# Function Pointer in C:-

In C, like normal data pointers (int \*, char \*, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

|  |
| --- |
| #include <stdio.h>  // A normal function with an int parameter  // and void return type  voidfun(inta)  {      printf("Value of a is %d\n", a);  }    intmain()  {      // fun\_ptr is a pointer to function fun()      void(\*fun\_ptr)(int) = &fun;        /\* The above line is equivalent of following two         void (\*fun\_ptr)(int);         fun\_ptr = &fun;      \*/        // Invoking fun() using fun\_ptr      (\*fun\_ptr)(10);        return0;  } |

# Implement Your Own sizeof:-

Here is an implementation.

|  |
| --- |
| #define my\_sizeof(type) (char \*)(&type+1)-(char\*)(&type)  intmain()  {      doublex;      printf("%d", my\_sizeof(x));      getchar();      return0;  } |

* When we use ***include***directive,  the contents of included header file (after preprocessing) are copied to the current file.  
  Angular brackets **<** and **>** instruct the preprocessor to look in the standard folder where all header files are held.  Double quotes **“** and **“** instruct the preprocessor to look into the current folder and if the file is not present in current folder, then in standard folder of all header files.

# Structure Padding:-

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

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

..

struct student

{

       int id1;

       int id2;

       char a;

       char b;

       float percentage;

};

..

* As per C concepts, int and float datatypes occupy 4 bytes each and char datatype occupies 1 byte for 32 bit processor. So, only 14 bytes (4+4+1+1+4) should be allocated for above structure.
* But, this is wrong.  Do you know why?
* Architecture of a computer processor is such a way that it can read 1 word from memory at a time.
* 1 word is equal to 4 bytes for 32 bit processor and 8 bytes for 64 bit processor. So, 32 bit processor always reads 4 bytes at a time and 64 bit processor always reads 8 bytes at a time.
* This concept is very useful to increase the processor speed.
* To make use of this advantage, memory is arranged as a group of 4 bytes in 32 bit processor and 8 bytes in 64 bit processor.
* Below C program is compiled and executed in 32 bit compiler. Please check memory allocated for structure1 and structure2 in below program.

#### ****Example program for structure padding in C:****

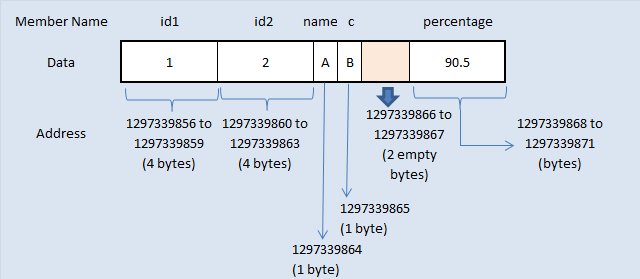
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | #include <stdio.h>  #include <string.h>    /\*  Below structure1 and structure2 are same.      They differ only in member's allignment \*/    structstructure1  {         intid1;         intid2;         charname;         charc;         floatpercentage;  };    structstructure2  {         intid1;         charname;         intid2;         charc;         floatpercentage;  };    intmain()  {      structstructure1a;      structstructure2b;        printf("size of structure1 in bytes : %d\n",              sizeof(a));      printf("\n   Address of id1        = %u",&a.id1);      printf("\n   Address of id2        = %u",&a.id2);      printf("\n   Address of name       = %u",&a.name);      printf("\n   Address of c          = %u",&a.c);      printf("\n   Address of percentage = %u",                     &a.percentage);        printf("   \n\nsize of structure2 in bytes : %d\n",                     sizeof(b));      printf("\n   Address of id1        = %u",&b.id1);      printf("\n   Address of name       = %u",&b.name);      printf("\n   Address of id2        = %u",&b.id2);      printf("\n   Address of c          = %u",&b.c);      printf("\n   Address of percentage = %u",                     &b.percentage);      getchar();      return0;  } |

#### ****Output:****

|  |
| --- |
| **size of structure1 in bytes : 16** Address of id1 = 1297339856 Address of id2 = 1297339860 Address of name = 1297339864 Address of c = 1297339865 Address of percentage = 1297339868 **size of structure2 in bytes : 20** Address of id1 = 1297339824 Address of name = 1297339828 Address of id2 = 1297339832 Address of c = 1297339836 Address of percentage = 1297339840 |

