Articles

1. Little and Big Endian Mystery

What are these?
Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.

- Memory representation of integer ox01234567 inside Big and little endian machines

How to see memory representation of multibyte data types on your machine?
Here is a sample C code that shows the byte representation of int, float and pointer.

```
#include <stdio.h>

/* function to show bytes in memory, from location start to start+n*/
void show_mem_rep(char *start, int n)
{
    int i;
    for (i = 0; i < n; i++)
         printf(" %.2x", start[i]);
    printf("\n");
}

/*Main function to call above function for 0x01234567*/
int main()
{
   int i = 0x01234567;
   show_mem_rep((char *)&i, sizeof(i));
   getchar();
   return 0;
}
```

When above program is run on little endian machine, gives "67 45 23 01" as output , while if it is run on endian machine, gives "01 23 45 67" as output.

Is there a quick way to determine endianness of your machine?
There are n no. of ways for determining endianness of your machine. Here is one quick way of doing the same.

```c
#include <stdio.h>
int main()
{
   unsigned int i = 1;
   char *c = (char*)&i;
   if (*c)
       printf("Little endian");
   else
       printf("Big endian");
   getchar();
   return 0;
}
```

In the above program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is de-referenced it will contain only first byte of integer. If machine is little endian then *c will be 1 (because last byte is stored first) and if machine is big endian then *c will be 0.

Does endianness matter for programmers?
Most of the times compiler takes care of endianness, however, endianness becomes an issue in following cases.

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

Standard byte order for networks is big endian, also known as network byte order. Before transferring data on network, data is first converted to network byte order (big endian).

Sometimes it matters when you are using type casting, below program is an example.

```c
#include <stdio.h>
int main()
{
    unsigned char arr[2] = {0x01, 0x00};
    unsigned short int x = *(unsigned short int *) arr;
    printf("%d", x);
    getchar();
    return 0;
}
```

In the above program, a char array is typecasted to an unsigned short integer type. When I run above program on little endian machine, I get 1 as output, while if I run it on a big endian machine I get 256. To make programs endianness independent, above programming style should be avoided.

What are bi-endians?
Bi-endian processors can run in both modes little and big endian.

What are the examples of little, big endian and bi-endian machines ?

Intel based processors are little endians. ARM processors were little endians. Current generation ARM processors are bi-endian.

Motorola 68K processors are big endians. PowerPC (by Motorola) and SPARK (by Sun) processors were big endian. Current version of these processors are bi-endians.

Does endianness effects file formats?
File formats which have 1 byte as a basic unit are independent of endianness e..g., ASCII files . Other file formats use some fixed endianness forrmat e.g, JPEG files are stored in big endian format.

Which one is better — little endian or big endian
The term little and big endian came from Gulliver's Travels by Jonathan Swift. Two groups could not agree by which end a egg should be opened -a-the little or the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, hence the arguments degenerate into bickering about sociopolitical issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

2. Understanding "extern" keyword in C

I'm sure that this post will be as interesting and informative to C virgins (i.e. beginners) as it will be to those who are well versed in C. So let me start with saying that extern keyword applies to C variables (data objects) and C functions. Basically extern keyword extends the visibility of the C variables and C functions. Probably that's is the reason why it was named as extern.

Though (almost) everyone knows the meaning of declaration and definition of a variable/function yet for the sake of completeness of this post, I would like to clarify them. 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 are the arguments to that functions, 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). So that’s all about declaration and definition.

Now coming back to our main objective: Understanding “extern” keyword in C. I’ve explained the role of declaration/definition because it’s mandatory to understand them to understand the “extern” keyword. Let us first take the easy case. 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.

Now let us the take the second and final case i.e. use of extern with C variables. I feel that it 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 it’s 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) So far so good. 🙂

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.

Now let us try to understand extern with examples.

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:

```c
#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:

```c
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.

So that was a preliminary look at “extern” keyword in C.

I’m sure that you want to have some take away from the reading of this post. And I would not disappoint you. 😃
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.

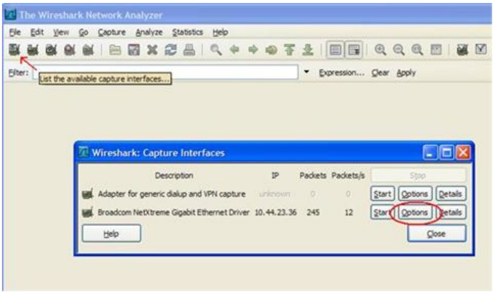
3. Let’s experiment with Networking

Most of us have studied Computer Networks in a very abstract manner. In other words, not many of us know how the abstract concepts of layers and packets translate in real life networks such as the Internet. Therefore let us do an experiment and see whether these layers, packets etc. exist in any real network also. So get, set and ready to delve into this wonderful world of practical and experimental Networking!

The outline of our experiment is as follows. We will capture some live packets, and to understand what is inside those packets, we will analyze those packets by dissecting them. Sounds surgical? Yup, it is. J

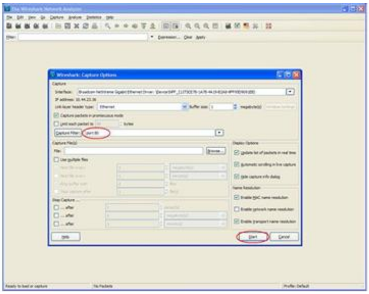
To start with, we need to have a PC running Windows XP and connected to the Internet. If you are reading this article online, the chances are high that you have everything ready to experiment. Now let's recall some of the theory stuff that we read in Networking Books. The first thing that almost every book tells us – networking architecture is layered; remember that 7 layer OSI protocol stack! So where are these protocol layers? In our experiment, we will use 5 layer Internet Protocol stack so that we can solve the mystery of these layers.

We start our experiment by installing Wireshark (earlier known as Ethereal). Wireshark is a Network Protocol Analyzer that can capture and analyze the packets transmitted/received via a Network Interface Card (NIC). [You need to bear with me this acronym because Networking is full of acronymsJ] We install Wireshark from http://www.wireshark.org/download.html (at the time of this writing, the latest Wireshark version is 1.0.3). While installing Wireshark, leave the default settings/options as it is. Now our experimental setup is ready. Run Wireshark and click on the first icon (List the available capture interfaces …). Now we see a pop up window that shows Capture Interfaces. See the snapshots as follows.

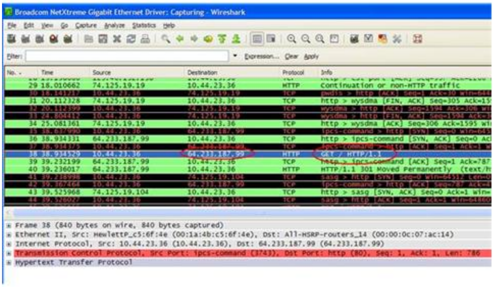

The Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Help
Filter:
List the available capture interfaces...
Expression... Clear Apply
Wireshark: Capture Interfaces
Stop
Help
Close

Description	IP	Packets	Packets/s			
Adapter for generic dialup and VPN capture	unknown	0	0	Start	Options	Details
Broadcom NetXtreme Gigabit Ethernet Driver	10.44.23.36	245	12	Start	Options	Details

The number and types of interfaces shown in Capture Interfaces window can be different for you depending on your PC's configuration. For me it shows two interfaces and my Internet connection is through Broadcom Gigabit Interface. So choose the interface through which your Internet connection is available to you. Now let's click on the Options button of this interface. Now we see a new window named Capture Options. In this window, type "port 80" in text field named Capture Filter. See the following snapshot for clarification.



Now we are ready to capture the packets passing through our NIC. By setting the filter to “port 80”, we have instructed Wireshark to capture only those packets that are because of http traffic (remember that we were always told that the default http port is 80!). Now click on the Start button on Capture Options window. You may see some packets in Wireshark if any program in your PC is accessing http traffic in the background; let’s not focus on that. Now open your browser and try to access http://google.com and now you should be seeing lot many packets getting captured in Wireshark. See the snapshot as follows.


