ADVANCED NOTES ON C LANGUAGE

(FOR ADVANCED SYSTEM PROGRAMMERS/INSTRUCTORS/TRAINERS)
Based on ANSI C-ISO/IEC 9899:1999

Authored By:

Peter Chacko
Netdiox Computing Systems
peter@nediox.com
www.netdiox.com

Keywords/index terms: C interview questions FAQ, C Programming, volatile, pointers, GNU C , compiler, kernel,training, Linux..

TABLE OF CONTENTS

1.INTRODUCTION

Why this notes?

It is generally observed that few people know how much they don't know in C, that cause them waste enormous amount of time debugging their C code for fixing core dumps, data corruption, abnormal program behavior etc which they could have avoided with better coding in C, in the first phase. Then people move to computer programming from different background, with non-uniform skills in fundamental Computer Science that is vital to be an efficient C Programmer. Lastly C depends on the underlying processor architecture, ABI(Application binary interface specification), Language tools, Operating System to get it's job done for a program on a given hardware- which cannot be explained in a C book completely. Hence people think that C is vast topic to master, which is not quite right, once you have the complete background to be a C Programmer. This notes try to throw some lights on these loose ends, and motivates the reader to gather more information on different domains , as it progresses.

Before you proceed always remember that C is a short language to finish reading out , but a vast language to understand very very deep and at the bottom you will see its interaction with other computer science domains , like operating systems , compilers , CPU architecture etc. So always be patient and be liberal in terms of how much computer science you want to know more, to be a master in C as a system programmer.

This article doesn't cover C language domains in its entirety, this is just a complementary notes on existing books on C. You also need to refer your C manual for the specific features/implementation of the C standard. That's not covered here to avoid duplication. Use this document as a directive to help you understand your weak areas in C.

Please understand that this article is not meant for those who don't have a system programming background.

2.DATA TYPE, CONVERSIONS AND EXPRESSIONS

A data type of an object indicates the number of bytes it takes to store it , as you know. If it is signed type, the most significant bit of the most significant byte is used to store that information. But when it comes to arrays and functions and pointers, data type plays some role.

Consider the following array,

int ar[10];

Here type of array is array of 10 integers.

How do we represent that information.? Let me give you more background to motivate you to understand this concept further.

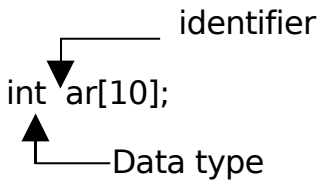
Consider the following declaration,

int a ;

This is an integer declaration, based on the grammar

datatype identifier;

When we declare an array of 10 integers, we follow the same rule



But instead of int[10] arr; we did int arr[10], that is how a data type construct and it's identifier bound together in a declaration of an array(or functions) .That is why it is called "derived type".

Now consider an example of type defining an array of 10 integers.

We do

typedef int ARR[10];

and

ARR ar;

Now 'ar' means an array of 10 integers.

Here we only followed that special rule of derived type to typedef. Many experienced C programmers find it confused when they have to typedef an array type, as they lack the above mentioned understanding of how arrays are "typified".

Similarly function declarations and pointer declarations. (identifier comes in between that different parts of data type.)

Hence we have 2 types of declaration

1st is of the form

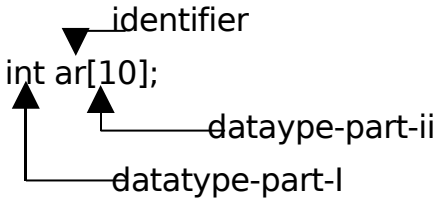
data-type identifier;

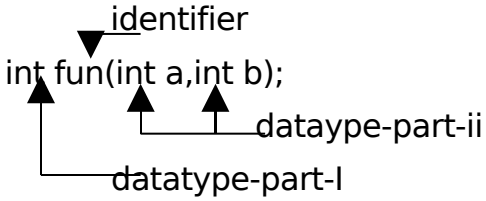
eg: int a;