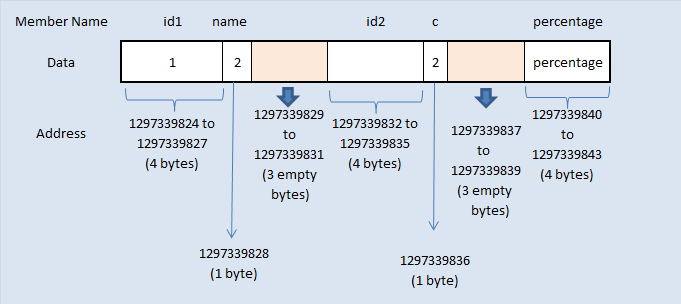
#### ****Structure padding analysis for above C program:****

#### ****Memory allocation for structure1:****



* In above program, memory for structure1 is allocated sequentially for first 4 members.
* Whereas, memory for 5th member “percentage” is not allocated immediate next to the end of member “c”.
* There are only 2 bytes remaining in the package of 4 bytes after memory allocated to member “c”.
* Range of this 4 byte package is from 1297339864 to 1297339867.
* Addresses 1297339864  and 1297339865 are used for members “name and c”. Addresses 1297339866  and 1297339867 only is available in this package.
* But, member “percentage” is datatype of float and requires 4 bytes. It can’t be stored in the same memory package as it requires 4 bytes. Only 2 bytes are free in that package.
* So, next 4 byte of memory package is chosen to store percentage data which is from 1297339868 to 1297339871.
* Because of this, memory 1297339866 and 1297339867 are not used by the program and those 2 bytes are left empty.
* So, size of structure1 is 16 bytes which is 2 bytes extra than what we think. Because, 2 bytes are left empty.

#### ****Memory allocation for structure2:****



* Memory for structure2 is also allocated as same as above concept. Please note that structure1 and structure2 are same. But, they differ only in the order of the members declared inside the structure.
* 4 bytes of memory is allocated for 1st structure member “id1” which occupies whole 4 byte of memory package.
* Then, 2nd structure member “name” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty. Because, 3rd structure member “id2” of datatype integer requires whole 4 byte of memory in the package. But, this is not possible as only 3 bytes available in the package.
* So, next whole 4 byte package is used for structure member “id2”.
* Again, 4th structure member “c” occupies only 1 byte of memory in next 4 byte package and remaining 3 bytes are left empty.
* Because, 5th structure member “percentage” of datatype float requires whole 4 byte of memory in the package.
* But, this is also not possible as only 3 bytes available in the package. So, next whole 4 byte package is used for structure member “percentage”.
* So, size of structure2 is 20 bytes which is 6 bytes extra than what we think. Because, 6 bytes are left empty.

#### ****How to avoid structure padding in C?****

* #pragma pack ( 1 ) directive can be used for arranging memory for structure members very next to the end of other structure members.
* VC++ supports this feature. But, some compilers such as Turbo C/C++ does not support this feature.

# Thread vs Process:-

**Threads** on the same **process** are much more lightweight. They reside on the same memory space and they can share handles, security context. Switching between **threads** on the same **process** is much **faster** because OS only switches registers, not memory mapping.

1. Inter-thread communication (sharing data etc.) is significantly simpler to program than inter-process communication.
2. Context switches between threads are faster than between processes. That is, it's quicker for the OS to stop one thread and start running another than do the same with two processes.

# Segmentation Fault:-

A segmentation fault occurs when a program attempts to access a [memory](https://en.wikipedia.org/wiki/Computer_memory) location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed (for example, attempting to write to a [read-only](https://en.wikipedia.org/wiki/Read-only_memory) location, or to overwrite part of the [operating system](https://en.wikipedia.org/wiki/Operating_system)).

Only the program's own address space is readable, and of this, only the stack and the read-write portion of the [data segment](https://en.wikipedia.org/wiki/Data_segment) of a program are writable, while read-only data and the [code segment](https://en.wikipedia.org/wiki/Code_segment) are not writable. Thus attempting to read outside of the program's address space, or writing to a read-only segment of the address space, results in a segmentation fault.