Broadcom NetXtreme Gigabit Ethernet Driver: Capturing - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression... Clear Apply

```
No. -  Time        Source          Destination     Protocol  Info
    29 18.010662   74.125.19.19    10.44.23.36     HTTP      Continuation or non-HTTP traffic
    30 18.141217   10.44.23.36     74.125.19.19    TCP       pwdis > http [ACK] Seq=1 Ack=30 Win=644
    31 20.112328   74.125.19.19    10.44.23.36     TCP       http > wysdma [FIN, ACK] Seq=305 Ack=15
    32 20.112399   10.44.23.36     74.125.19.19    TCP       wysdma > http [ACK] Seq=1594 Ack=306 Wi
    33 24.804412   10.44.23.36     74.125.19.19    TCP       wysdma > http [FIN, ACK] Seq=1594 Ack=3
    34 25.081361   74.125.19.19    10.44.23.36     TCP       http > wysdma [ACK] Seq=306 Ack=1595 Wi
    35 38.637990   10.44.23.36     64.233.187.99   TCP       ipcs-command > http [SYN] Seq=0 Win=645
    36 38.934331   64.233.187.99   10.44.23.36     TCP       http > ipcs-command [SYN, ACK] Seq=0 Ac
    37 38.934375   10.44.23.36     64.233.187.99   TCP       ipcs-command > http [ACK] Seq=1 Ack=1 W
    38 38.934529   10.44.23.36     64.233.187.99   HTTP      GET / HTTP/1.1
    39 39.232199   64.233.187.99   10.44.23.36     TCP       http > ipcs-command [ACK] Seq=1 Ack=787
    40 39.236017   64.233.187.99   10.44.23.36     HTTP      HTTP/1.1 301 Moved Permanently  (text/h
    41 39.238998   10.44.23.36     74.125.19.104   TCP       sasg > http [SYN] Seq=0 Win=64512 Len=0
    42 39.367468   10.44.23.36     64.233.187.99   TCP       ipcs-command > http [ACK] Seq=787 Ack=4
    43 39.525968   74.125.19.104   10.44.23.36     TCP       http > sasg [SYN, ACK] Seq=0 Ack=1 Win=
    44 39.526027   10.44.23.36     74.125.19.104   TCP       sasg > http [ACK] Seq=1 Ack=1 Win=64860

⊞ Frame 38 (840 bytes on wire, 840 bytes captured)
⊞ Ethernet II, Src: HewlettP_c5:6f:4e (00:1a:4b:c5:6f:4e), Dst: All-HSRP-routers_14 (00:00:0c:07:ac:14)
⊞ Internet Protocol, Src: 10.44.23.36 (10.44.23.36), Dst: 64.233.187.99 (64.233.187.99)
⊞ Transmission Control Protocol, Src Port: ipcs-command (3743), Dst Port: http (80), Seq: 1, Ack: 1, Len: 786
⊞ Hypertext Transfer Protocol
```

Let’s start analyzing the captured packets. First of all, find the first instance of http packet that has GET / HTTP/1.1 in its Info field. In the above snapshot, it’s shown in blue. If we take a closer look, we see that this packet has the headers of the all the 5 layers of the Internet Protocol stack.

Layer 1 – It is the Physical layer. Here Frames are shown at the physical layer.

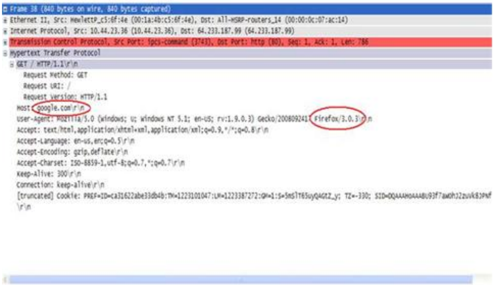
Layer 2 – It is the Data Link layer. In this packet, we can see that Ethernet II is used as data link layer protocol. We can find the MAC address of the source and destination in this header.

Layer 3 – It is the Network layer. In this packet, we see that IP is used as Network layer protocol. We can see the source and destination IP in this header.

Layer 4 – It is the Transport layer. In this packet, TCP is used as Transport layer protocol. We can find the source and destination ports in this header.

Layer 5 – It is the Application layer. In this packet, HTTP is used as Application layer protocol.

Let’s explore one of the layers. Other layers can be explored further in the similar fashion. If we expand the Layer 5 i.e. HTTP header, it looks as follows.



```
⊟ Frame 38 (840 bytes on wire, 840 bytes captured)
⊞ Ethernet II, Src: HewlettP_c5:6f:4e (00:1a:4b:c5:6f:4e), Dst: All-HSRP-routers_14 (00:00:0c:07:ac:14)
⊞ Internet Protocol, Src: 10.44.23.36 (10.44.23.36), Dst: 64.233.187.99 (64.233.187.99)
⊞ Transmission Control Protocol, Src Port: fpcs-command (3743), Dst Port: http (80), Seq: 1, Ack: 1, Len: 786
⊟ Hypertext Transfer Protocol
  ⊟ GET / HTTP/1.1\r\n
      Request Method: GET
      Request URI: /
      Request Version: HTTP/1.1
    Host: google.com\r\n
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.3) Gecko/2008092417 Firefox/3.0.3\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: en-us,en;q=0.5\r\n
    Accept-Encoding: gzip,deflate\r\n
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
    Keep-Alive: 300\r\n
    Connection: keep-alive\r\n
    [truncated] Cookie: PREF=ID=ca31622abe33db4b:TM=1223101047:LM=1223387272:GM=1:S=5mSlT65uyQAGtZ_y; TZ=-330; SID=DQAAAHoAAABU93f7awOhJ2zuVk8JPNf
    \r\n
```

Here we see that Host is mentioned as google.com that is what we tried to access from browser. User Agent field of the HTTP header shows the browser details. In my case, it is Mozilla Firefox as evidenced from this header. Destination IP 64.233.187.99 should be one of the IP addresses assigned to Google server where the web server is hosted. It can be verified using a very handy utility command “nslookup”. The details of the other fields can be explored in the headers of the HTTP, TCP, IP, Ethernet II protocols. Some of the interesting fields are – Differentiated Services Field (also known as QoS field) in IP header, Window size in TCP header etc.

So we have seen that all those rhetorical concepts of layers etc. do exist in real networks also. And it sounds interesting when you dissect all the packets that pass through your interface card. By doing so, you can get to know what goes/comes through your PC!

The idea of this experiment is to provide a conducive platform so that you can explore your own the exciting world of Networking. So welcome aboard!

4. What is Memory Leak? How can we avoid?

Memory leak occurs when programmers create a memory in heap and forget to delete it.
Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

```c
/* Function with memory leak */
#include <stdlib.h>

void f()
{
   int *ptr = (int *) malloc(sizeof(int));

   /* Do some work */

   return; /* Return without freeing ptr*/
}
```

To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

```c
/* Function without memory leak */
#include <stdlib.h>;

void f()
{
   int *ptr = (int *) malloc(sizeof(int));

   /* Do some work */

   free(ptr);
   return;
}
```

5. What are Wild Pointers? How can we avoid?

Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location and may cause a program to crash or behave badly.

```c
int main()
{
  int *p;  /* wild pointer */
  *p = 12; /* Some unknown memory location is being corrupted. This sh
}
```

Please note that if a pointer p points to a known variable then it's not a wild pointer. In the below program, p is a wild pointer till this points to a.

```c
int main()
{
  int  *p; /* wild pointer */
  int a = 10;
  p = &a;  /* p is not a wild pointer now*/
  *p = 12; /* This is fine. Value of a is changed */
}
```

If we want pointer to a value (or set of values) without having a variable for the value, we should explicitly allocate memory and put the value in allocated memory.

```
int main()
{
   int *p = malloc(sizeof(int));
  *p = 12; /* This is fine (assuming malloc doesn't return NULL) */
}
```

6. Understanding “register” keyword in C

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. It’s 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.

Please write comments if you find anything incorrect in the above article or you want to share more information about register keyword.

7. Storage for Strings in C

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
Using character pointer strings can be stored in two ways:

1) Read only string in a shared segment.
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.

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 varial
 *(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 alse 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 segm
  *(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;
}
```

Please write comments if you find anything incorrect in the above article, or you want to share more information about storage of strings

8. C function to Swap strings

Let us consider the below program.

```
#include<stdio.h>
void swap(char *str1, char *str2)
{
  char *temp = str1;
  str1 = str2;
  str2 = temp;
}

int main()
{
  char *str1 = "geeks";
  char *str2 = "forgeeks";
  swap(str1, str2);
  printf("str1 is %s, str2 is %s", str1, str2);
  getchar();
  return 0;
}
```

Output of the program is *str1 is geeks, str2 is forgeeks*. So the above swap() function doesn’t swap strings. The function just changes local pointer variables and the changes are not reflected outside the function.

Let us see the correct ways for swapping strings:

Method 1(Swap Pointers)
If you are using character pointer for strings (not arrays) then change str1 and str2 to point each other’s data. i.e., swap pointers. In a function, if we want to change a pointer (and obviously we want changes to be reflected outside the function) then we need to pass a pointer to the pointer.

```c
#include<stdio.h>

/* Swaps strings by swapping pointers */
void swap1(char **str1_ptr, char **str2_ptr)
{
  char *temp = *str1_ptr;
  *str1_ptr = *str2_ptr;
  *str2_ptr = temp;
}

int main()
{
  char *str1 = "geeks";
  char *str2 = "forgeeks";
  swap1(&str1, &str2);
  printf("str1 is %s, str2 is %s", str1, str2);
  getchar();
  return 0;
}
```

This method cannot be applied if strings are stored using character arrays.

Method 2(Swap Data)

If you are using character arrays to store strings then preferred way is to swap the data of both arrays.

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

/* Swaps strings by swapping data*/
void swap2(char *str1, char *str2)
{
  char *temp = (char *)malloc((strlen(str1) + 1) * sizeof(char));
  strcpy(temp, str1);
  strcpy(str1, str2);
  strcpy(str2, temp);
  free(temp);
}

int main()
{
  char str1[10] = "geeks";
  char str2[10] = "forgeeks";
  swap2(str1, str2);
  printf("str1 is %s, str2 is %s", str1, str2);
  getchar();
  return 0;
}
```

This method cannot be applied for strings stored in read only block of memory.

Please write comments if you find anything incorrect in the above article, or you want to share more information about the topic discussed above.