2nd type is

datatype-part-I identifier dataype-part-ii

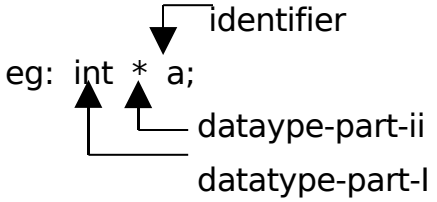
eg:





3rd type is of the form,

data-type-part-I data-type-part-ii identifier;



You can use this information when you work with complex declarations in your projects. Please think over these topics and understand the concepts well.

2.1 CONVERSION /DATA TRANSFORMATION

Again we restrict our discussion to basic data types. Reader is advised to refer the C standard to see the details of other specific data types for his/her specific tasks.

Consider the following,

char A=128;

int B=A;

what is the value stored in ‘B’?

These are the kind of issues that will cause many obscene bug in the form of data corruption.

Here when we store 128 to a char variable, we set the sign bit as 128 , means 10000000 in binary. Now when this is stored in to an integer variable, what is stored in B now is

11111111 11111111 11111111 10000000

as char->int assignment involves an implicit promotion with sign-bit preservation.

So when you print B as signed quantity you will get -128. you now know what is happening.

Now another example

unsigned int A=2;

int B=-1;

if (A>B)

{……..}

else

{ .....}

In this case the expression (A>B)would evaluate false as B gets converted to an unsigned integer which is now

11111111 11111111 11111111 11111111,

(which will be read as -1 without conversion. As negative numbers are stored as 2's complement method)

This conversion is caused by mixing signed and unsigned in an expression.

But most compilers doesn't promote those operands which are part of a sibling sub expression of it's parent expression having bigger data types invoked. ANSI is silent on this topic.

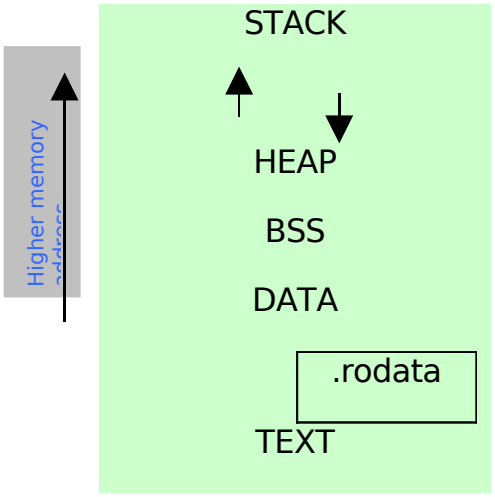
The following operators are special.

&&, ||, ,, ? : etc

All operators are evaluated left to right and there is a sequence point (ie, side effect- safe) after the evaluation of the first operand. (This means that (i++ && i) cause i++ be fully evaluated, applied and then the second operand is used)

3. POINTERS

A beginner or experienced, sometimes or other finds that pointer handling in C could be easier. One reason is that many miss the underlying memory architecture. Consider a typical implementation, as shown by the following logical diagram.



When a program is built(compilation, assembler invocation, and linking) it is stored in a file with a special format. One such format is ELF(Executable and Linking Format)in Unix like systems. It has the necessary information (in the ELF header and program header) to help the program loader to load and create the process image in memory at runtime, and also do run time linking (for shared libraries linked dynamically).

At the time program is built, storage for all uninitialised data (static or global) are not allocated, but only the total size is noted in program header table of the ELF file. Later when program is being loaded, loader read the metadata from the ELF file and allocate one large chunk (page aligned) of memory to hose all such variables. This section is called BSS(Block Static Storage or Block started by symbol).

Data(storage for all initialized static and global variables) and STACK section should be familiar to you. HEAP is created by dynamic invocation by memory-alloc routines of your program. In addition to this there will be shared memory mappings as well. If you process a file using mmap() system call interface, those regions are also mapped to your program. Now the sum total of all such memory is called the address space of your program. You can only manipulate these sections of memory and some which are not to be written in to. Hence all sorts of pointer problems are due to the following primary reasons.