### Writing to read-only memory

Writing to read-only memory raises a segmentation fault. At the level of code errors, this occurs when the program writes to part of its own [code segment](https://en.wikipedia.org/wiki/Code_segment) or the read-only portion of the [data segment](https://en.wikipedia.org/wiki/Data_segment), as these are loaded by the OS into read-only memory.

intmain(void)

{

char\*s="hello world";

\*s='H';

}

When the program containing this code is compiled, the string "hello world" is placed in the [rodata](https://en.wikipedia.org/wiki/Rodata) section of the program [executable file](https://en.wikipedia.org/wiki/Executable_file): the read-only section of the [data segment](https://en.wikipedia.org/wiki/Data_segment). When loaded, the operating system places it with other strings and [constant](https://en.wikipedia.org/wiki/Constant_(programming)) data in a read-only segment of memory. When executed, a variable, s, is set to point to the string's location, and an attempt is made to write an H character through the variable into the memory, causing a segmentation fault. Compiling such a program with a compiler that does not check for the assignment of read-only locations at compile time, and running it on a Unix-like operating system produces the following [runtime error](https://en.wikipedia.org/wiki/Runtime_error):

$ gccsegfault.c -g -o segfault

$ ./segfault

Segmentation fault

[Backtrace](https://en.wikipedia.org/wiki/Backtrace) of the core file from [GDB](https://en.wikipedia.org/wiki/GDB):

ProgramreceivedsignalSIGSEGV,Segmentationfault.

0x1c0005c2inmain()atsegfault.c:6

6\*s='H';

This code can be corrected by using an array instead of a character pointer, as this allocates memory on stack and initializes it to the value of the string literal:

chars[]="hello world";

s[0]='H';*// equivalently, \*s = 'H';*

### Null pointer dereference

Because a very common program error is a [null pointer](https://en.wikipedia.org/wiki/Null_pointer) [dereference](https://en.wikipedia.org/wiki/Dereference_operator) (a read or write through a null pointer, used in C to mean "pointer to no object" and as an error indicator), most operating systems map the null pointer's address such that accessing it causes a segmentation fault.

int\*ptr=NULL;

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

This sample code creates a [null pointer](https://en.wikipedia.org/wiki/Null_pointer), and then tries to access its value (read the value). Doing so causes a segmentation fault at runtime on many operating systems.

Dereferencing a null pointer and then assigning to it (writing a value to a non-existent target) also usually causes a segmentation fault:

int\*ptr=NULL;

\*ptr=1;

The following code includes a null pointer dereference, but when compiled will often not result in a segmentation fault, as the value is unused and thus the dereference will often be optimized away by [dead code elimination](https://en.wikipedia.org/wiki/Dead_code_elimination):

int\*ptr=NULL;

\*ptr;

Another segfault happens when you try to write to a portion of memory that was marked as read-only:

char \*str = "Foo"; // Compiler marks the constant string as read-only

\*str = 'b'; // Which means this is illegal and results in a segfault

Dangling pointer points to a thing that does not exist any more, like here:

char \*p = NULL;

{

char c;

p = &c;

}

// Now p is dangling

The pointer p dangles because it points to character variable c that ceased to exist after the block ended. And when you try to dereference dangling pointer (like \*p='A'), you would probably get a segfault.

# Structure vs Union:-

The difference between structure and union is,  
1. The amount of memory required to store a structure variable is the sum of the size of all the members.  
On the other hand, in case of unions, the amount of memory required is always equal to that required by its largest member.

2. In case of structure, each member have their own memory space but In union, one block is used by all the member of the union.

**Detailed Example:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | structstu  {  char c;  int l;  float p;  }; |
| 1  2  3  4  5  6 | unionemp  {  char c;  int l;  float p;  }; |