9. gets() is risky to use!

Asked by geek4u

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;
}
```

Please write comments if you find anything incorrect in the above article, or you want to share more information about the topic discussed above.

10. Do not use sizeof for array parameters

Consider the below program.

```
#include<stdio.h>
void fun(int arr[])
{
  int i;

  /* sizeof should not be used here to get number
    of elements in array*/
  int arr_size = sizeof(arr)/sizeof(arr[0]); /* incorrect use of siseo
  for (i = 0; i < arr_size; i++)
  {
    arr[i] = i;  /*executed only once */
  }
}

int main()
{
  int i;
  int arr[4] = {0, 0 ,0, 0};
  fun(arr);

  /* use of sizeof is fine here*/
  for(i = 0; i < sizeof(arr)/sizeof(arr[0]); i++)
    printf(" %d " ,arr[i]);

  getchar();
  return 0;
}
```

Output: 0 0 0 0 on a IA-32 machine.

The function fun() receives an array parameter arr[] and tries to find out number of elements in arr[] using sizeof operator.
In C, array parameters are treated as pointers (See http://geeksforgeeks.org/?p=4088 for details). So the expression sizeof(arr)/sizeof(arr[0]) becomes sizeof(int *)/sizeof(int) which results in 1 for IA 32 bit machine (size of int and int * is 4) and the for loop inside fun() is executed only once irrespective of the size of the array.

Therefore, sizeof should not be used to get number of elements in such cases. A separate parameter for array size (or length) should be passed to fun(). So the **corrected program is:**

```
#include<stdio.h>
void fun(int arr[], size_t arr_size)
{
  int i;
  for (i = 0; i < arr_size; i++)
  {
    arr[i] = i;
  }
}

int main()
{
  int i;
  int arr[4] = {0, 0 ,0, 0};
  fun(arr, 4);

  for(i = 0; i < sizeof(arr)/sizeof(arr[0]); i++)
    printf(" %d ", arr[i]);

  getchar();
  return 0;
}
```

Please write comments if you find anything incorrect in the above article or you want to share more information about the topic discussed above.

11. Quine – A self-reproducing program

A quine is a program which prints a copy of its own as the only output. A quine takes no input. Quines are named after the American mathematician and logician Willard Van Orman Quine (1908–2000). The interesting thing is you are not allowed to use open and then print file of the program.

To the best of our knowledge, below is the shortest quine in C.

```
main() { char *s="main() { char *s=%c%s%c; printf(s,34,s,34); }"; prin
```

This program uses the printf function without including its corresponding header (#include), which can result in undefined behavior. Also, the return type declaration for main has been left off to reduce the length of the program. Two 34s are used to print double quotes around the string s.

Following is a shorter version of the above program suggested by Narendra.

```
main(a){printf(a="main(a){printf(a=%c%s%c,34,a,34);}",34,a,34);}
```

If you find a shorter C quine or you want to share quine in other programming languages, then please do write in the comment section.

Source:
http://en.wikipedia.org/wiki/Quine_%28computing%29

12. exit(), abort() and assert()

exit()

```
void exit ( int status );
```

exit() terminates the process normally.
status: Status value returned to the parent process. Generally, a status value of 0 or EXIT_SUCCESS indicates success, and any other value or the constant EXIT_FAILURE is used to indicate an error. exit() performs following operations.
* Flushes unwritten buffered data.
* Closes all open files.
* Removes temporary files.
* Returns an integer exit status to the operating system.

The C standard atexit() function can be used to customize exit() to perform additional actions at program termination.

Example use of exit.

```
/* exit example */
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  FILE * pFile;
  pFile = fopen ("myfile.txt", "r");
  if (pFile == NULL)
  {
    printf ("Error opening file");
    exit (1);
  }
  else
  {
    /* file operations here */
  }
  return 0;
}
```

abort()

```
void abort ( void );
```

Unlike exit() function, abort() may not close files that are open. It may also not delete temporary files and may not flush stream buffer. Also, it does not call functions registered with atexit().

This function actually terminates the process by raising a SIGABRT signal, and your program can include a handler to intercept this signal (see this).

So programs like below might not write “Geeks for Geeks” to “tempfile.txt”

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
  FILE *fp = fopen("C:\\myfile.txt", "w");

  if(fp == NULL)
  {
    printf("\n could not open file ");
    getchar();
    exit(1);
  }

  fprintf(fp, "%s", "Geeks for Geeks");

  /* ....... */
  /* ....... */
  /* Something went wrong so terminate here */
  abort();

  getchar();
  return 0;
}
```

If we want to make sure that data is written to files and/or buffers are flushed then we should either use exit() or include a signal handler for SIGABRT.

assert()

```
void assert( int expression );
```

If expression evaluates to 0 (false), then the expression, sourcecode filename, and line number are sent to the standard error, and then abort() function is called. If the identifier NDEBUG (“no debug”) is defined with #define NDEBUG then the macro assert does nothing.

Common error outputting is in the form:

Assertion failed: expression, file filename, line line-number

```
#include<assert.h>

void open_record(char *record_name)
{
    assert(record_name != NULL);
    /* Rest of code */
}

int main(void)
{
   open_record(NULL);
}
```

This article is contributed by **Rahul Gupta**. Please write comments if you find anything incorrect in the above article or you want to share more information about the topic discussed above.
References:
http://www.cplusplus.com/reference/clibrary/cstdlib/abort/
http://www.cplusplus.com/reference/clibrary/cassert/assert/
http://www.acm.uiuc.edu/webmonkeys/book/c_guide/2.1.html
https://www.securecoding.cert.org/confluence/display/seccode/ERR06-C.+Understand+the+termination+behavior+of+assert%28%29+and+abort%28%29
https://www.securecoding.cert.org/confluence/display/seccode/ERR04-C.+Choose+an+appropriate+termination+strategy

13. Recursive Functions

Recursion:

In programming terms a recursive function can be defined as a routine that calls itself directly or indirectly.

Using recursive algorithm, certain problems can be solved quite easily. Towers of Hanoi (TOH) is one such programming exercise. Try to write an *iterative* algorithm for TOH. Moreover, every recursive program can be written using iterative methods (Refer Data Structures by Lipschutz).

Mathematically recursion helps to solve few puzzles easily.

For example, a routine interview question,

In a party of N people, each person will shake her/his hand with each other person only once. On total how many hand-shakes would happen?

Solution:

It can be solved in different ways, graphs, recursion, etc. Let us see, how recursively it can be solved.

There are N persons. Each person shake-hand with each other only once. Considering N-th person, (s)he has to shake-hand with (N-1) persons. Now the problem reduced to small instance of (N-1) persons. Assuming T_N as total shake-hands, it can be formulated recursively.

$T_N = (N-1) + T_{N-1}$ [$T_1 = 0$, i.e. the last person have already shook-hand with every one]

Solving it recursively yields an arithmetic series, which can be evaluated to N(N-1)/2.

Exercise: In a party of N couples, only one gender (either male or female) can shake hand with every one. How many shake-hands would happen?

Usually recursive programs results in poor time complexities. An example is Fibonacci series. The time complexity of calculating n-th Fibonacci number using recursion is approximately 1.6^n. It means the same computer takes almost 60% more time for next Fibonacci number. Recursive Fibonacci algorithm has overlapped subproblems. There are other techniques like *dynamic programming* to improve such overlapped algorithms.

However, few algorithms, (e.g. merge sort, quick sort, etc…) results in optimal time complexity using recursion.

Base Case:

One critical requirement of recursive functions is termination point or base case. Every recursive program must have base case to make sure that the function will terminate. Missing base case results in unexpected behaviour.

Different Ways of Writing Recursive Functions in C/C++:

Function calling itself: (Direct way)

Most of us aware atleast two different ways of writing recursive programs. Given below is towers of Hanoi code. It is an example of direct calling.

```
// Assuming n-th disk is bottom disk (count down)
void tower(int n, char sourcePole, char destinationPole, char auxiliar
{
   // Base case (termination condition)
   if(0 == n)
     return;

   // Move first n-1 disks from source pole
   // to auxiliary pole using destination as
   // temporary pole
   tower(n-1, sourcePole, auxiliaryPole,
      destinationPole);

// Move the remaining disk from source
   // pole to destination pole
   printf("Move the disk %d from %c to %c\n", n,
     sourcePole, destinationPole);

   // Move the n-1 disks from auxiliary (now source)
   // pole to destination pole using source pole as
   // temporary (auxiliary) pole
   tower(n-1, auxiliaryPole, destinationPole,
     sourcePole);
}

void main()
{
   tower(3, 'S', 'D', 'A');
}
```

The time complexity of TOH can be calculated by formulating number of moves.

We need to move the first N-1 disks from Source to Auxiliary and from Auxiliary to Destination, i.e. the first N-1 disks requires two moves. One more move of last disk from Source to Destination. Mathematically it can be defined recursively.

$M_N = 2M_{N-1} + 1$.

We can easily solve the above recursive relation (2^N-1), which is exponential.

Recursion using mutual function call: (Indirect way)

Indirect calling. Though least pratical, a function [funA()] can call another function [funB()] which inturn calls [funA()] former function. In this case both the functions should have the base case.

