step takes O(mLogm) time. Refer this post for details of this step.
2) Sort the second Linked List using merge sort. This step takes O(nLogn) time. Refer this post for details of this step.
3) Linearly scan both sorted lists to get the union and intersection. This step takes O(m + n) time. This step can be implemented using the same algorithm as sorted arrays algorithm discussedhere.

Time complexity of this method is O(mLogm + nLogn) which is better than method 1's time complexity.

Method 3 (Use Hashing)
Union (list1, list2)
Initialize the result list as NULL and create an empty hash table. Traverse both lists one by one, for each element being visited, look the element in hash table. If the element is not present, then insert the element to result list. If the element is present, then ignore it.

Intersection (list1, list2)
Initialize the result list as NULL and create an empty hash table. Traverse list1. For each element being visited in list1, insert the element in hash table. Traverse list2, for each element being visited in list2, look the element in hash table. If the element is present, then insert the element to result list. If the element is not present, then ignore it.

Both of the above methods assume that there are no duplicates.

Time complexity of this method depends on the hashing technique used and the distribution of elements in input lists. In practical, this approach may turn out to be better than above 2 methods.

32. Find a triplet from three linked lists with sum equal to a given number

Given three linked lists, say a, b and c, find one node from each list such that the sum of the values of the nodes is equal to a given number.
For example, if the three linked lists are 12->6->29, 23->5->8 and 90->20->59, and the given number is 101, the output should be tripel "6 5 90″.

In the following solutions, size of all three linked lists is assumed same for simplicity of analysis. The following solutions work for linked lists of different sizes also.

A simple method to solve this problem is to run three nested loops. The outermost loop picks an element from list a, the middle loop picks an element from b and the innermost loop picks from c. The innermost loop also checks whether the sum of values of current

nodes of a, b and c is equal to given number. The time complexity of this method will be O(n^3).

Sorting can be used to reduce the time complexity to O(n*n). Following are the detailed steps.

1) Sort list b in ascending order, and list c in descending order.
2) After the b and c are sorted, one by one pick an element from list a and find the pair by traversing both b and c. See isSumSorted() in the following code. The idea is similar to Quadratic algorithm of 3 sum problem.

Following code implements step 2 only. The solution can be easily modified for unsorted lists by adding the merge sort code discussed here.

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* A utility function to insert a node at the begining of a linked list*/
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* A function to chech if there are three elements in a, b and c whose
sum
   is equal to givenNumber.  The function assumes that the list b is
sorted
   in ascending order and c is sorted in descending order. */
bool isSumSorted(struct node *headA, struct node *headB,
                 struct node *headC, int givenNumber)
{
    struct node *a = headA;
```

```
    // Traverse through all nodes of a
    while (a != NULL)
    {
        struct node *b = headB;
        struct node *c = headC;

        // For every node of list a, prick two nodes from lists b abd c
        while (b != NULL && c != NULL)
        {
            // If this a triplet with given sum, print it and return
true
            int sum = a->data + b->data + c->data;
            if (sum == givenNumber)
            {
               printf ("Triplet Found: %d %d %d ", a->data, b->data, c-
>data);
               return true;
            }

            // If sum of this triplet is smaller, look for greater values
in b
            else if (sum < givenNumber)
                b = b->next;
            else // If sum is greater, look for smaller values in c
                c = c->next;
        }
        a = a->next;  // Move ahead in list a
    }

    printf ("No such triplet");
    return false;
}


/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* headA = NULL;
    struct node* headB = NULL;
    struct node* headC = NULL;

    /*create a linked list 'a' 10->15->5->20 */
    push (&headA, 20);
    push (&headA, 4);
    push (&headA, 15);
    push (&headA, 10);

    /*create a sorted linked list 'b' 2->4->9->10 */
    push (&headB, 10);
    push (&headB, 9);
    push (&headB, 4);
    push (&headB, 2);

    /*create another sorted linked list 'c' 8->4->2->1 */
    push (&headC, 1);
```

```
    push (&headC, 2);
    push (&headC, 4);
    push (&headC, 8);

    int givenNumber = 25;

    isSumSorted (headA, headB, headC, givenNumber);

    return 0;
}
```

Output:

```
Triplet Found: 15 2 8
```

Time complexity: The linked lists b and c can be sorted in O(nLogn) time using Merge Sort (Seethis). The step 2 takes O(n*n) time. So the overall time complexity is O(nlogn) + O(nlogn) + O(n*n) = O(n*n).

In this approach, the linked lists b and c are sorted first, so their original order will be lost. If we want to retain the original order of b and c, we can create copy of b and c.

33. Rotate a Linked List

Given a singly linked list, rotate the linked list counter-clockwise by k nodes. Where k is a given positive integer. For example, if the given linked list is 10->20->30->40->50->60 and k is 4, the list should be modified to 50->60->10->20->30->40. Assume that k is smaller than the count of nodes in linked list.

To rotate the linked list, we need to change next of kth node to NULL, next of last node to previous head node, and finally change head to (k+1)th node. So we need to get hold of three nodes: kth node, (k+1)th node and last node. Traverse the list from beginning and stop at kth node. Store pointer to kth node. We can get (k+1)th node using kthNode->next. Keep traversing till end and store pointer to last node also. Finally, change pointers as stated above.