**In the above example size of the structure stu is 7 and size of union emp is 4.**

# TCP vs UDP:-

|  | **TCP** | **UDP** | |
| --- | --- | --- | --- |
| **Acronym for** | Transmission Control Protocol | User Datagram Protocol or Universal Datagram Protocol |
| **Connection** | TCP is a connection-oriented protocol. | UDP is a connectionless protocol. |
| **Function** | As a message makes its way across the [internet](http://www.diffen.com/difference/Internet_vs_World_Wide_Web) from one computer to another. This is connection based. | UDP is also a protocol used in message transport or transfer. This is not connection based which means that one program can send a load of packets to another and that would be the end of the relationship. |
| **Usage** | TCP is suited for applications that require high reliability, and transmission time is relatively less critical. | UDP is suitable for applications that need fast, efficient transmission, such as games. UDP's stateless nature is also useful for servers that answer small queries from huge numbers of clients. |
| **Use by other protocols** | HTTP, HTTPs, FTP, SMTP, Telnet | DNS, DHCP, TFTP, SNMP, RIP, VOIP. |
| **Ordering of data packets** | TCP rearranges [data](http://www.diffen.com/difference/Data_vs_Information) packets in the order specified. | UDP has no inherent order as all packets are independent of each other. If ordering is required, it has to be managed by the application layer. |
| **Speed of transfer** | The speed for TCP is slower than UDP. | UDP is faster because error recovery is not attempted. It is a "best effort" protocol. |
| **Reliability** | There is absolute guarantee that the data transferred remains intact and arrives in the same order in which it was sent. | There is no guarantee that the messages or packets sent would reach at all. |
| **Header Size** | TCP header size is 20 bytes | UDP Header size is 8 bytes. |
| **Common Header Fields** | Source port, Destination port, Check Sum | Source port, Destination port, Check Sum |
| **Streaming of data** | Data is read as a byte stream, no distinguishing indications are transmitted to signal message (segment) boundaries. | Packets are sent individually and are checked for integrity only if they arrive. Packets have definite boundaries which are honored upon receipt, meaning a read operation at the receiver socket will yield an entire message as it was originally sent. |
| **Weight** | TCP is heavy-weight. TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control. | UDP is lightweight. There is no ordering of messages, no tracking connections, etc. It is a small transport layer designed on top of IP. |
| **Data Flow Control** | TCP does Flow Control. TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control. | UDP does not have an option for flow control |
| **Error Checking** | TCP does error checking and error recovery. Erroneous packets are retransmitted from the source to the destination. | UDP does error checking but simply discards erroneous packets. Error recovery is not attempted. |
| **Fields** | 1. Sequence Number, 2. AcK number, 3. Data offset, 4. Reserved, 5. Control bit, 6. Window, 7. Urgent Pointer 8. Options, 9. Padding, 10. Check Sum, 11. Source port, 12. Destination port | 1. Length, 2. Source port, 3. Destination port, 4. Check Sum |
| **Acknowledgement** | Acknowledgement segments | No Acknowledgment |
| **Handshake** | SYN, SYN-ACK, ACK | No handshake (connectionless protocol) |

# Dangling pointer:-

If any pointer is pointing the memory address of any variable but after some variable has deleted from that memory location while pointer is still pointing such memory location. Such pointer is known as dangling pointer and this problem is known as dangling pointer problem.

# Wild Pointer:-

# A pointer in c which has not been initialized is known as wild pointer.

# C program to find size of structure without using sizeofoperator:-

# #include <stdio.h>

# #define my\_sizeof(type) (char \*)(&type+1)-(char\*)(&type)

# #pragma pack(1)

# struct ABC{

# int a;

# float b;

# char c;

# };

# #pragma pop()

# int main()

# {

# struct ABC x;

# printf("Hello, World!\n");

# printf("%d\n", my\_sizeof(x));

# return 0;

# }

# Difference between including the header file with-in angular braces <> and double quotes “ “ :-

If a header file is included with in <> then the compiler searches for the particular header file only with in the built in include path. If a header file is included with in “ “, then the compiler searches for the particular header file first in the current working directory, if not found then in the built in include path.

**Fork() system call:-**

# Fork is a system call used for creating child processes of a parent process.it returns the process id of the created child process. After that pid(process id ) is checked if its negative, it means no child process is created Pid==0implies the id of the newly created process and pid>0 is the id of child process given to the parent process. The statements following fork system call are executed by both the parent and child process. And one more thing, the parent and child process have the exact copy of address space but it exist separately for the two processes.

**Deadlock:-**

# Deadlock occurs because of  1.no preemption 2.circular waiting i.e..when A process is waiting for the resources which are held by the processB,which is waiting  for the resources engaged by process A.