Defensive Programming:

We can combine defensive coding techniques with recursion for graceful functionality of application. Usually recursive programming is not allowed in safety critical applications, such as flight controls, health monitoring, etc. However, one can use a static count technique to avoid uncontrolled calls (NOT in safety critical systems, may be used in soft real time systems).

```
void recursive(int data)
{
   static callDepth;
   if(callDepth > MAX_DEPTH)
      return;

   // Increase call depth
   callDepth++;

   // do other processing
   recursive(data);

   // do other processing
   // Decrease call depth
   callDepth--;
}
```

callDepth depth depends on function stack frame size and maximum stack size.

Recursion using function pointers: (Indirect way)

Recursion can also implemented with function pointers. An example is signal handler in POSIX complaint systems. If the handler causes to trigger same event due to which the handler being called, the function will reenter.

We will cover function pointer approach and iterative solution to TOH puzzle in later article.

Thanks to Venki for writing the above post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

14. The OFFSETOF() macro

We know that the elements in a structure will be stored in sequential order of their declaration.

How to extract the displacement of an element in a structure? We can make use of offsetof macro.

Usually we call structure and union types (or *classes with trivial constructors*) as *plain old data* (POD) types, which will be used to *aggregate other data types*. The following non-standard macro can be used to get the displacement of an element in bytes from the base address of the structure variable.

```
#define OFFSETOF(TYPE, ELEMENT) ((size_t)&(((TYPE *)0)->ELEMENT))
```

Zero is casted to type of structure and required element’s address is accessed, which is casted to *size_t*. As per standard *size_t* is of type *unsigned int*. The overall expression results in the number of bytes after which the ELEMENT being placed in the structure.

For example, the following code returns 16 bytes (padding is considered on 32 bit machine) as displacement of the character variable *c* in the structure Pod.

```
#include <stdio.h>

#define OFFSETOF(TYPE, ELEMENT) ((size_t)&(((TYPE *)0)->ELEMENT))

typedef struct PodTag
{
   int     i;
   double  d;
   char    c;
} PodType;

int main()
{
   printf("%d", OFFSETOF(PodType, c) );

   getchar();
   return 0;
}
```

In the above code, the following expression will return the displacement of element *c* in the structure *PodType*.

```
OFFSETOF(PodType, c);
```

After preprocessing stage the above macro expands to

```
((size_t)&(((PodType *)0)->c))
```

Since we are considering 0 as address of the structure variable, c will be placed after 16 bytes of its base address i.e. 0x00 + 0x10. Applying & on the structure element (in this case it is c) returns the address of the element which is 0x10. Casting the address to *unsigned int* (size_t) results in number of bytes the element is placed in the structure.

Note: We may consider the address operator & is redundant. Without address operator in macro, the code de-references the element of structure placed at NULL address. It causes an access violation exception (segmentation fault) at runtime.

Note that there are other ways to implement offsetof macro according to compiler behaviour. The ultimate goal is to extract displacement of the element. ***We will see practical usage of offsetof macro in liked lists to connect similar objects (for example thread pool) in another article.***

Article compiled by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

1. Linux Kernel code.

2. http://msdn.microsoft.com/en-us/library/dz4y9b9a.aspx

3. GNU C/C++ Compiler Documentation

15. Reentrant Function

What we mean by reentrancy and reentrant function? What is it's significance in programming? What are the conditions that a function to be reentrant?

Reentrancy is applicable in concurrent programming. Cooperative scheduling need not to consider reentrancy. Prior to discussing on reentrant functions, we need to understand few keywords. Read atomicity and critical section posts.

If one thread tries to change the value of shared data at the same time as another thread tries to read the value, the result is not predictable. We call it as race condition.

A reentrant function guarantees it's functionality even when the function is invoked (reentered) from concurrent multiple threads. If the function is using variables with static extent, the results will be unpredictable because of race conditions. A library developer need to care in writing reentrant code. Sometimes the bug lies in the library rather than programmer's code. It is not easy to recreate such bugs during fix.

A Reentrant Function shall satisfy the following conditions,

1. Should not call another non-reentrant function
2. Should not access static life time variables (static/extern)
3. Should not include self modifying code

Thread safety and Reentrant functions

Thread safety and reentrant functions are connected. Every thread safe function is reentrant, but the converse need not be true. A thread safe function can be called from multiple threads even when the function accessing shared data. A thread safe function is guaranteed to serialize accessing any shared data.

An example is string class (C++). Most implementations of string class are reentrant but not thread safe. One can create different instances of string class across multiple threads, but can't access same instance atomically from multiple threads. For more details of string class see QString.

Reentrancy is key while writing critical sections, signal handlers, interrupt service routines, etc…

Reentrant Function vs Functions with Reentrancy:

The above statement may look strange. There are different memory models of processors. Some architectures use a common set of memory locations to pass arguments to functions. In such case the functions can't maintain reentrancy when invoked multiple times. Code development for such processors must qualify the function prototype with compiler provided keywords to ensure reentrancy.

Article compiled by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

16. Structure Member Alignment, Padding and Data Packing

What do we mean by data alignment, structure packing and padding?

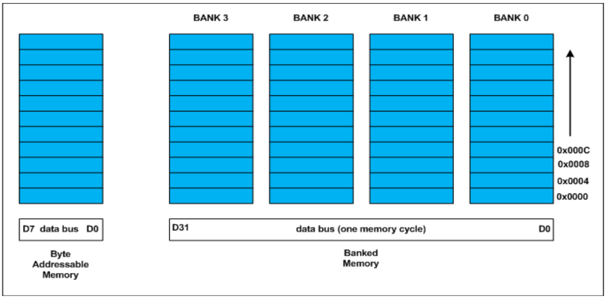
Predict the output of following program.

```
#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:

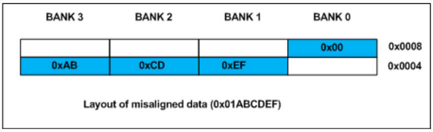
Every data type in C/C++ will have alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.



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.

```c
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.

– – – by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

17. Floating Point Representation – Basics

There are posts on representation of floating point format. The objective of this article is to provide a brief introduction to floating point format.

The following description explains terminology and primary details of IEEE 754 binary floating point representation. The discussion confines to single and double precision formats.

Usually, a real number in binary will be represented in the following format,

$$I_mI_{m-1}\ldots I_2I_1I_0.F_1F_2\ldots F_nF_{n-1}$$

Where I_m and F_n will be either 0 or 1 of integer and fraction parts respectively.

A finite number can also represented by four integers components, a sign (s), a base (b), a significand (m), and an exponent (e). Then the numerical value of the number is evaluated as

$$(-1)^s \text{ x m x } b^e \text{ ——— Where } m < |b|$$

Depending on base and the number of bits used to encode various components, the IEEE 754 standard defines five basic formats. Among the five formats, the binary32 and the binary64 formats are single precision and double precision formats respectively in which the base is 2.

Table – 1 Precision Representation

Precision	Base	Sign	Exponent	Significand
Single precision	2	1	8	23+1
Double precision	2	1	11	52+1

Single Precision Format:

As mentioned in Table 1 the single precision format has 23 bits for significand (1 represents implied bit, details below), 8 bits for exponent and 1 bit for sign.

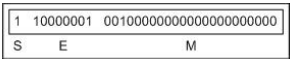
For example, the rational number 9÷2 can be converted to single precision float format as following,

$$9_{(10)} \div 2_{(10)} = 4.5_{(10)} = 100.1_{(2)}$$

The result said to be ***normalized***, if it is represented with leading 1 bit, i.e. $1.001_{(2)}$ x 2^2. (Similarly when the number $0.000000001101_{(2)}$ x 2^3 is normalized, it appears as $1.101_{(2)}$ x 2^{-6}). Omitting this implied 1 on left extreme gives us the ***mantissa*** of float number. A normalized number provides more accuracy than corresponding ***de-normalized*** number. The implied most significant bit can be used to represent even more accurate significand (23 + 1 = 24 bits) which is called ***subnormal*** representation. *The floating point numbers are to be represented in normalized form*.

The subnormal numbers fall into the category of de-normalized numbers. The subnormal representation slightly reduces the exponent range and can't be normalized since that would result in an exponent which doesn't fit in the field. Subnormal numbers are less accurate, i.e. they have less room for nonzero bits in the fraction field, than normalized numbers. Indeed, the accuracy drops as the size of the subnormal number decreases. However, the subnormal representation is useful in filing gaps of floating point scale near zero.

In other words, the above result can be written as $(-1)^0$ x $1.001_{(2)}$ x 2^2 which yields the integer components as s = 0, b = 2, significand (m) = 1.001, mantissa = 001 and e = 2. The corresponding single precision floating number can be represented in binary as shown below,



```
1  10000001  00100000000000000000000
S     E                M
```

Where the exponent field is supposed to be 2, yet encoded as 129 (127+2) called ***biased exponent***. The exponent field is in plain binary format which also represents negative exponents with an encoding (like sign magnitude, 1's compliment, 2's complement, etc.). The biased exponent is used for representation of negative exponents. The biased exponent has advantages over other negative representations in performing bitwise comparing of two floating point numbers for equality.