```
// Program to rotate a linked list counter clock wise
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

// This function rotates a linked list counter-clockwise and updates the
head.
```

```
// The function assumes that k is smaller than size of linked list. It
doesn't
// modify the list if k is greater than or equal to size
void rotate (struct node **head_ref, int k)
{
     if (k == 0)
       return;

    // Let us understand the below code for example k = 4 and
    // list = 10->20->30->40->50->60.
    struct node* current = *head_ref;

    // current will either point to kth or NULL after this loop.
    //  current will point to node 40 in the above example
    int count = 1;
    while (count < k && current != NULL)
    {
        current = current->next;
        count++;
    }

    // If current is NULL, k is greater than or equal to count
    // of nodes in linked list. Don't change the list in this case
    if (current == NULL)
        return;

    // current points to kth node. Store it in a variable.
    // kthNode points to node 40 in the above example
    struct node *kthNode = current;

    // current will point to last node after this loop
    // current will point to node 60 in the above example
    while (current->next != NULL)
        current = current->next;

    // Change next of last node to previous head
    // Next of 60 is now changed to node 10
    current->next = *head_ref;

    // Change head to (k+1)th node
    // head is now changed to node 50
    *head_ref = kthNode->next;

    // change next of kth node to NULL
    // next of 40 is now NULL
    kthNode->next = NULL;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));
```

```
    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    // create a list 10->20->30->40->50->60
    for (int i = 60; i > 0; i -= 10)
        push(&head, i);

    printf("Given linked list \n");
    printList(head);
    rotate(&head, 4);

    printf("\nRotated Linked list \n");
    printList(head);

    return (0);
}
```

Output:

```
Given linked list
10  20  30  40  50  60
Rotated Linked list
50  60  10  20  30  40
```

Time Complexity: O(n) where n is the number of nodes in Linked List. The code traverses the linked list only once

34. Flattening a Linked List

Given a linked list where every node represents a linked list and contains two pointers of its type:

(i) Pointer to next node in the main list (we call it 'right' pointer in below code)
(ii) Pointer to a linked list where this node is head (we call it 'down' pointer in below code).
All linked lists are sorted. See the following example

```
5 -> 10 -> 19 -> 28
|     |      |     |
V     V      V     V
7     20     22    35
|            |     |
V            V     V
8            50    40
|                  |
V                  V
30                 45
```

Write a function flatten() to flatten the lists into a single linked list. The flattened linked list should also be sorted. For example, for the above input list, output list should be 5->7->8->10->19->20->22->28->30->35->40->45->50.

The idea is to use Merge() process of merge sort for linked lists. We use merge() to merge lists one by one. We recursively merge() the current list with already flattened list.
The down pointer is used to link nodes of the flattened list.

Following is C implementation.

```
#include <stdio.h>
#include <stdlib.h>

// A Linked List Node
typedef struct Node
{
    int data;
    struct Node *right;
    struct Node *down;
} Node;

/* A utility function to insert a new node at the begining
   of linked list */
void push (Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = (Node *) malloc(sizeof(Node));
```

```
    new_node->right = NULL;

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->down = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in the flattened linked list */
void printList(Node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->down;
    }
}

// A utility function to merge two sorted linked lists
Node* merge( Node* a, Node* b )
{
    // If first list is empty, the second list is result
    if (a == NULL)
        return b;

    // If second list is empty, the second list is result
    if (b == NULL)
        return a;

    // Compare the data members of head nodes of both lists
    // and put the smaller one in result
    Node* result;
    if( a->data < b->data )
    {
        result = a;
        result->down = merge( a->down, b );
    }
    else
    {
        result = b;
        result->down = merge( a, b->down );
    }

    return result;
}

// The main function that flattens a given linked list
Node* flatten (Node* root)
{
    // Base cases
    if ( root == NULL || root->right == NULL )
        return root;
```

```
    // Merge this list with the list on right side
    return merge( root, flatten(root->right) );
}

// Driver program to test above functions
int main()
{
    Node* root = NULL;

    /* Let us create the following linked list
       5 -> 10 -> 19 -> 28
       |    |     |     |
       V    V     V     V
       7    20    22    35
       |          |     |
       V          V     V
       8          50    40
       |                |
       V                V
       30               45
    */
    push( &root, 30 );
    push( &root, 8 );
    push( &root, 7 );
    push( &root, 5 );

    push( &( root->right ), 20 );
    push( &( root->right ), 10 );

    push( &( root->right->right ), 50 );
    push( &( root->right->right ), 22 );
    push( &( root->right->right ), 19 );

    push( &( root->right->right->right ), 45 );
    push( &( root->right->right->right ), 40 );
    push( &( root->right->right->right ), 35 );
    push( &( root->right->right->right ), 20 );

    // Let us flatten the list
    root = flatten(root);

    // Let us print the flatened linked list
    printList(root);

    return 0;
}
```

Output:

```
5 7 8 10 19 20 20 22 30 35 40 45 50
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

35. Add two numbers represented by linked lists | Set 2

Given two numbers represented by two linked lists, write a function that returns sum list. The sum list is linked list representation of addition of two input numbers. It is not allowed to modify the lists. Also, not allowed to use explicit extra space (Hint: Use Recursion).

Example

```
Input:
  First List: 5->6->3  // represents number 563
  Second List: 8->4->2 //  represents number 842
Output
  Resultant list: 1->4->0->5  // represents number 1405
```

We have discussed a solution here which is for linked lists where least significant digit is first node of lists and most significant digit is last node. In this problem, most significant node is first node and least significant digit is last node and we are not allowed to modify the lists. Recursion is used here to calculate sum from right to left.