**C Constant Pointers and Pointer to Constants:-**

### Constant Pointers

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

A constant pointer is declared as follows :

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

An example declaration would look like :

int \* const ptr;

Lets take a small code to illustrate these type of pointers :

#include<stdio.h>

int main(void)

{

int var1 = 0, var2 = 0;

int \*const ptr = &var1;

ptr = &var2;

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

return 0;

}

In the above example :

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

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

Lets now compile the program :

$ gcc -Wall constptr.c -o constptr

constptr.c: In function ‘main’:

constptr.c:7: error: assignment of read-only variable ‘ptr’

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

### Pointer to Constant

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

A pointer to constant is defined as :

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

An example of definition could be :

const int\* ptr;

Lets take a small code to illustrate a pointer to a constant :

#include<stdio.h>

int main(void)

{

int var1 = 0;

const int\* ptr = &var1;

\*ptr = 1;

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

 return 0;

}

In the code above :

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

Now, when the above program is compiled :

$ gcc -Wall constptr.c -o constptr

constptr.c: In function ‘main’:

constptr.c:7: error: assignment of read-only location ‘\*ptr’

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

### Constant Pointer to a Constant

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

A constant pointer to constant is defined as :

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

for example :

const int\* const ptr;

Lets look at a piece of code to understand this :

#include<stdio.h>

int main(void)

{

int var1 = 0,var2 = 0;

const int\* const ptr = &var1;

\*ptr = 1;

ptr = &var2;

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

 return 0;

}

In the code above :

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

When the code was compiled :

$ gcc -Wall constptr.c -o constptr

constptr.c: In function ‘main’:

constptr.c:7: error: assignment of read-only location ‘\*ptr’

constptr.c:8: error: assignment of read-only variable ‘ptr’

# Array of Function Pointer :-

|  |  |
| --- | --- |
|  | You'd declare an array of function pointers as  T (\*afp[N])();  for some type T. Since you're dynamically allocating the array, you'd do something like  T (\*\*pfp)() = calloc(num\_elements, sizeof \*pfp);  or  T (\*\*pfp)() = malloc(num\_elements \* sizeof \*pfp);  You'd then call each function as  T x = (\*pfp[i])();  or  T x = pfp[i](); // pfp[i] is implicitly dereferenced  If you want to be unorthodox, you can declare a pointer to an array of pointers to functions, and then allocate that as follows:  T (\*(\*pafp)[N])() = malloc(sizeof \*pafp);  although you would have to deference the array pointer when making the call:  x = (\*(\*pafp)[i])(); |

# Find the Missing Number:-

1. Get the sum of numbers

total = n\*(n+1)/2

2 Subtract all the numbers from sum and

you will get the missing number.

|  |
| --- |
| #include<stdio.h>    /\* getMissingNo takes array and size of array as arguments\*/  int getMissingNo (int a[], int n)  {      int i, total;      total  = (n+1)\*(n+2)/2;      for ( i = 0; i< n; i++)         total -= a[i];      return total;  }    /\*program to test above function \*/  int main()  {      int a[] = {1,2,4,5,6};      int miss = getMissingNo(a,5);      printf("%d", miss);      getchar();  } |

# Write your own memcpy() and memmove()

The [memcpy](http://geeksquiz.com/memcpy-in-cc/)function is used to copy a block of data from a source address to a destination address. Below is its prototype.

void \* memcpy(void \* destination, const void \* source, size\_t num);

The idea is to simply typecast given addresses to char \*(char takes 1 byte). Then one by one copy data from source to destination. Below is implementation of this idea.

|  |
| --- |
| // A C implementation of memcpy()  #include<stdio.h>  #include<string.h>   void myMemCpy(void \*dest, void \*src, size\_t n)  {     // Typecast src and dest addresses to (char \*)     char \*csrc = (char \*)src;     char \*cdest = (char \*)dest;      // Copy contents of src[] to dest[]     for (int i=0; i<n; i++)         cdest[i] = csrc[i];  }   // Driver program  int main()  {     char csrc[] = "GeeksforGeeks";     char cdest[100];     myMemCpy(cdest, csrc, strlen(csrc)+1);     printf("Copied string is %s", cdest);      int isrc[] = {10, 20, 30, 40, 50};     int n = sizeof(isrc)/sizeof(isrc[0]);     int idest[n], i;     myMemCpy(idest, isrc,  sizeof(isrc));     printf("\nCopied array is ");     for (i=0; i<n; i++)       printf("%d ", idest[i]);     return 0;  } |