A ***bias*** of (2^{n-1} – 1), where **n** is # of bits used in exponent, is added to the exponent (e) to get biased exponent (*E*). So, the biased exponent (*E*) of *single precision* number can be obtained as

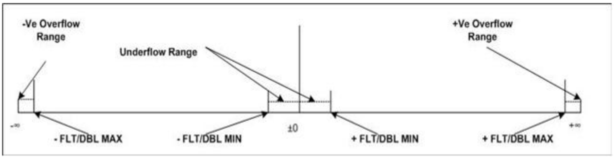
$$\boldsymbol{E = e + 127}$$

The range of exponent in single precision format is -126 to +127. Other values are used for special symbols.

Note: When we unpack a floating point number the exponent obtained is biased exponent. Subtracting 127 from the biased exponent we can extract unbiased exponent.

Float Scale:

The following figure represents floating point scale.



Double Precision Format:

As mentioned in Table – 1 the double precision format has 52 bits for significand (1 represents implied bit), 10 bits for exponent and 1 bit for sign. All other definitions are same for double precision format, except for the size of various components.

Precision:

The smallest change that can be represented in floating point representation is called as precision. The fractional part of a single precision normalized number has exactly 23 bits of resolution, (24 bits with the implied bit). This corresponds to $\log_{(10)} (2^{23}) = 6.924$ = 7 (the characteristic of logarithm) decimal digits of accuracy. Similarly, in case of double precision numbers the precision is $\log_{(10)} (2^{52}) = 15.654$ = 16 decimal digits.

Accuracy:

Accuracy in floating point representation is governed by number of significand bits, whereas range is limited by exponent. Not all real numbers can exactly be represented in floating point format. For any numberwhich is not floating point number, there are two options for floating point approximation, say, the closest floating point number *less than x* as x_ and the closest floating point number *greater than x* as x+. A ***rounding*** operation is performed on number of significant bits in the mantissa field based on the selected mode. The ***round down*** mode causes x set to x_, the ***round up*** mode causes x set to x+, the ***round towards zero*** mode causes x is either x_ or x+ whichever is between zero and. The ***round to nearest*** mode sets x to x_ or x+ whichever is nearest to x. Usually ***round to nearest*** is most used mode. The closeness of floating point representation to the actual value is called as ***accuracy***.

Special Bit Patterns:

The standard defines few special floating point bit patterns. Zero can't have most significant 1 bit, hence can't be normalized. The hidden bit representation requires a special technique for storing zero. We will have two different bit patterns +0 and -0 for the same numerical value zero. For single precision floating point representation, these patterns are given below,

0 00000000 00000000000000000000000 = +0

1 00000000 00000000000000000000000 = -0

Similarly, the standard represents two different bit patters for +INF and -INF. The same are given below,

0 11111111 00000000000000000000000 = +INF

1 11111111 00000000000000000000000 = -INF

All of these special numbers, as well as other special numbers (below) are subnormal numbers, represented through the use of a special bit pattern in the exponent field. This slightly reduces the exponent range, but this is quite acceptable since the range is so large.

An attempt to compute expressions like 0 x INF, 0 ÷ INF, etc. make no mathematical sense. The standard calls the result of such expressions as Not a Number (NaN). Any subsequent expression with NaN yields NaN. The representation of NaN has non-zero significand and all 1s in the exponent field. These are shown below for single precision format (x is don’t care bits),

x 11111111 1**m**000000000000000000000

Where ***m*** can be 0 or 1. This gives us two different representations of NaN.

0 11111111 11000000000000000000000 _____________ Signaling NaN (SNaN)

0 11111111 10000000000000000000000 _____________Quiet NaN (QNaN)

Usually QNaN and SNaN are used for error handling. QNaN do not raise any exceptions as they propagate through most operations. Whereas SNaN are which when consumed by most operations will raise an invalid exception.

Overflow and Underflow:

Overflow is said to occur when the true result of an arithmetic operation is finite but larger in magnitude than the largest floating point number which can be stored using the given precision. ***Underflow*** is said to occur when the true result of an arithmetic operation is smaller in magnitude (infinitesimal) than the smallest normalized floating point number which can be stored. Overflow can’t be ignored in calculations whereas underflow can effectively be replaced by zero.

Endianness:

The IEEE 754 standard defines a binary floating point format. The architecture details are left to the hardware manufacturers. The storage order of individual bytes in binary floating point number varies from architecture to architecture.

Thanks to Venki for writing the above article. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

18. OOD Principles | SOLID

Object Oriented Programming paradigm deals with centralizing data and associated behaviours in a single entity. The entities will communicate by message passing.

The high level languages like C++, Java, C#, etc… provide rich features in designing applications. One can learn the language constructs easily. However, few design principles guide the programmer for better utilization of language features. The following principles help programmer to arrive at flexible class design.

1. Single Responsibility Principle
2. Open Closed Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

All the above five principles are collectively called as SOLID principles. We will have detailed explanation of each principle.

Note that there are few more principles that will be useful in OOD. We will expand the post when respective principles are published (We are sorry, at present the post is a moving target).

Contributed by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

19. Mutex vs Semaphore

What are the differences between Mutex vs Semaphore? When to use mutex and when to use semaphore?

Concrete understanding of Operating System concepts is required to design/develop smart applications. Our objective is to educate the reader on these concepts and learn from other expert geeks.

As per operating system terminology, the mutex and semaphore are kernel resources that provide synchronization services (also called as *synchronization primitives*). *Why do we need such synchronization primitives? Won't be only one sufficient?* To answer these questions, we need to understand few keywords. Please read the posts on atomicity and critical section. We will illustrate with examples to understand these concepts well, rather than following usual OS textual description.

The producer-consumer problem:

Note that the content is generalized explanation. Practical details will vary from implementation.

Consider the standard producer-consumer problem. Assume, we have a buffer of 4096 byte length. A producer thread will collect the data and writes it to the buffer. A consumer thread will process the collected data from the buffer. Objective is, both the threads should not run at the same time.

Using Mutex:

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, only one thread can work with the *entire* buffer. The concept can be generalized using semaphore.

Using Semaphore:

A semaphore is a generalized mutex. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Misconception:

There is an ambiguity between *binary semaphore* and *mutex*. We might have come across that a mutex is binary semaphore. *But they are not*! The purpose of mutex and semaphore are different. May be, due to similarity in their implementation a mutex would be referred as binary semaphore.

Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there will be ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend called you, an interrupt will be triggered upon which an interrupt service routine (ISR) will signal the call processing task to wakeup.

General Questions:

1. Can a thread acquire more than one lock (Mutex)?

Yes, it is possible that a thread will be in need of more than one resource, hence the locks. If any lock is not available the thread will wait (block) on the lock.

2. Can a mutex be locked more than once?

A mutex is a lock. Only one state (locked/unlocked) is associated with it. However, a *recursive mutex* can be locked more than once (POSIX complaint systems), in which a count is associated with it, yet retains only one state (locked/unlocked). The programmer must unlock the mutex as many number times as it was locked.

3. What will happen if a non-recursive mutex is locked more than once.

Deadlock. If a thread which had already locked a mutex, tries to lock the mutex again, it will enter into the waiting list of that mutex, which results in deadlock. It is because no other thread can unlock the mutex. An operating system implementer can exercise care in identifying the owner of mutex and return if it is already locked by same thread to prevent deadlocks.

4. Are binary semaphore and mutex same?

No. We will suggest to treat them separately, as it was explained signalling vs locking mechanisms. But a binary semaphore may experience the same critical issues (e.g. priority inversion) associated with mutex. We will cover these later article.

A programmer can prefer mutex rather than creating a semaphore with count 1.

5. What is a mutex and critical section?

Some operating systems use the same word *critical section* in the API. Usually a mutex is costly operation due to protection protocols associated with it. At last, the objective of mutex is atomic access. There are other ways to achieve atomic access like disabling interrupts which can be much faster but ruins responsiveness. The alternate API makes use of disabling interrupts.

6. What are events?

The semantics of mutex, semaphore, event, critical section, etc… are same. All are synchronization primitives. Based on their cost in using them they are different. We should consult the OS documentation for exact details.

7. Can we acquire mutex/semaphore in an Interrupt Service Routine?

An ISR will run asynchronously in the context of current running thread. It is **not recommended** to query (blocking call) the availability of synchronization primitives in an ISR. The ISR are meant be short, the call to mutex/semaphore may block the current running thread. However, an ISR can signal a semaphore or unlock a mutex.

8. What we mean by “thread blocking on mutex/semaphore” when they are not available?

Every synchronization primitive will have waiting list associated with it. When the resource is not available, the requesting thread will be moved from the running list of processor to the waiting list of the synchronization primitive. When the resource is available, the higher priority thread on the waiting list will get resource (more precisely, it depends on the scheduling policies).

9. Is it necessary that a thread must block always when resource is not available?

Not necessarily. If the design is sure ‘*what has to be done when resource is not available*‘, the thread can take up that work (a different code branch). To support application requirements the OS provides non-blocking API.

For example POSIX pthread_mutex_trylock() API. When the mutex is not available the function will return immediately where as the API pthread_mutex_lock() will block the thread till resource is available.