1. You manipulate memory that is not in your program's address space.
2. You manipulate memory in your address space wrongly(like, writing to a write-protected region etc) or corrupting other variables.

For instance, Consider the following declaration ,

```
int ar[4]; int b=100;
ar[4]=200;
```

Will corrupt the variable b, because you exceeded the array bound.[manipulated memory in your region wrongly..

```
int * a;
*a=100;
```

will cause 100 be written to a memory address ‘0’. You have manipulated a memory region outside your address space. Most architecture doesn’t support de-returning 0 address. So accessing a Null pointer is considered as accessing a region outside the address space of any program

Now what’s an address? Is that a virtual address, physical address or logical address?

The value you see when you apply an & operator to a C variable is actually the program-relative logical address. It has to be processed by segmentation unit(coupled with the value in the CS register) to create a linear address and then processed by the paging unit to create the physical address in your RAM. So these details are not at the control of C, but is at the control of the memory management subsystems of the operating system. But a knowledge in these details help you debug your pointer issues faster, as some one who knows his game. Pointers to pointers are well understood as another layer of this indirection. The compelling reason why we have pointers to pointers primarily is to modify a pointer itself across a function call(by passing the address of the pointer). The function in question need to have a pointer to pointer as the formal parameter.(if you are doctor, you need another doctor-to-doctor to cure you). Other wise same concepts apply.

4.STRINGS IN C

Strings are the main object a C programmer work with, which is a sequence of bytes. String literals are null terminated. Its generally accepted that all library functions that process strings expect the strings as null terminated. Consider the following,

```
char ch = ‘A’;

char *ch1 = “A”;

char *ch2 = “”;

char *ch3 = 0;
```

ch is a character variable, not a string. *ch1* points to a location where the string “A” is stored.(a byte sequence where ascii value of A and then a null character). *ch2* points to null string, but has a valid memory address as it’s value. c*h3* is a NULL pointer. Typically strings in C are stored in a write-protected section called .rodata, just above the text segment. It can be changed with the

compiler option, -writable-strings in some compilers. If its stored in .rodata, you get a segmentation fault when you try to modify these region, through the pointer. Always understand that “any string” is an expression, that has the value of the address where this string is stored in the process image.

Some details about the implementation of arrays in C

Arrays to C programmer is what a surgery to surgeon. If you know C arrays well, you know C. Lets invest some time on that here.

Consider the declaration

int ar[4];

int * p=malloc(4*sizeof(int));

the difference in the above 2 uses are.

1. ‘*ar*’ is not a variable, no memory is allocated to store *‘ar’* itself. (some say, it is a constant pointer, wrongly). *‘ar’* is resolved at program translation time. Total memory allocated is only 4 words. In the second case *‘p’* it self is allocated memory, which is a variable. Total memory is 4+1 words(assuming a pointer takes only 4 bytes).

2. when you apply the ‘&’ operator to *ar*, it gives the same value as *ar* itself. When you apply ‘&’ operator to *p* it gives the address of *p*. Reason is obvious, ‘ar’ has no address because its not program symbol seen by the run time and has no address.

The second difference in semantics of arrays name brings another point. How can operator has no meaning to an array name? *think for few minutes….*

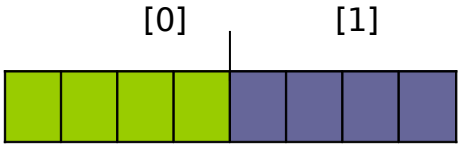
When you apply & operator to a structure or union you set the address of the first by the of the first member(or the shared object in case of union) and when you just refer the variable name of the structure , you mean the entire object. All other variables have the same semantics. But when you just mention

the array by name, you don’t refer it as the entire array, rather an address of the first element. This means that in C,

> “there is no such operation meaningful to entire array”.