Run on IDE

Output:

Copied string is GeeksforGeeks

Copied array is 10 20 30 40 50

## What is [memmove()](http://geeksquiz.com/memmove-in-cc/)?

memmove() is similar to memcpy() as it also copies data from a source to destination. memcpy() leads to problems when source and destination addresses overlap as memcpy() simply copies data one by one from one location to another. For example consider below program.

|  |
| --- |
| // Sample program to show that memcpy() can loose data.  #include <stdio.h>  #include <string.h>  int main()  {     char csrc[100] = "Geeksfor";     memcpy(csrc+5, csrc, strlen(csrc)+1);     printf("%s", csrc);     return 0;  } |

Run on IDE

Output:

GeeksGeeksGeek

Since the input addresses are overlapping, the above program overwrites the original string and causes data loss.

|  |
| --- |
| // Sample program to show that memmove() is better than memcpy()  // when addresses overlap.  #include <stdio.h>  #include <string.h>  int main()  {     char csrc[100] = "Geeksfor";     memmove(csrc+5, csrc, strlen(csrc)+1);     printf("%s", csrc);     return 0;  } |

Output:

GeeksGeeksfor

## How to implement memmove()?

The trick here is to use a temp array instead of directly copying from src to dest. The use of temp array is important to handle cases when source and destination addresses are overlapping.

|  |
| --- |
| //C++ program to demonstrate implementation of memmove()  #include<stdio.h>  #include<string.h>    // A function to copy block of 'n' bytes from source  // address 'src' to destination address 'dest'.  void myMemMove(void \*dest, void \*src, size\_t n)  {     // Typecast src and dest addresses to (char \*)     char \*csrc = (char \*)src;     char \*cdest = (char \*)dest;      // Create a temporary array to hold data of src     char \*temp = new char[n];      // Copy data from csrc[] to temp[]     for (int i=0; i<n; i++)         temp[i] = csrc[i];      // Copy data from temp[] to cdest[]     for (int i=0; i<n; i++)         cdest[i] = temp[i];       delete [] temp;  }   // Driver program  int main()  {     char csrc[100] = "Geeksfor";     myMemMove(csrc+5, csrc, strlen(csrc)+1);     printf("%s", csrc);     return 0;  } |

Output:

GeeksGeeksfor

## Setting a bit

Use the bitwise OR operator (|) to set a bit.

number |= 1 << x;

That will set bit x.

## Clearing a bit

Use the bitwise AND operator (&) to clear a bit.

number &= ~(1 << x);

That will clear bit x. You must invert the bit string with the bitwise NOT operator (~), then AND it.

## Toggling a bit

The XOR operator (^) can be used to toggle a bit.

number ^= 1 << x;

That will toggle bit x.

## Checking a bit

You didn't ask for this but I might as well add it.

To check a bit, shift the number x to the right, then bitwise AND it:

bit = (number >> x) & 1;

That will put the value of bit x into the variable bit.

## Changing the nth bit to x

Setting the nth bit to either 1 or 0 can be achieved with the following:

number ^= (-x ^ number) & (1 << n);

Bit n will be set if x is 1, and cleared if x is 0.

## Count set bits in an integer

1.

/\* Function to get no of set bits in binary

   representation of passed binary no. \*/

int countSetBits(unsigned int n)

{

  unsigned int count = 0;

  while(n)

  {

    count += n & 1;

    n >>= 1;

  }

  return count;

}

/\* Program to test function countSetBits \*/

int main()

{

    int i = 9;

    printf("%d", countSetBits(i));

    getchar();

    return 0;

}

2.

#include<stdio.h>

/\* Function to get no of set bits in binary

   representation of passed binary no. \*/

int countSetBits(int n)

{

    unsigned int count = 0;

    while (n)

    {

      n &= (n-1) ;

      count++;

    }

    return count;

}

/\* Program to test function countSetBits \*/

int main()

{

    int i = 9;

    printf("%d", countSetBits(i));

    getchar();

    return 0;

}