Following are the steps.

1) Calculate sizes of given two linked lists.
2) If sizes are same, then calculate sum using recursion. Hold all nodes in recursion call stack till the rightmost node, calculate sum of rightmost nodes and forward carry to left side.
3) If size is not same, then follow below steps:
    a) Calculate difference of sizes of two linked lists. Let the difference be diff
    b) Move diff nodes ahead in the bigger linked list. Now use step 2 to calculate sum of smaller list and right sub-list (of same size) of larger list. Also, store the carry of this sum.
    c) Calculate sum of the carry (calculated in previous step) with the remaining left sub-list of larger list. Nodes of this sum are added at the beginning of sum list obtained previous step.

Following is C implementation of the above approach.

```
// A recursive program to add two linked lists

#include <stdio.h>
#include <stdlib.h>

// A linked List Node
struct node
```

```
{
    int data;
    struct node* next;
};

typedef struct node node;

/* A utility function to insert a node at the beginning of linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* A utility function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
    printf("\n");
}

// A utility function to swap two pointers
void swapPointer( node** a, node** b )
{
    node* t = *a;
    *a = *b;
    *b = t;
}

/* A utility function to get size of linked list */
int getSize(struct node *node)
{
    int size = 0;
    while (node != NULL)
    {
        node = node->next;
        size++;
    }
    return size;
}

// Adds two linked lists of same size represented by head1 and head2 and
returns
```

```
// head of the resultant linked list. Carry is propagated while returning
from
// the recursion
node* addSameSize(node* head1, node* head2, int* carry)
{
    // Since the function assumes linked lists are of same size,
    // check any of the two head pointers
    if (head1 == NULL)
        return NULL;

    int sum;

    // Allocate memory for sum node of current two nodes
    node* result = (node *)malloc(sizeof(node));

    // Recursively add remaining nodes and get the carry
    result->next = addSameSize(head1->next, head2->next, carry);

    // add digits of current nodes and propagated carry
    sum = head1->data + head2->data + *carry;
    *carry = sum / 10;
    sum = sum % 10;

    // Assigne the sum to current node of resultant list
    result->data = sum;

    return result;
}

// This function is called after the smaller list is added to the bigger
// lists's sublist of same size.  Once the right sublist is added, the
carry
// must be added toe left side of larger list to get the final result.
void  addCarryToRemaining(node*  head1,  node*  cur,  int*  carry,  node**
result)
{
    int sum;

    // If diff. number of nodes are not traversed, add carry
    if (head1 != cur)
    {
        addCarryToRemaining(head1->next, cur, carry, result);

        sum = head1->data + *carry;
        *carry = sum/10;
        sum %= 10;

        // add this node to the front of the result
        push(result, sum);
    }
}

// The main function that adds two linked lists represented by head1 and
head2.
// The sum of two lists is stored in a list referred by result
void addList(node* head1, node* head2, node** result)
```

```
{
    node *cur;

    // first list is empty
    if (head1 == NULL)
    {
        *result = head2;
        return;
    }

    // second list is empty
    else if (head2 == NULL)
    {
        *result = head1;
        return;
    }

    int size1 = getSize(head1);
    int size2 = getSize(head2) ;

    int carry = 0;

    // Add same size lists
    if (size1 == size2)
        *result = addSameSize(head1, head2, &carry);

    else
    {
        int diff = abs(size1 - size2);

        // First list should always be larger than second list.
        // If not, swap pointers
        if (size1 < size2)
            swapPointer(&head1, &head2);

        // move diff. number of nodes in first list
        for (cur = head1; diff--; cur = cur->next);

        // get addition of same size lists
        *result = addSameSize(cur, head2, &carry);

        // get addition of remaining first list and carry
        addCarryToRemaining(head1, cur, &carry, result);
    }

    // if some carry is still there, add a new node to the fron of
    // the result list. e.g. 999 and 87
    if (carry)
        push(result, carry);
}

// Driver program to test above functions
int main()
{
    node *head1 = NULL, *head2 = NULL, *result = NULL;
```

```
    int arr1[] = {9, 9, 9};
    int arr2[] = {1, 8};

    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);

    // Create first list as 9->9->9
    int i;
    for (i = size1-1; i >= 0; --i)
        push(&head1, arr1[i]);

    // Create second list as 1->8
    for (i = size2-1; i >= 0; --i)
        push(&head2, arr2[i]);

    addList(head1, head2, &result);

    printList(result);

    return 0;
}
```

Output:

```
1  0  1  7
```

Time Complexity: O(m+n) where m and n are the sizes of given two linked lists.

36. Sort a linked list of 0s, 1s and 2s

Given a linked list of 0s, 1s and 2s, sort it.

Following steps can be used to sort the given linked list.