You cannot pass or return an entire array to a function or from it. C Creator has decided this special meaning of array name because, all array members of same type, All array members are stored consecutively. (no padding in between), and no reason to treat any member special.

Now consider the array, int ar[2]; which is,



Now as you can see that each element of the array itself is an array of 4 bytes. Hence we can say that this is an array of , array of 4 bytes. Which is nothing but

char ar [2][4];

Hence a multidimensional array is just the perspective exported by the C compiler . It’s informal representation is just a sequence of bytes. when you interpret it as a super object, you get a multi-dimesional array, when you interpret it as super-sub-object, you get a single array. The matrix form is only for your understanding. Another way to describe this is with the statement that C is a row major language. As C is a row major language it has some difference in the following code

int arr[100][100];

1.
```
for(int i=0;i<100; i++)
        for(int j=0;j<100;j++)
                ar[i][j]=i*j;
```
2.
```
for (int j=0;j<100;j++)
        for(int i=0; i<100; i++)
```

```
ar[i][j]=i*j;
```

Here the first version is faster as we access row by row , where as in the second case we access it by column by column. In the first case, hardware cache line stall only at every CACHE-LINE size of bytes, whereas in the second case hardware cache line stall happens on every memory access. This is one example to show that you need to know the underlying hardware architecture , to be an effective C programmer. Mastering C is not just learning the C language.

We can see that arrays and pointers are similar. We have pointers to array as well.

```
int (*ar)[10];
```

Declare a poiner to an array, each element is one large array itself. You can store to it like this

```
int b [2][10];

ar=b;
```

Or

```
ar=malloc(2* sizeof(int (*)[10]); // But all pointers are of same size anyway//

*ar=b[0]; (or b[1]);
```

now if you do **ar, it means the first integer stored in the element array. When you do ++a, you are jumping off by 1 array itself.

See you have never declared a pointer to a pointer, but you de-referenced it as a pointer-to pointer(using **ar) , this is the beauty(or wildness?) of pointer to an array.

Many think that the following assignment is valid given the declarations

```
int ** p;

int ar[2][4];

p=ar;
```

Here compiler throws error. She is right. You are actually storing an object of type pointer to an array of 4 integers, to a plain pointer to pointer to an integer. You should declare a pointer to an array of 4 integers and then store it. You should do this in the function argument declarations as well.

5.1 STRUCTURES/UNIONS

structure tag, member name, structure name all fall into different name spaces. That is you can have

```
struct NewStruct{

        int NewStruct;

        }NewStruct
```

without a compilation error.

As memory reads happens at the boundaries of every word(typical 4 bytes for a 32 bit system, 8 bytes or 64 bits system), if any multi-byte element crosses this boundary CPU has to issue more than 1 read/write cycle to access the memory, causing performance issues. Hence compiler add padding bytes to cause members aligned on these boundaries compiler never pads in the beginning. (that is address of the structure and the address of the first element of the structure is always same.)

All bit fields representation is implementation dependant.

6. HARDWARE CACHE LINES AND C STRUCTURES

When you declare variables in a structure, always declare related variables (meaning, the variables you always access together) as close as possible. That way, one variable access always cause other variables also found in the underlying cache line. When ever you declare large object, always align that on a cache line boundary(typically 32 bytes or 64 bytes depends on the architecture). If you have an important member variable in a structure, which is split across multiple cache lines, then your program will show lower performance. Important kernel data structures in the linux kernel carefully structure its member variables to exploit this mechanism. All frequently accessed variables in a structure should always be moved to the front of the structure storage space.

CPU has the registers EBP, ESP ,EIP and the STACK segment at its disposal to implement control transfer.

When a program is executing, it has to be in some function. And that function's frame pointer is stored in EBP and it's stack pointer is stored in ESP. And the next instruction to be executed is stored in EIP. When we move to the new function, that new function also has to use these registers for the same purpose. Hence we need a way to save & restore these values in between the calls.