References:

http://www.netrino.com/node/202

http://doc.trolltech.com/4.7/qsemaphore.html

Also compare mutex/semaphores with Peterson’s algorithm and Dekker’s algorithm. A good reference is the *Art of Concurrency* book. Also explore reader locks and writer locks in Qt documentation.

Exercise:

Implement a program that prints a message “An instance is running” when executed more than once in the same session. For example, if we observe word application or Adobe reader in Windows, we can see only one instance in the task manager. How to implement it?

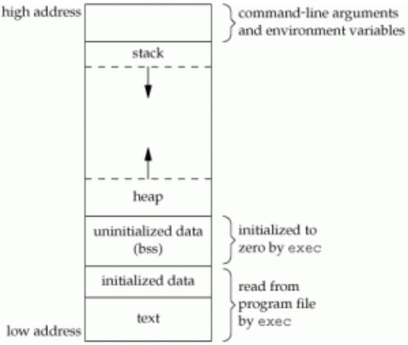
Article compiled by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

20. 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. Stack
5. Heap


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

A typical memory layout of a running process

1. Text 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 global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char* string = “hello world” 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: static int i = 10 will be stored in data segment and global int i = 10 will also be 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. 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.

5. 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.

Examples.

The size(1) command reports the sizes (in bytes) of the text, data, and bss segments. (for more details please refer man page of size(1))

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
```

This article is compiled by **Narendra Kangralkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source:
http://en.wikipedia.org/wiki/Data_segment
http://en.wikipedia.org/wiki/Code_segment
http://en.wikipedia.org/wiki/.bss
http://www.amazon.com/Advanced-Programming-UNIX-Environment-2nd/dp/0201433079

21. 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  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.

```c
/* 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  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.

```c
/* 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  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

This article is compiled by "Narendra Kangralkar" and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

22. Pure Functions

A function is called pure function if it always returns the same result for same argument values and it has no side effects like modifying an argument (or global variable) or outputting something. The only result of calling a pure function is the return value. Examples of pure functions are strlen(), pow(), sqrt() etc. Examples of impure functions are printf(), rand(), time(), etc.

If a function is known as pure to compiler then Loop optimization and subexpression elimination can be applied to it. In GCC, we can mark functions as pure using the "pure" attribute.

```
__attribute__ ((pure)) return-type fun-name(arguments1, …)
{
    /* function body */
}
```

Following is an example pure function that returns square of a passed integer.

```
__attribute__ _((pure)) int my_square(int val)
{
    return val*val;
}
```

Consider the below example

```
for (len = 0; len < strlen(str); ++len)
    printf("%c", toupper(str[len]));
```

If “strlen()” function is not marked as pure function then compiler will invoke the “strlen()” function with each iteration of the loop, and if function is marked as pure function then compiler knows that value of “strlen()” function will be same for each call, that’s why compiler optimizes the for loop and generates code like following.

```
int len = strlen(str);

for (i = 0; i < len; ++i)
    printf("%c", toupper((str[i]));
```

Let us write our own pure function to calculate string length.

```
__attribute__ ((pure)) size_t my_strlen(const char *str)
{
    const char *ptr = str;
    while (*ptr)
       ++ptr;

    return (ptr – str);
}
```

Marking function as pure says that the hypothetical function “my_strlen()” is safe to call fewer times than the program says.

This article is compiled by “Narendra Kangralkar” and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

23. 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: 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 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 accessib
              // 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;
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

24. Analysis of Algorithms | Set 1 (Asymptotic Analysis)

Why performance analysis?
There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?
The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above

things. Another reason for studying performance is – speed is fun!

Given two algorithms for a task, how do we find out which one is better?
One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.
1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.
For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic). To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Does Asymptotic Analysis always work?
Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000nLogn and 2nLogn time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is nLogn). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.
Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

We will covering more on analysis of algorithms in some more posts on this topic.

References:
MIT's Video lecture 1 on Introduction to Algorithms.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

25. Analysis of Algorithms | Set 2 (Worst, Average and Best Cases)

In the previous post, we discussed how Asymptotic analysis overcomes the problems of naive way of analyzing algorithms. In this post, we will take an example of Linear Search and analyze it using Asymptotic analysis.

We can have three cases to analyze an algorithm:
1) Worst Case
2) Average Case
3) Best Case

Let us consider the following implementation of Linear Search.

```
#include <stdio.h>

// Linearly search x in arr[].  If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
       if (arr[i] == x)
         return i;
    }
    return -1;
}

/* Driver program to test above functions*/
int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

Worst Case Analysis (Usually Done)
In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time

complexity of linear search would be $\theta(n)$.

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

```
Average Case Time =


                  =


                  =
```

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\theta(1)$

Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example, Merge Sort. Merge Sort does $\theta(nLogn)$operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

References:

MIT's Video lecture 1 on Introduction to Algorithms.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

26. Reservoir Sampling

Reservoir sampling is a family of randomized algorithms for randomly choosing *k* samples from a list of *n* items, where *n* is either a very large or unknown number. Typically *n* is large enough that the list doesn't fit into main memory. For example, a list of search queries in Google and Facebook.

So we are given a big array (or stream) of numbers (to simplify), and we need to write an efficient function to randomly select *k* numbers where *1* <= *k* <= *n*. Let the input array be *stream[].*

A **simple solution** is to create an array *reservoir[]* of maximum size *k*. One by one randomly select an item from *stream[0..n-1]*. If the selected item is not previously selected, then put it in *reservoir[]*. To check if an item is previously selected or not, we need to search the item in *reservoir[]*. The time complexity of this algorithm will be *O(k^2)*. This can be costly if *k* is big. Also, this is not efficient if the input is in the form of a stream.

It **can be solved in *O(n)* time**. The solution also suits well for input in the form of stream. The idea is similar to this post. Following are the steps.

1) Create an array *reservoir[0..k-1]* and copy first *k* items of *stream[]* to it.
2) Now one by one consider all items from *(k+1)*th item to *n*th item.
…**a)** Generate a random number from 0 to *i* where *i* is index of current item in *stream[]*. Let the generated random number is *j*.
…**b)** If *j* is in range 0 to *k-1*, replace *reservoir[j]* with *arr[i]*

Following is C implementation of the above algorithm.

```
// An efficient program to randomly select k items from a stream of it

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

// A utility function to print an array
void printArray(int stream[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", stream[i]);
    printf("\n");
}

// A function to randomly select k items from stream[0..n-1].
void selectKItems(int stream[], int n, int k)
{
    int i;  // index for elements in stream[]

    // reservoir[] is the output array. Initialize it with
    // first k elements from stream[]
    int reservoir[k];
    for (i = 0; i < k; i++)
        reservoir[i] = stream[i];

    // Use a different seed value so that we don't get
    // same result each time we run this program
    srand(time(NULL));

    // Iterate from the (k+1)th element to nth element
    for (; i < n; i++)
    {
        // Pick a random index from 0 to i.
        int j = rand() % (i+1);

        // If the randomly  picked index is smaller than k, then repla
        // the element present at the index with new element from stre
        if (j < k)
          reservoir[j] = stream[i];
    }

    printf("Following are k randomly selected items \n");
    printArray(reservoir, k);
}

// Driver program to test above function.
int main()
{
    int stream[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    int n = sizeof(stream)/sizeof(stream[0]);
    int k = 5;
    selectKItems(stream, n, k);
    return 0;
}
```

Output:

```
 Following are k randomly selected items
6 2 11 8 12
```

Time Complexity: O(n)

How does this work?
To prove that this solution works perfectly, we must prove that the probability that any item *stream[i]* where *0* <= *i* < *n* will be in final *reservoir[]* is *k/n*. Let us divide the proof in two cases as first *k* items are treated differently.

Case 1: For last *n-k* stream items, i.e., for *stream[i]* where *k* <= *i* < *n*
For every such stream item *stream[i]*, we pick a random index from 0 to *i* and if the picked index is one of the first *k* indexes, we replace the element at picked index with *stream[i]*

To simplify the proof, let us first consider the *last item*. The probability that the last item is in final reservoir = The probability that one of the first *k* indexes is picked for last item = *k/n* (the probability of picking one of the *k* items from a list of size *n*)

Let us now consider the *second last item*. The probability that the second last item is in final *reservoir[]* = [Probability that one of the first *k* indexes is picked in iteration for *stream[n-2]*] X [Probability that the index picked in iteration for *stream[n-1]* is not same as index picked for *stream[n-2]*] = [*k/(n-1)]*[(n-1)/n*] = *k/n*.

Similarly, we can consider other items for all stream items from *stream[n-1]* to *stream[k]* and generalize the proof.

Case 2: For first *k* stream items, i.e., for *stream[i]* where *0* <= *i* < *k*
The first *k* items are initially copied to *reservoir[]* and may be removed later in iterations for *stream[k]* to *stream[n]*.
The probability that an item from *stream[0..k-1]* is in final array = Probability that the item is not picked when items *stream[k], stream[k+1], …. stream[n-1]* are considered = *[k/(k+1)] x [(k+1)/(k+2)] x [(k+2)/(k+3)] x … x [(n-1)/n] = k/n*

References:
http://en.wikipedia.org/wiki/Reservoir_sampling

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

27. The Ubiquitous Binary Search | Set 1

We all aware of binary search algorithm. Binary search is easiest difficult algorithm to get it right. I present some interesting problems that I collected on binary search. There were some requests on binary search.