1) Traverse the list and count the number of 0s, 1s and 2s. Let the counts be n1, n2 and n3 respectively.
2) Traverse the list again, fill the first n1 nodes with 0, then n2 nodes with 1 and finally n3 nodes with 2.

```
// Program to sort a linked list 0s, 1s or 2s
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

// Function to sort a linked list of 0s, 1s and 2s
void sortList(struct node *head)
{
```

```
    int count[3] = {0, 0, 0};  // Initialize count of '0', '1' and '2'
as 0
    struct node *ptr = head;

    /* count total number of '0', '1' and '2'
     * count[0] will store total number of '0's
     * count[1] will store total number of '1's
     * count[2] will store total number of '2's  */
    while (ptr != NULL)
    {
        count[ptr->data] += 1;
        ptr = ptr->next;
    }

    int i = 0;
    ptr = head;

    /* Let say count[0] = n1, count[1] = n2 and count[2] = n3
     * now start traversing list from head node,
     * 1) fill the list with 0, till n1 > 0
     * 2) fill the list with 1, till n2 > 0
     * 3) fill the list with 2, till n3 > 0  */
    while (ptr != NULL)
    {
        if (count[i] == 0)
            ++i;
        else
        {
            ptr->data = i;
            --count[i];
            ptr = ptr->next;
        }
    }
}

/* Function to push a node */
void push (struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
```

```
    {
        printf("%d  ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Drier program to test above function*/
int main(void)
{
    struct node *head = NULL;
    push(&head, 0);
    push(&head, 1);
    push(&head, 0);
    push(&head, 2);
    push(&head, 1);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);
    push(&head, 2);

    printf("Linked List Before Sorting\n");
    printList(head);

    sortList(head);

    printf("Linked List After Sorting\n");
    printList(head);

    return 0;
}
```

Output:

```
Linked List Before Sorting
2  1  2  1  1  2  0  1  0
Linked List After Sorting
0  0  1  1  1  1  2  2  2
```

Time Complexity: O(n) Auxiliary Space: O(1)

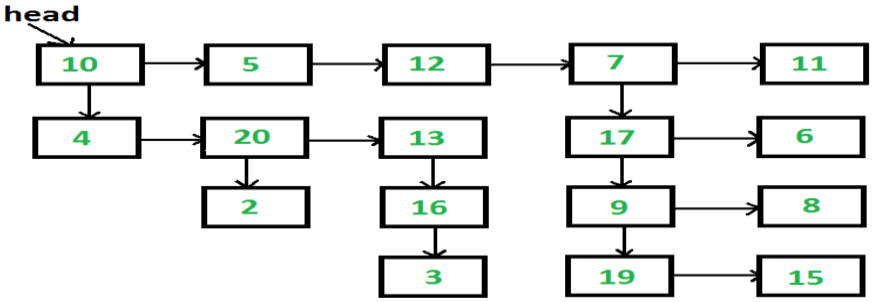
37. Flatten a multilevel linked list

Given a linked list where in addition to the next pointer, each node has a child pointer, which may or may not point to a separate list. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure, as shown in below figure.You are given the head of the first level of the list. Flatten the list so that all the nodes appear in a single-level linked list. You need to flatten the list in way that all nodes at first level should come first, then nodes of second level, and so on.

Each node is a C struct with the following definition.

```
struct list
```

```
{
    int data;
    struct list *next;
    struct list *child;
};
```



The above list should be converted to 10->5->12->7->11->4->20->13->17->6->2->16->9->8->3->19->15

The problem clearly say that we need to flatten level by level. The idea of solution is, we start from first level, process all nodes one by one, if a node has a child, then we append the child at the end of list, otherwise we don't do anything. After the first level is processed, all next level nodes will be appended after first level. Same process is followed for the appended nodes.

```
1) Take "cur" pointer, which will point to head of the fist level of the
list
2) Take "tail" pointer, which will point to end of the first level of
the list
3) Repeat the below procedure while "curr" is not NULL.
    I) if current node has a child then
       a) append this new child list to the "tail"
              tail->next = cur->child
       b) find the last node of new child list and update "tail"
              tmp = cur->child;
              while (tmp->next != NULL)
                     tmp = tmp->next;
              tail = tmp;
    II) move to the next node. i.e. cur = cur->next
```