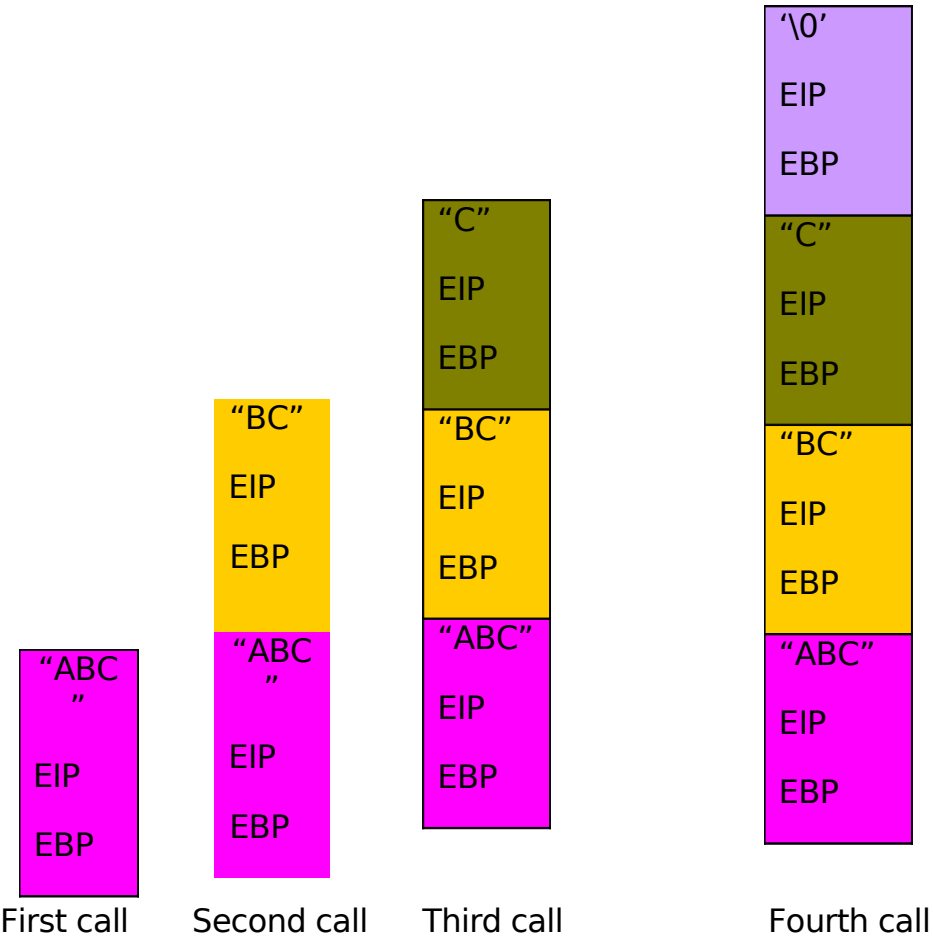
First, all arguments to the call are passed to the stack(or registers based on the implementation). C is a right pusher. (It pushes right most argument first(after it's evaluation). Then the return address is saved on the stack, followed by the current frame pointer . Now what is left to be saved is the current ESP. fortunately current ESP is the new EBP. Hence ESP is just saved in the new EBP after the call. Hence CPU execute a instruction and we are in the new function. Old ESP becomes the new EBP. Old ESP becomes the new EBP. Old ESP will be modified by the new ESP values(We already save the content in EBP). EIP will be used by the instruction streams of the new function and we do our job in the new function. When the function is done, EBP is stored back to ESP. Then previous EBP is popped. Then what is left in the stack frame is saved EIP which is also popped to EIP. And the CPU go to the saved instruction, which is one right after the function call.