I request you to honor the code, "I sincerely attempt to solve the problem and ensure there are no corner cases". After reading each problem minimize the browser and try solving it.

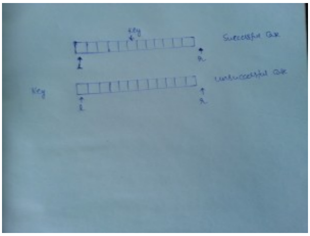
Problem Statement: Given a sorted array of N distinct elements. Find a key in the array using least number of comparisons. (Do you think binary search is optimal to search a key in sorted array?)

Without much theory, here is typical binary search algorithm.

```
// Returns location of key, or -1 if not found
int BinarySearch(int A[], int l, int r, int key)
{
    int m;

    while( l <= r )
    {
        m = l + (r-l)/2;

        if( A[m] == key ) // first comparison
            return m;

        if( A[m] < key ) // second comparison
            l = m + 1;
        else
            r = m - 1;
    }

    return -1;
}
```

Theoretically we need *log N + 1* comparisons in worst case. If we observe, we are using two comparisons per iteration except during final successful match, if any. In practice, comparison would be costly operation, it won't be just primitive type comparison. It is more economical to minimize comparisons as that of theoretical limit.

See below figure on initialize of indices in the next implementation.



The following implementation uses fewer number of comparisons.

```
// Invariant: A[l] <= key and A[r] > key
// Boundary: |r - l| = 1
// Input: A[l .... r-1]
int BinarySearch(int A[], int l, int r, int key)
{
    int m;

    while( r - l > 1 )
    {
        m = l + (r-l)/2;

        if( A[m] <= key )
            l = m;
        else
            r = m;
    }

    if( A[l] == key )
        return l;
    else
        return -1;
}
```

In the while loop we are depending only on one comparison. The search space converges to place *l* and *r* point two different consecutive elements. We need one more comparison to trace search status.

You can see sample test case http://ideone.com/76bad0. (*C++11 code*)

Problem Statement: Given an array of N distinct integers, find floor value of input 'key'. Say, A = {-1, 2, 3, 5, 6, 8, 9, 10} and key = 7, we should return 6 as outcome.

We can use the above optimized implementation to find floor value of key. We keep moving the left pointer to right most as long as the invariant holds. Eventually left pointer points an element less than or equal to key (by definition floor value). The following are possible corner cases,

—> If all elements in the array are smaller than key, left pointer moves till last element.

—> If all elements in the array are greater than key, it is an error condition.

—> If all elements in the array equal and <= key, it is worst case input to our implementation.

Here is implementation,

```
// largest value <= key
// Invariant: A[l] <= key and A[r] > key
// Boundary: |r - l| = 1
// Input: A[l .... r-1]
// Precondition: A[l] <= key <= A[r]
int Floor(int A[], int l, int r, int key)
{
    int m;

    while( r - l > 1 )
    {
        m = l + (r - l)/2;

        if( A[m] <= key )
            l = m;
        else
            r = m;
    }

    return A[l];
}

// Initial call
int Floor(int A[], int size, int key)
{
    // Add error checking if key < A[0]
    if( key < A[0] )
        return -1;

    // Observe boundaries
    return Floor(A, 0, size, key);
}
```

You can see some test cases http://ideone.com/z0Kx4a.

Problem Statement: Given a sorted array with possible duplicate elements. Find number of occurrences of input ‘key’ in *log N* time.

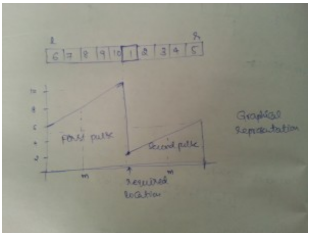
The idea here is finding left and right most occurrences of key in the array using binary search. We can modify floor function to trace right most occurrence and left most occurrence. Here is implementation,

```
// Input: Indices Range [l ... r)
// Invariant: A[l] <= key and A[r] > key
int GetRightPosition(int A[], int l, int r, int key)
{
    int m;

    while( r - l > 1 )
    {
        m = l + (r - l)/2;

        if( A[m] <= key )
            l = m;
        else
            r = m;
    }

    return l;
}

// Input: Indices Range (l ... r]
// Invariant: A[r] >= key and A[l] > key
int GetLeftPosition(int A[], int l, int r, int key)
{
    int m;

    while( r - l > 1 )
    {
        m = l + (r - l)/2;

        if( A[m] >= key )
            r = m;
        else
            l = m;
    }

    return r;
}

int CountOccurances(int A[], int size, int key)
{
    // Observe boundary conditions
    int left = GetLeftPosition(A, -1, size-1, key);
    int right = GetRightPosition(A, 0, size, key);

    // What if the element doesn't exists in the array?
    // The checks helps to trace that element exists
    return (A[left] == key && key == A[right])?
           (right - left + 1) : 0;
}
```

Sample code http://ideone.com/zn6R6a.

Problem Statement: Given a sorted array of distinct elements, and the array is rotated at an unknown position. Find minimum element in the array.

We can see pictorial representation of sample input array in the below figure.



We converge the search space till *l* and *r* points single element. If the middle location falls in the first pulse, the condition A[m] < A[r] doesn't satisfy, we exclude the range A[m+1 … r]. If the middle location falls in the second pulse, the condition A[m] < A[r] satisfied, we converge our search space to A[m+1 … r]. At every iteration we check for search space size, if it is 1, we are done.

Given below is implementation of algorithm. *Can you come up with different implementation?*

```
int BinarySearchIndexOfMinimumRotatedArray(int A[], int l, int r)
{
    // extreme condition, size zero or size two
    int m;

    // Precondition: A[l] > A[r]
    if( A[l] <= A[r] )
        return l;

    while( l <= r )
    {
        // Termination condition (l will eventually falls on r, and r
        // point minimum possible value)
        if( l == r )
            return l;

        m = l + (r-l)/2; // 'm' can fall in first pulse,
                         // second pulse or exactly in the middle

        if( A[m] < A[r] )
            // min can't be in the range
            // (m < i <= r), we can exclude A[m+1 ... r]
            r = m;
        else
            // min must be in the range (m < i <= r),
            // we must search in A[m+1 ... r]
            l = m+1;
    }

    return -1;
}

int BinarySearchIndexOfMinimumRotatedArray(int A[], int size)
{
    return BinarySearchIndexOfMinimumRotatedArray(A, 0, size-1);
}
```

See sample test cases http://ideone.com/KbwDrk.

Exercises:

1. A function called *signum(x, y)* is defined as,

```
signum(x, y) = -1 if x < y
             =  0 if x = y
             =  1 if x > y
```

Did you come across any instruction set in which a comparison behaves like *signum* function? Can it make first implementation of binary search optimal?

2. Implement ceil function replica of floor function.

3. Discuss with your friends on “Is binary search optimal (results in least number of comparisons)? Why not ternary search or interpolation search on sorted array? When do you prefer ternary or interpolation search over binary search?”

4. Draw a tree representation of binary search (believe me, it helps you a lot to understand many internals of binary search).

Stay tuned, I will cover few more interesting problems using binary search in upcoming articles. I welcome your comments.

– – – by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

28. Static and Dynamic Libraries | Set 1

When a C program is compiled, the compiler generates object code. After generating the object code, the compiler also invokes linker. One of the main tasks for linker is to make code of library functions (eg printf(), scanf(), sqrt(), ..etc) available to your program. A linker can accomplish this task in two ways, by copying the code of library function to your object code, or by making some arrangements so that the complete code of library functions is not copied, but made available at run-time.

Static Linking and Static Libraries is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are, ***.a*** files in Linux and ***.lib*** files in Windows.

Steps to create a static library Let us create and use a Static Library in UNIX or UNIX like OS.

1. Create a C file that contains functions in your library.

```
/* Filename: lib_mylib.c */
#include <stdio.h>
void fun(void)
{
  printf("fun() called from a static library");
}
```

We have created only one file for simplicity. We can also create multiple files in a library.

2. Create a header file for the library

```
/* Filename: lib_mylib.h */
void fun(void);
```

3. Compile library files.

```
gcc -c lib_mylib.c -o lib_mylib.o
```

4. Create static library. This step is to bundle multiple object files in one static library (see ar for details). The output of this step is static library.

```
ar rcs lib_mylib.a lib_mylib.o
```

5. Now our static library is ready to use. At this point we could just copy lib_hello_static.a somewhere else to use it. For demo purposes, let us keep the library in the current directory.

Let us create a driver program that uses above created static library.

1. Create a C file with main function

```
/* filename: driver.c  */
#include "lib_mylib.h"
void main()
{
  fun();
}
```

2. Compile the driver program.

```
gcc -c driver.c -o driver.o
```

3. Link the compiled driver program to the static library. Note that -L. is used to tell that the static library is in current folder (See this for details of -L and -l options).

```
gcc -o driver driver.o -L. -l_mylib
```

4. Run the driver program

```
./driver
fun() called from a static library
```

Following are some important points about static libraries.

1. For a static library, the actual code is extracted from the library by the linker and used to build the final executable at the point you compile/build your application.