Following is C implementation of the above algorithm.

```
// Program to flatten list with next and child pointers
#include <stdio.h>
#include <stdlib.h>

// Macro to find number of elements in array
#define SIZE(arr) (sizeof(arr)/sizeof(arr[0]))

// A linked list node has data, next pointer and child pointer
```

```
struct node
{
    int data;
    struct node *next;
    struct node *child;
};

// A utility function to create a linked list with n nodes. The data
// of nodes is taken from arr[].  All child pointers are set as NULL
struct node *createList(int *arr, int n)
{
    struct node *head = NULL;
    struct node *p;

    int i;
    for (i = 0; i < n; ++i) {
        if (head == NULL)
            head = p = (struct node *)malloc(sizeof(*p));
        else {
            p->next = (struct node *)malloc(sizeof(*p));
            p = p->next;
        }
        p->data = arr[i];
        p->next = p->child = NULL;
    }
    return head;
}

// A utility function to print all nodes of a linked list
void printList(struct node *head)
{
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

// This function creates the input list.  The created list is same
// as shown in the above figure
struct node *createList(void)
{
    int arr1[] = {10, 5, 12, 7, 11};
    int arr2[] = {4, 20, 13};
    int arr3[] = {17, 6};
    int arr4[] = {9, 8};
    int arr5[] = {19, 15};
    int arr6[] = {2};
    int arr7[] = {16};
    int arr8[] = {3};

    /* create 8 linked lists */
    struct node *head1 = createList(arr1, SIZE(arr1));
    struct node *head2 = createList(arr2, SIZE(arr2));
    struct node *head3 = createList(arr3, SIZE(arr3));
    struct node *head4 = createList(arr4, SIZE(arr4));
```

```
    struct node *head5 = createList(arr5, SIZE(arr5));
    struct node *head6 = createList(arr6, SIZE(arr6));
    struct node *head7 = createList(arr7, SIZE(arr7));
    struct node *head8 = createList(arr8, SIZE(arr8));


    /* modify child pointers to create the list shown above */
    head1->child = head2;
    head1->next->next->next->child = head3;
    head3->child = head4;
    head4->child = head5;
    head2->next->child = head6;
    head2->next->next->child = head7;
    head7->child = head8;


    /* Return head pointer of first linked list.  Note that all nodes
are
       reachable from head1 */
    return head1;
}

/* The main function that flattens a multilevel linked list */
void flattenList(struct node *head)
{
    /*Base case*/
    if (head == NULL)
       return;

    struct node *tmp;

    /* Find tail node of first level linked list */
    struct node *tail = head;
    while (tail->next != NULL)
        tail = tail->next;

    // One by one traverse through all nodes of first level
    // linked list till we reach the tail node
    struct node *cur = head;
    while (cur != tail)
    {
        // If current node has a child
        if (cur->child)
        {
            // then append the child at the end of current list
            tail->next = cur->child;

            // and update the tail to new last node
            tmp = cur->child;
            while (tmp->next)
                tmp = tmp->next;
            tail = tmp;
        }

        // Change current node
        cur = cur->next;
```

```
    }
}

// A driver program to test above functions
int main(void)
{
    struct node *head = NULL;
    head = createList();
    flattenList(head);
    printList(head);
    return 0;
}
```

Output:

```
10 5 12 7 11 4 20 13 17 6 2 16 9 8 3 19 15
```

Time Complexity: Since every node is visited at most twice, the time complexity is O(n) where n is the number of nodes in given linked list.

38. Delete N nodes after M nodes of a linked list

Given a linked list and two integers M and N. Traverse the linked list such that you retain M nodes then delete next N nodes, continue the same till end of the linked list.

Examples:

```
Input:
M = 2, N = 2
Linked List: 1->2->3->4->5->6->7->8
Output:
Linked List: 1->2->5->6

Input:
M = 3, N = 2
Linked List: 1->2->3->4->5->6->7->8->9->10
Output:
Linked List: 1->2->3->6->7->8

Input:
M = 1, N = 1
Linked List: 1->2->3->4->5->6->7->8->9->10
Output:
Linked List: 1->3->5->7->9
```

The main part of the problem is to maintain proper links between nodes, make sure that all corner cases are handled. Following is C implementation of function skipMdeleteN() that skips M nodes and delete N nodes till end of list. It is assumed that M cannot be 0.

```
// C program to delete N nodes after M nodes of a linked list
#include <stdio.h>
```

```
#include <stdlib.h>

// A linked list node
struct node
{
    int data;
    struct node *next;
};

/* Function to insert a node at the beginning */
void push(struct node ** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)  = new_node;
}

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Function to skip M nodes and then delete N nodes of the linked list.
void skipMdeleteN(struct node  *head, int M, int N)
{
    struct node *curr = head, *t;
    int count;

    // The main loop that traverses through the whole list
    while (curr)
    {
        // Skip M nodes
        for (count = 1; count<M && curr!= NULL; count++)
            curr = curr->next;

        // If we reached end of list, then return
        if (curr == NULL)
            return;

        // Start from next node and delete N nodes
        t = curr->next;
```

```
        for (count = 1; count<=N && t!= NULL; count++)
        {
            struct node *temp = t;
            t = t->next;
            free(temp);
        }
        curr->next = t; // Link the previous list with remaining nodes

        // Set current pointer for next iteration
        curr = t;
    }
}

// Driver program to test above functions
int main()
{
    /* Create following linked list
      1->2->3->4->5->6->7->8->9->10 */
    struct node* head = NULL;
    int M=2, N=3;
    push(&head, 10);
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);

    printf("M = %d, N = %d \nGiven Linked list is :\n", M, N);
    printList(head);

    skipMdeleteN(head, M, N);

    printf("\nLinked list after deletion is :\n");
    printList(head);

    return 0;
}
```

Output:

```
M = 2, N = 3
Given Linked list is :
1 2 3 4 5 6 7 8 9 10

Linked list after deletion is :
1 2 6 7
```

Time Complexity: O(n) where n is number of nodes in linked list.