Recursion is the property by which a function can call by itself. The only difference between this iterative invocation is that Stack frames are not removed until the function start doing the stack unwinding. Assume that a function that prints a string in reverse order. Here is the function

```
void strReverse(char* st)
{       if(!st) return;
        if(!*st) return;
        strReverse(st + 1 );
        printf("%c", *str);
```

And assume also that we call this function with the string “ABC”

Then the stack frame will be (address decrease as Go up in most of the architecures)



As shown on the fourth call function gets a null character and start unwinding, each stack frame see a different argument as shown. Hence the printf function prints the character in reverse. The important point to understand is that, all code after the recursive call is executed only after the stack –unwinding process begins.

8.VOLATILE VARIABLES IN C

An advanced discussion in C will not be complete without talking about volatile variables. Please refer the paper , “Synchronization/Locking in the Linux kernel on x86 architecture” (published by the the same author, Peter Chacko) to see a detailed coverage

9. APPENDICES

9.1 GNU C COMPILER EXTENSIONS

C language is a portable assembly, but when you use GNU extensions it becomes unportable. But to make the optimum use of the underlying hardware, you have to have a non-portable code in the lower most layer of the abstraction.. And it will add your strength as a kernel developer/System programmer if you understand the important GNU extensions. Following are a partial list of extensions. (For full list, please refer a GNU C manual)

A. Like in ANSI C 99(it refers as a flexible array member of size[1]), GNU C allows a variable length object by having a last array element of size zero.

This is very useful network programming tool when you want to send a message of variable size. Assume that you have a function that fill the variable message data to a structure. The following code does the trick utilizing this mechanism.

```
struct message {

  unsigned int length;

  char msg-buf[0];

};


struct message *NewMsg = (struct message *)

  malloc (sizeof (struct message) + CurrentMsgLength);

NewMsg->length = CurrentMsgLength;
```

Here, you declare the message as a structure having a zero-length array, and depends on the size of your current message (denoted by CurrentMsgLength), you allocate enough memory at the end of the structure. As a consequence of this special use, you cannot have an array of this structure.

B. **Case ranges**

You can specify multiple, consecutive case values in a single case label, like the following.

case *10* ... 20:

which is equivalent to 11 separate case labels.

C . Attributes of a function

These extensions are helpful in passing enough information the compiler, to optimize the code better. This keyword is followed by an attribute specification

inside double parentheses. The following are some useful attributes (please refer the GNU manual for a full list). no_return(indicates whether the function is not to return) , pure(indicates side-effects-free function), always_inline(instruct the compiler to in-line the function),no_inline, deprecated(will cause a warning if the function is called), nonull(to cause a compiler error if non-null arguments are used).

Eg: The following declaration,

```
:       extern void *

 your_strcpy(void *dest, const void *src, unsigned  len)

        __attribute__((nonnull (1, 2)));
```

Will cause a compiler error if you invoke the function with null pointers for the arguments first and second. If nonull is used with no arguments, all arguments are checked against NULL .

E. Attribute syntax for variables

The following are the important attributes for variables

Aligned(to specify the alignment requirements), packed(to specify that only the minimum alignment is required. (1-bit for bitfields, 1-bye for other objects). Which means that all are packed together !!, transparent union(to specify that some variables are actually a union in disguise and to avoid type checking or optimizations based on that). For instance,

Here is a structure in which the field `b` is packed, so that it immediately follows `a` without any padding bytes, using the attribute syntax for variables,

```
struct tag

{

  char a;

  int b __attribute__ ((packed));
```

```
}
```

Now the above structure occupy only 5 bytes, not 8 !!

You can also apply these attributes to a structure or union type as well, as if you use attribute syntax declarations for the all variables.

F. Thread local storage :

To specify a thread specific storage for variables, use _thread specifier, like the following example:

_thread int I; will cause the variable I hosted in a storage that's specific to the calling thread.