2. Each process gets its own copy of the code and data. Where as in case of dynamic libraries it is only code shared, data is specific to each process. For static libraries memory footprints are larger. For example, if all the window system tools were statically linked, several tens of megabytes of RAM would be wasted for a typical user, and the user would be slowed down by a lot of paging.

3. Since library code is connected at compile time, the final executable has no dependencies on the the library at run time i.e. no additional run-time loading costs, it means that you don't need to carry along a copy of the library that is being used and you have everything under your control and there is no dependency.

4. In static libraries, once everything is bundled into your application, you don't have to worry that the client will have the right library (and version) available on their system.

5. One drawback of static libraries is, for any change(up-gradation) in the static libraries, you have to recompile the main program every time.

6. One major advantage of static libraries being preferred even now "is speed". There will be no dynamic querying of symbols in static libraries. Many production line software use static libraries even today.

Dynamic linking and Dynamic Libraries Dynamic Linking doesn't require the code to be copied, it is done by just placing name of the library in the binary file. The actual linking happens when the program is run, when both the binary file and the library are in memory. Examples of Dynamic libraries (libraries which are linked at run-time) are, ***.so*** in Linux and ***.dll*** in Windows.

We will soon be covering more points on Dynamic Libraries and steps to create them.

This article is compiled by **Abhijit Saha** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

29. NP-Completeness | Set 1 (Introduction)

We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

Can all computational problems be solved by a computer? There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source Halting Problem).
Status of NP Complete problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

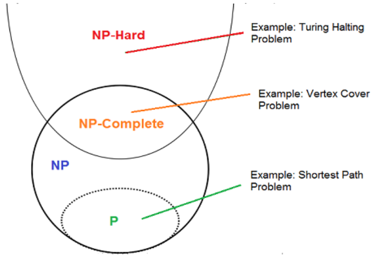
What are NP, P, NP-complete and NP-Hard problems?
P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time.

NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).
Informally, NP is set of decision problems which can be solved by a polynomial time via a “Lucky Algorithm”, a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:

1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn’t need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.


NP-Hard
Example: Turing Halting Problem
Example: Vertex Cover Problem
NP-Complete
NP
Example: Shortest Path Problem
P

This diagram assumes that P != NP

Decision vs Optimization Problems

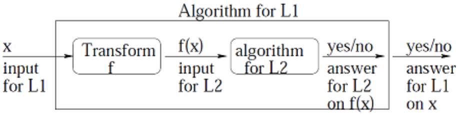
NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref 2).

For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not.

The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

What was the first problem proved as NP-Complete?

There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (Boolean satisfiability problem) is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

We will soon be discussing more NP-Complete problems and their proof for NP-Completeness.

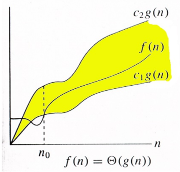
References:

MIT Video Lecture on Computational Complexity
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
http://www.ics.uci.edu/~eppstein/161/960312.html

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

30. Analysis of Algorithms | Set 3 (Asymptotic Notations)

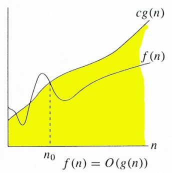
We have discussed Asymptotic Analysis, and Worst, Average and Best Cases of Algorithms. The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.



1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.
$3n^3 + 6n^2 + 6000 = \Theta(n^3)$
Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ beats $\Theta(n^2)$ irrespective of the constants involved.
For a given function g(n), we denote $\Theta(g(n))$ is following set of functions.

```
((g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that
                  0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}
```

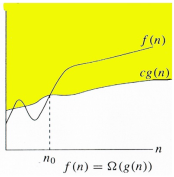
The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.
If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

```
O(g(n)) = { f(n): there exist positive constants c and n0 such that
            0 <= f(n) <= cg(n) for all n >= n0}
```



3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation< can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

```
 (g(n)) = {f(n): there exist positive constants c and n0 such that
                          0 <= cg(n) <= f(n) for all n >= n0}.
```

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

Exercise:
Which of the following statements is/are valid?
1. Time Complexity of QuickSort is Θ(n^2)

2. Time Complexity of QuickSort is O(n^2)
3. For any two functions f(n) and g(n), we have f(n) = Θ (g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).
4. Time complexity of all computer algorithms can be written as Ω (1)

References:

Lec 1 | MIT (Introduction to Algorithms)

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

31. Analysis of Algorithms | Set 4 (Analysis of Loops)

We have discussed Asymptotic Analysis, Worst, Average and Best Cases and Asymptotic Notations in previous posts. In this post, analysis of iterative programs with simple examples is discussed.

1) O(1): Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

```
// set of non-recursive and non-loop statements
```

For example swap() function has O(1) time complexity.
A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
     // some O(1) expressions
}
```

2) O(n): Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}
```

```
for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

3) $O(n^c)$: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have $O(n^2)$ time complexity

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
       // some O(1) expressions
    }
}

for (int i = n; i > 0; i += c) {
    for (int j = i+1; j <=n; j += c) {
       // some O(1) expressions
}
```

For example Selection sort and Insertion Sort have $O(n^2)$ time complexity.

4) O(Logn) Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

For example Binary Search(refer iterative implementation) has O(Logn) time complexity.

5) O(LogLogn) Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
```

```
for (int i = n; i > 0; i = fun(i)) {
    // some O(1) expressions
}
```

See this for more explanation.

How to combine time complexities of consecutive loops?
When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c) {
     // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
     // some O(1) expressions
}
Time complexity of above code is O(m) + O(n) which is O(m+n)
If m == n, the time complexity becomes O(2n) which is O(n).
```

How to calculate time complexity when there are many if, else statements inside loops?
As discussed here, worst case time complexity is the most useful among best, average and worst. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.
For example consider the linear search function where we consider the case when element is present at the end or not present at all.
When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.

How to calculate time complexity of recursive functions?
Time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences. We will soon be discussing recurrence solving techniques as a separate post.

Quiz on Analysis of Algorithms

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

32. A Problem in Many Binary Search Implementations

Consider the following C implementation of Binary Search function, is there anything wrong in this?

```
// A iterative binary search function. It returns location of x in
// given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        // find index of middle element
        int m = (l+r)/2;

        // Check if x is present at mid
        if (arr[m] == x) return m;

        // If x greater, ignore left half
        if (arr[m] < x) l = m + 1;

        // If x is smaller, ignore right half
        else r = m - 1;
    }

    // if we reach here, then element was not present
    return -1;
}
```

The above looks fine except one subtle thing, the expression “m = (l+r)/2″. It fails for large values of l and r. Specifically, it fails if the sum of low and high is greater than the maximum positive int value (2^{31} – 1). The sum overflows to a negative value, and the value stays negative when divided by two. In C this causes an array index out of bounds with unpredictable results.

What is the way to resolve this problem?

Following is one way:

```
        int mid = low + ((high - low) / 2);
```

Probably faster, and arguably as clear is (works only in Java, refer this):

```
        int mid = (low + high) >>> 1;
```

In C and C++ (where you don’t have the >>> operator), you can do this:

```
        mid = ((unsigned int)low + (unsigned int)high)) >> 1
```

The similar problem appears in Merge Sort as well.

The above content is taken from google reasearch blog.

Please refer this as well, it points out that the above solutions may not always work.

The above problem occurs when array length is 2^{30} or greater and the search repeatedly moves to second half of the array. This much size of array is not likely to appear most of the time. For example, when we try the below program with 32 bit Code Blocks compiler, we get compiler error.

```
int main()
{
    int arr[1<<30];
    return 0;
}
```

Output:

```
error: size of array 'arr' is too large
```

Even when we try boolean array, the program compiles fine, but crashes when run in Windows 7.0 and Code Blocks 32 bit compiler

```
#include <stdbool.h>
int main()
{
    bool arr[1<<30];
    return 0;
}
```

Output: No compiler error, but crashes at run time.

Sources:
http://googleresearch.blogspot.in/2006/06/extra-extra-read-all-about-it-nearly.html
http://locklessinc.com/articles/binary_search/

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

33. An interesting time complexity question

What is the time complexity of following function fun()?

```
int fun(int n)
{
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j < n; j += i)
        {
            // Some O(1) task
        }
    }
}
```

For i = 1, the inner loop is executed n times.

For i = 2, the inner loop is executed approximately n/2 times.
For i = 3, the inner loop is executed approximately n/3 times.
For i = 4, the inner loop is executed approximately n/4 times.
……………………………………………………….
……………………………………………………….
For i = n, the inner loop is executed approximately n/n times.

So the total time complexity of the above algorithm is (n + n/2 + n/3 + … + n/n)

Which becomes n * (1/1 + 1/2 + 1/3 + … + 1/n)

The important thing about series (1/1 + 1/2 + 1/3 + … + 1/n) is, it is equal to Θ(Logn) (See this for reference). So the time complexity of the above code is Θ(nLogn).

As a side note, the sum of infinite hormonic series is counterintuitive as the series diverges. The value of $\sum_{n=1}^{\infty} \frac{1}{n}$ is ∞. This is unlike geometric series as geometric series with ratio less than 1 converges.

Reference:
http://en.wikipedia.org/wiki/Harmonic_series_%28mathematics%29#Rate_of_divergence
http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap03.htm

This article is contributed by **Rahul**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above