39. QuickSort on Singly Linked List

QuickSort on Doubly Linked List is discussed here. QuickSort on Singly linked list was given as an exercise. Following is C++ implementation for same. The important things about implementation are, it changes pointers rather swapping data and time complexity is same as the implementation for Doubly Linked List.
In partition(), we consider last element as pivot. We traverse through the current list and if a node has value greater than pivot, we move it after tail. If the node has smaller value, we keep it at its current position.
In QuickSortRecur(), we first call partition() which places pivot at correct position and returns pivot. After pivot is placed at correct position, we find tail node of left side (list before pivot) and recur for left list. Finally, we recur for right list.

```
// C++ program for Quick Sort on Singly Linled List
#include <iostream>
#include <cstdio>
using namespace std;

/* a node of the singly linked list */
struct node
{
    int data;
    struct node *next;
};

/* A utility function to insert a node at the beginning of linked list
*/
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = new node;

    /* put in the data  */
    new_node->data  = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* A utility function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
    printf("\n");
```

```
}

// Returns the last node of the list
struct node *getTail(struct node *cur)
{
    while (cur != NULL && cur->next != NULL)
        cur = cur->next;
    return cur;
}

// Partitions the list taking the last element as the pivot
struct node *partition(struct node *head, struct node *end,
                       struct node **newHead, struct node **newEnd)
{
    struct node *pivot = end;
    struct node *prev = NULL, *cur = head, *tail = pivot;

    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot)
    {
        if (cur->data < pivot->data)
        {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        }
        else // If cur node is greater than pivot
        {
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }

    // If the pivot data is the smallest element in the current list,
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;

    // Update newEnd to the current last node
    (*newEnd) = tail;

    // Return the pivot node
    return pivot;
}
```

```
//here the sorting happens exclusive of the end node
struct node *quickSortRecur(struct node *head, struct node *end)
{
    // base condition
    if (!head || head == end)
        return head;

    node *newHead = NULL, *newEnd = NULL;

    // Partition the list, newHead and newEnd will be updated
    // by the partition function
    struct node *pivot = partition(head, end, &newHead, &newEnd);

    // If pivot is the smallest element - no need to recur for
    // the left part.
    if (newHead != pivot)
    {
        // Set the node before the pivot node as NULL
        struct node *tmp = newHead;
        while (tmp->next != pivot)
            tmp = tmp->next;
        tmp->next = NULL;

        // Recur for the list before pivot
        newHead = quickSortRecur(newHead, tmp);

        // Change next of last node of the left half to pivot
        tmp = getTail(newHead);
        tmp->next =  pivot;
    }

    // Recur for the list after the pivot element
    pivot->next = quickSortRecur(pivot->next, newEnd);

    return newHead;
}

// The main function for quick sort. This is a wrapper over recursive
// function quickSortRecur()
void quickSort(struct node **headRef)
{
    (*headRef) = quickSortRecur(*headRef, getTail(*headRef));
    return;
}

// Driver program to test above functions
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);
```

```
    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(&a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}
```

Output:

```
Linked List before sorting
30  3  4  20  5
Linked List after sorting
3  4  5  20  30
```

40. Merge a linked list into another linked list at alternate positions

Given two linked lists, insert nodes of second list into first list at alternate positions of first list.

For example, if first list is 5->7->17->13->11 and second is 12->10->2->4->6, the first list should become 5->12->7->10->17->2->13->4->11->6 and second list should become empty. The nodes of second list should only be inserted when there are positions available. For example, if the first list is 1->2->3 and second list is 4->5->6->7->8, then first list should become 1->4->2->5->3->6 and second list to 7->8.

Use of extra space is not allowed (Not allowed to create additional nodes), i.e., insertion must be done in-place. Expected time complexity is O(n) where n is number of nodes in first list.

The idea is to run a loop while there are available positions in first loop and insert nodes of second list by changing pointers. Following is C implementation of this approach.

```
// C implementation of above program.
#include <stdio.h>
#include <stdlib.h>

// A nexted list node
struct node
{
    int data;
    struct node *next;
};

/* Function to insert a node at the beginning */
void push(struct node ** head_ref, int new_data)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->data  = new_data;
```

```
    new_node->next = (*head_ref);
    (*head_ref)  = new_node;
}

/* Utility function to print a singly linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while (temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function that inserts nodes of linked list q into p at alternate
// positions. Since head of first list never changes and head of second
list
// may change, we need single pointer for first list and double pointer
for
// second list.
void merge(struct node *p, struct node **q)
{
     struct node *p_curr = p, *q_curr = *q;
     struct node *p_next, *q_next;

     // While therre are avialable positions in p
     while (p_curr != NULL && q_curr != NULL)
     {
         // Save next pointers
         p_next = p_curr->next;
         q_next = q_curr->next;

         // Make q_curr as next of p_curr
         q_curr->next = p_next;  // Change next pointer of q_curr
         p_curr->next = q_curr;  // Change next pointer of p_curr

         // Update current pointers for next iteration
         p_curr = p_next;
         q_curr = q_next;
    }

    *q = q_curr; // Update head pointer of second list
}

// Driver program to test above functions
int main()
{
     struct node *p = NULL, *q = NULL;
     push(&p, 3);
     push(&p, 2);
     push(&p, 1);
     printf("First Linked List:\n");
     printList(p);
```

```
    push(&q, 8);
    push(&q, 7);
    push(&q, 6);
    push(&q, 5);
    push(&q, 4);
    printf("Second Linked List:\n");
    printList(q);

    merge(p, &q);

    printf("Modified First Linked List:\n");
    printList(p);

    printf("Modified Second Linked List:\n");
    printList(q);

    getchar();
    return 0;
}
```

Output:

```
First Linked List:
1 2 3
Second Linked List:
4 5 6 7 8
Modified First Linked List:
1 4 2 5 3 6
Modified Second Linked List:
7 8
```

41. Pairwise swap elements of a given linked list by changing links

Given a singly linked list, write a function to swap elements pairwise. For example, if the linked list is 1->2->3->4->5->6->7 then the function should change it to 2->1->4->3->6->5->7, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5