G. Some built-in functions for atomic operations :

You can use the family of the following functions , to implement memory access operations atomically(for example as the basic building block of a synchronization primitives. These family of built-ins issue enough "lock" prefixed instructions to realize atomic access to the shared memory objects. (Not yet stable though in many architectures, today). (Please refer a GNU manual for details)

`_sync_add_and_fetch (`*type* `*ptr,` *type* `value, ...) or _sync_fetch_and_add() style`

`__sync_sub_and_fetch (`*type* `*ptr,` *type* `value, ...) or __sync_fetch_and_sub() style`

`__sync_or_and_fetch (`*type* `*ptr,` *type* `value, ...) or __sync_fetch_and_or() style`

`__sync_and_and_fetch (`*type* `*ptr,` *type* `value, ...) or _sync_fetch_and_and() style`

`__sync_xor_and_fetch(type *ptr, type value,…) or __sync_fetch_and_xor() style.`

H. Built in function to avoid the cache-miss latency.

void **__builtin_prefetch** (*const void *addr, ...*)

This function cause the memory object pointed to by the addr are in the cache line, after the execution this function. 2 optional arguments can be passed. The first one specify whether it is a read (value 0) or write(value 1). Second argument specify how local it is (meaning how to replicate it in all hierarchies of the caches). 0 means global(should be there in the outer most cache) and 3 being local(should be kept only in the on-chip CPU cache) and 2 means in-between.

I. Miscellenous GC features used by the kernel developers:

likely(0, unlikely() macros are very much used by kernel code to pass hints to the compiler for the branch prediction(unlikely(0) cause to avoid prefetching the code that follows, to instruction cache.). You can refer kernel sources to see examples. Inline functions are also heavily used by the kernel code. New structure initializer syntax of c99 is another GCC extensions you can find in kernel code in many files.

Invoking assembly code from C is pretty simple, once you understand the basics.

asm(“nop \t\n”); will cause a “nop” assembly instruction executed from your C function. You can also invoke multiple assembly instructions , as multiple, string encoded instructions as shown below. (This implement exit system call library.)

```
asm("movl  $1,%eax\t\n”
     “xor %ebx,%ebx\t\n”
     “int $0x80 \t\n”);
```

```
we are moving 1 to eax as the syscall number of exit is 1. Then
clearing the ebx(as we are not to return from the control
path), and then making the software interrupt 128 to trap into
the kernel.
```

You can also use your C expressions as part of the operands in your assembly code, using extended inline assembly. When you make some registers clobbered, you need to tell the GCC about that..You use extended in-line assembly for this. It has the following form,

asm (assembler template

: output operands constraints (optional)

: input operands constraints (optional)

: list of clobbered registers (optinal)

);

following section shows some examples of how to use these constraints. (Please take a look at various Linux kernel sources/header files to find rich examples in action).

A. example for specific register constrains

Now let's take a look at how to specify individual registers as constraints for the operands. In the following example, the cpuid instruction takes the input in the %eax register and gives output in four registers: %eax, %ebx, %ecx, %edx. CPUID gets it's input from "option"in the eax register, as cpuid expects it to. The a, b, c, and d constraints are used to collect the results

```
int  main() {

int var1,var2,var3,var4,option;

asm ("cpuid"

               : "=a" (var1),

                 "=b" (var2),

                 "=c" (var3),

                 "=d" (var4)

               : "a" (option));

}
```

from the registers EAX, EBX,ECX,EDX to the variables var1,var2,var3,var4 respectively.

B. memory constraints

Typically atomic_inc() dec functions use this in the kernel as we want a memory-to-memory operations in this case. Example follows.

Use of memory operand constraint

Consider the following atomic increment operation:

myLock;

```
asm __volatile__(

                "lock; incl %0"

                :"=m" (myLock)

                :"m" (myLock));
```

Here the memory constraint is necessary to get the desired behaviour as otherwise myLock wouldn't be consistent.