This problem has been discussed here. The solution provided there swaps data of nodes. If data contains many fields, there will be many swap operations. So changing links is a better idea in general. Following is a C implementation that changes links instead of swapping data.

```
/* This program swaps the nodes of linked list rather than swapping the
field from the nodes.
Imagine a case where a node contains many fields, there will be plenty
of unnecessary swap calls. */

#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
```

```
/* A linked list node */
struct node
{
    int data;
    struct node *next;
};

/* Function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node **head)
{
    // If linked list is empty or there is only one node in list
    if (*head == NULL || (*head)->next == NULL)
        return;

    // Initialize previous and current pointers
    struct node *prev = *head;
    struct node *curr = (*head)->next;

    *head = curr;  // Change head before proceeeding

    // Traverse the list
    while (true)
    {
        struct node *next = curr->next;
        curr->next = prev; // Change next of current as previous node

        // If next NULL or next is the last node
        if (next == NULL || next->next == NULL)
        {
            prev->next = next;
            break;
        }

        // Change next of previous to next next
        prev->next = next->next;

        // Update previous and curr
        prev = next;
        curr = prev->next;
    }
}

/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

    /* put in the data  */
    new_node->data  = new_data;
```

```
    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Druver program to test above function */
int main()
{
    struct node *start = NULL;

    /* The constructed linked list is:
     1->2->3->4->5->6->7 */
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    printf("\n Linked list before calling  pairWiseSwap() ");
    printList(start);

    pairWiseSwap(&start);

    printf("\n Linked list after calling  pairWiseSwap() ");
    printList(start);

    getchar();
    return 0;
}
```

Output:

```
 Linked list before calling  pairWiseSwap() 1 2 3 4 5 6 7
 Linked list after calling  pairWiseSwap() 2 1 4 3 6 5 7
```

42. Given a linked list of line segments, remove middle points

Given a linked list of co-ordinates where adjacent points either form a vertical line or a horizontal line. Delete points from the linked list which are in the middle of a horizontal or vertical line.

Examples:

```
Input:  (0,10)->(1,10)->(5,10)->(7,10)
                                  |
                                (7,5)->(20,5)->(40,5)
Output: Linked List should be changed to following
        (0,10)->(7,10)
                  |
                (7,5)->(40,5)
The given linked list represents a horizontal line from (0,10)
to (7, 10) followed by a vertical line from (7, 10) to (7, 5),
followed by a horizontal line from (7, 5) to (40, 5).

Input:     (2,3)->(4,3)->(6,3)->(10,3)->(12,3)
Output: Linked List should be changed to following
    (2,3)->(12,3)
There is only one vertical line, so all middle points are removed.
```

The idea is to keep track of current node, next node and next-next node. While the next node is same as next-next node, keep deleting the next node. In this complete procedure we need to keep an eye on shifting of pointers and checking for NULL values.

Following is C implementation of above idea.

```
// C program to remove intermediate points in a linked list that
represents
// horizontal and vertical line segments
#include <stdio.h>
#include <stdlib.h>

// Node has 3 fields including x, y coordinates and a pointer to next
node
struct node
{
    int x, y;
    struct node *next;
};

/* Function to insert a node at the beginning */
void push(struct node ** head_ref, int x,int y)
{
    struct node* new_node = (struct node*) malloc(sizeof(struct node));
    new_node->x  = x;
    new_node->y  = y;
    new_node->next = (*head_ref);
    (*head_ref)  = new_node;
```

```
}

/* Utility function to print a singly linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while (temp != NULL)
    {
        printf("(%d,%d)-> ", temp->x,temp->y);
        temp = temp->next;
    }
    printf("\n");

}

// Utility function to remove Next from linked list and link nodes
// after it to head
void deleteNode(struct node *head, struct node *Next)
{
    head->next = Next->next;
    Next->next = NULL;
    free(Next);
}

// This function deletes middle nodes in a sequence of horizontal and
// vertical line segments represented by linked list.
struct node* deleteMiddle(struct node *head)
{
    // If only one node or no node...Return back
    if (head==NULL || head->next ==NULL || head->next->next==NULL)
        return head;

    struct node* Next = head->next;
    struct node *NextNext = Next->next ;

    // Check if this is a vertical line or horizontal line
    if (head->x == Next->x)
    {
        // Find middle nodes with same x value, and delete them
        while (NextNext !=NULL && Next->x==NextNext->x)
        {
            deleteNode(head, Next);

            // Update Next and NextNext for next iteration
            Next = NextNext;
            NextNext = NextNext->next;
        }
    }
    else if (head->y==Next->y) // If horizontal line
    {
        // Find middle nodes with same y value, and delete them
        while (NextNext !=NULL && Next->y==NextNext->y)
        {
            deleteNode(head, Next);

            // Update Next and NextNext for next iteration
```

```
            Next = NextNext;
            NextNext = NextNext->next;
        }
    }
    else  // Adjacent points must have either same x or same y
    {
        puts("Given linked list is not valid");
        return NULL;
    }

    // Recur for next segment
    deleteMiddle(head->next);

    return head;
}

// Driver program to tsst above functions
int main()
{
    struct node *head = NULL;

    push(&head, 40,5);
    push(&head, 20,5);
    push(&head, 10,5);
    push(&head, 10,8);
    push(&head, 10,10);
    push(&head, 3,10);
    push(&head, 1,10);
    push(&head, 0,10);
    printf("Given Linked List: \n");
    printList(head);

    if (deleteMiddle(head) != NULL);
    {
        printf("Modified Linked List: \n");
        printList(head);
    }
    return 0;
}
```

Output:

```
Given Linked List:
(0,10)-> (1,10)-> (3,10)-> (10,10)-> (10,8)-> (10,5)-> (20,5)-> (40,5)-
>
Modified Linked List:
(0,10)-> (10,10)-> (10,5)-> (40,5)->
```

Time Complexity of the above solution is O(n) where n is number of nodes in given linked list.

Exercise:

The above code is recursive, write an iterative code for the same problem.

43. Construct a Maximum Sum Linked List out of two Sorted Linked Lists having some Common nodes

Given two sorted linked lists, construct a linked list that contains maximum sum path from start to end. The result list may contain nodes from both input lists. When constructing the result list, we may switch to the other input list only at the point of intersection (which mean the two node with the same value in the lists). You are allowed to use O(1) extra space.

```
Input:
List1 =  1->3->30->90->120->240->511
List2 =  0->3->12->32->90->125->240->249

Output: Following is maximum sum linked list out of two input lists
list =  1->3->12->32->90->125->240->511
we switch at 3 and 240 to get above maximum sum linked list
```

The idea here in the below solution is to adjust next pointers after common nodes.

1. Start with head of both linked lists and find first common node. Use merging technique of sorted linked list for that.
2. Keep track of sum of the elements too while doing this and set head of result list based on greater sum till first common node.
3. After this till the current pointers of both lists don't become NULL we need to adjust the next of prev pointers based on greater sum.

This way it can be done in-place with constant extra space. Time complexity of the below solution is O(n).

```
// C++ program to construct the maximum sum linked
// list out of two given sorted lists
#include<iostream>
using namespace std;

//A linked list node
struct Node
{
    int data; //data belong to that node
    Node *next; //next pointer
};

// Push the data to the head of the linked list
void push(Node **head, int data)
{
    //Alocation memory to the new node
    Node *newnode = new Node;

    //Assigning data to the new node
```

```
    newnode->data = data;

    //Adjusting next pointer of the new node
    newnode->next = *head;

    //New node becomes the head of the list
    *head = newnode;
}

// Method that adjusts the pointers and prints the final list
void finalMaxSumList(Node *a, Node *b)
{
    Node *result = NULL;

    // Assigning pre and cur to the head of the
    // linked list.
    Node *pre1 = a, *curr1 = a;
    Node *pre2 = b, *curr2 = b;

    // Till either of the current pointers is not
    // NULL execute the loop
    while (curr1 != NULL || curr2 != NULL)
    {
        // Keeping 2 local variables at the start of every
        // loop run to keep track of the sum between pre
        // and cur pointer elements.
        int sum1 = 0, sum2 = 0;

        // Calculating sum by traversing the nodes of linked
        // list as the merging of two linked list.  The loop
        // stops at a common node
        while (curr1!=NULL && curr2!=NULL && curr1->data!=curr2->data)
        {
            if (curr1->data < curr2->data)
            {
                sum1 += curr1->data;
                curr1 = curr1->next;
            }
            else // (curr2->data < curr1->data)
            {
                sum2 += curr2->data;
                curr2 = curr2->next;
            }
        }

        // If either of current pointers becomes NULL
        // carry on the sum calculation for other one.
        if (curr1 == NULL)
        {
            while (curr2 != NULL)
            {
                sum2 += curr2->data;
                curr2 = curr2->next;
            }
        }
        if (curr2 == NULL)
```

```
        {
            while (curr1 != NULL)
            {
                sum1 += curr1->data;
                curr1 = curr1->next;
            }
        }

        // First time adjustment of resultant head based on
        // the maximum sum.
        if (pre1 == a && pre2 == b)
            result = (sum1 > sum2)? pre1 : pre2;

        // If pre1 and pre2 don't contain the head pointers of
        // lists adjust the next pointers of previous pointers.
        else
        {
            if (sum1 > sum2)
                pre2->next = pre1->next;
            else
                pre1->next = pre2->next;
        }

        // Adjusting previous pointers
        pre1 = curr1, pre2 = curr2;

        // If curr1 is not NULL move to the next.
        if (curr1)
            curr1 = curr1->next;
        // If curr2 is not NULL move to the next.
        if (curr2)
            curr2 = curr2->next;
    }

    // Print the resultant list.
    while (result != NULL)
    {
        cout << result->data << " ";
        result = result->next;
    }
}

//Main driver program
int main()
{
    //Linked List 1 : 1->3->30->90->110->120->NULL
    //Linked List 2 : 0->3->12->32->90->100->120->130->NULL
    Node *head1 = NULL, *head2 = NULL;
    push(&head1, 120);
    push(&head1, 110);
    push(&head1, 90);
    push(&head1, 30);
    push(&head1, 3);
    push(&head1, 1);

    push(&head2, 130);
```

```
    push(&head2, 120);
    push(&head2, 100);
    push(&head2, 90);
    push(&head2, 32);
    push(&head2, 12);
    push(&head2, 3);
    push(&head2, 0);

    finalMaxSumList(head1, head2);
    return 0;
}
```

Output:

```
1 3 12 32 90 110 120 130
```

Time complexity = O(n) where n is the length of bigger linked list
Auxiliary space = O(1)

However a problem in this solution is that the original lists are changed.

Exercise

1. Try this problem when auxiliary space is not a constraint.
2. Try this problem when we don’t modify the actual list and create the resultant list.