C . Matching operand constraints :

Consider a typical libc implementation of read system call.

```
int read (int fd, void *dataBuf, size len)
{
  long ret;

  __asm__ __volatile__ ("int $0x80"
                      : "=a" (ret)
                      : "0" (SYS_READ), "b" ((long) fd),
                        "c" ((long) dataBuf), "d" ((long)len):
"bx");
```

Above code specify that ret value of the invocation to be stored in EAX, first input is on the matching 0^{th} output register(EAX itself)(we assume that SYS_READ contains the value of the syscall number of the read system call), and the other input values are stored in the EBX, ECX, and EDX registers. And gcc is also instructed to note that ebx will be clobbered. it invokes interrupt 128 and go to kernel mode with all input values, and output the result into EAX.

This is not an interview note. But it is our interest that right candidates are picked during the interview process. Many candidates prepare from ‘C FAQs’ or ‘C for tests’ like notes. The right candidate for the job may score less as his knowledge comes from his C experience, not the incorrect C questions that interviewers see in the net . Here is just a partial list.

1. Can we call main function from main itself?

   How many times you do you need to do this? The behavior depends on how the “hosted behavior” is defined for the specific implementation. C is silent here. And also if he does know what is the specific behavior , that is not a great skill. Invocation of the main is done , normally by the program interpreter(stored in the .interp section of the ELF binary file). The initial stack layout/main program arguments are laid out at this time by the runtime linker/loader tools which all are implementation dependent. Hence when you successfully see that you can call main recursively, it doesn’t mean that its universally implemented.

2. int i=1;

   printf(“%d%d”,i++,++i);

   what is the output?

   This depends on the candidate’s knowledge that

   a) C -compiler is a right pusher. b) there is sequence point after every argument evaluation. And that how arguments are pushed to the stack.

   This is a good question to compiler writers , not to a C programmer, as a conservative C programmer never use side effect operators like this.

3. Can a function return more than one value?

   This is stupid interview question called by many in-experienced C programmers . A function only return one value. Even though you return C pointers that

points to a location containing multiple values, what you return is a single value, pointer.

4. Never ask any C question whose behavior depends on the underlying operating system, libraries, compilers or the underlying architecture for a specific answer.

5. If I do malloc(1) , does that mean malloc only return 1 bytes?

   This is another semi-legal question floated in the net(and dutifully followed by many great interviewers)

   It is true in most OS that malloc will make the underlying system call (like brk() in UNIX), which allocates in chunks of 4k multiple of pages, which will be part of the requesting program's address space, which they can access, without causing the core dump. That doesn't mean that we should re-interpret the meaning of malloc. Its just the way OS allocation does allocation which can be modifies, without changing C, crossing beyond the 1 byte, is semi-begun. malloc libraries can be re-implemented at any time and this is not at the control of C language.

6. Some companies ask questions related to the specifics of library functions, which is not uniformly implemented. Some mix questions related to threads, which are also not yet standardized. When you evaluate a C programmer, always check on his understanding of operators, pointers, and his ability in implementing efficient data structures for the right purpose. C language is for efficiency, and evaluate C programmer for his efficient programming, rather than some details which are not used in 99% of the projects. Check his knowledge in mixed mode arithmetic/operator conversions and his skills in writing great C software having impeccable algorithmic content.

Peter Chacko has been working on system/networking/storage systems development since 1994. His career spans Hard-real time/embedded systems, OS kernels like Unix and Linux, distributed storage and networking systems, virtualization, data security and related computer science disciplines. He had been working for companies like Bellcore (Bell labs spin-off), HP, IBM, US West communications and a couple of startups as a consultant in the United states for 6+ years. He holds a Masters degree in Computer science &applications(MCA), Bachelors in Physics. He is currently the founder & CTO of Bangalore-based cloudStorage startup(Sciendix data systems pvt.ltd). He also run NetDiox computing systems as a not-for-profit research center-cum-educational center on system and networking software alongside. Prior to this he was working for calsoftlabs heading all the R&D activities of storage networking business unit. His linked-in profile is at http://www.linkedin.com/pub/peter-chacko/b/608/608