January 1, 2011



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.

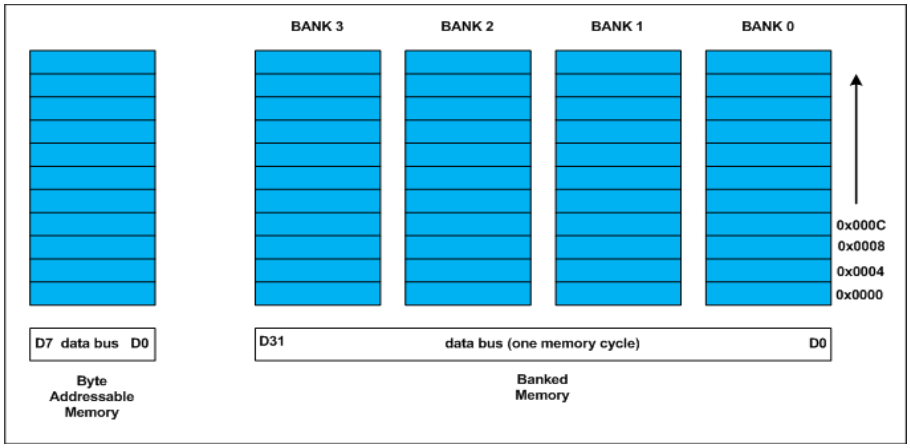
**Data Alignment:**

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.

Interview Experiences
Advanced Data Structures
Dynamic Programming
Greedy Algorithms
Backtracking
Pattern Searching
Divide & Conquer
Graph
Mathematical Algorithms
Recursion
Java



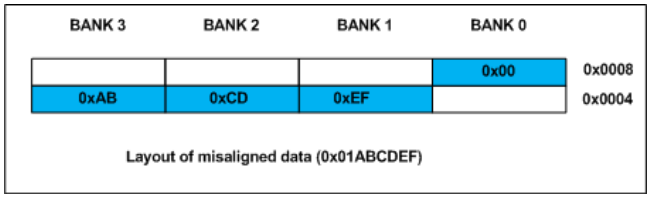
All permutations of a given string
Memory Layout of C Programs
Understanding “extern” keyword in C




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 0×0000), 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*

Median of two sorted arrays
Tree traversal without recursion and without stack!
Structure Member Alignment, Padding and Data Packing
Intersection point of two Linked Lists
Lowest Common Ancestor in a BST.
Check if a binary tree is BST or not
Sorted Linked List to Balanced BST



Balasubramanian on Expression Evaluation
iceman_w on Given an array of of size n and a number k, finds all elements that appear more than n/k times
alexchao on Sort elements by frequency | Set 2
Nikhil Agrawal on Program to count leaf nodes in a binary tree
Nikhil Agrawal on Program to count leaf nodes in a binary tree
Nikhil Agrawal on Program to count leaf nodes in a binary tree
Niks on Dynamic Programming | Set 10 ( 0-1 Knapsack Problem)
Niks on Dynamic Programming | Set 10 ( 0-1 Knapsack Problem)

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 0×0000 for easy calculations. If the structc_t occupies 20 (0×14) bytes as we calculated, the second structc_t array element (indexed at 1) will be at 0×0000 + 0×0014 = 0×0014. It is the start address of index 1 element of array. The double member of this structc_t will be allocated on 0×0014 + 0×1 + 0×7 = 0x001C (decimal 28) which is not multiple of 8 and conflicting with the alignment requirements of double. As we mentioned on the top, the alignment requirement of double is 8 bytes.

Inorder to avoid such misalignment, compiler will introduce alignment requirement to every structure. It will be as that of the largest member of the structure. In our case alignment of structa_t is 2, structb_t is 4 and structc_t is 8. If we need nested structures, the size of largest inner structure will be the alignment of immediate larger structure.

In structc_t of the above program, there will be padding of 4 bytes after int member to make the structure size multiple of its alignment. Thus the sizeof (structc_t) is 24 bytes. It guarantees correct alignment even in arrays. You can cross check.

**structure D - How to Reduce Padding?**

By now, it may be clear that padding is unavoidable. There is a way to minimize padding. The programmer should declare the structure members in their increasing/decreasing order of size. An example is structd_t given in our code, whose size is 16 bytes in lieu of 24 bytes of structc_t.

**What is structure packing?**

Some times it is mandatory to avoid padded bytes among the members of structure. For example, reading contents of ELF file header or BMP or JPEG file header. We need to define a structure similar to that of the header layout and map it. However, care should be exercised in accessing such members. Typically reading byte by byte is an option to avoid misaligned exceptions. There will be hit on performance.

Most of the compilers provide non standard extensions to switch off the default padding like pragmas or command line switches. Consult the documentation of respective compiler for more details.

**Pointer Mishaps:**

There is possibility of potential error while dealing with pointer arithmetic. For example, dereferencing a generic pointer (void *) as shown below can cause misaligned exception,

```
// Deferencing a generic pointer (not safe)
// There is no guarantee that pGeneric is integer aligned
*(int *)pGeneric;
```

It is possible above type of code in programming. If the pointer *pGeneric* is not aligned as per the requirements of casted data type, there is possibility to get misaligned exception.

Infact few processors will not have the last two bits of address decoding, and there is no way to access *misaligned* address. The processor generates misaligned exception, if the programmer tries to access such address.

**A note on malloc() returned pointer**

The pointer returned by malloc() is *void* *. It can be converted to any data type as per the need of programmer. The implementer of malloc() should return a pointer that is aligned to maximum size of primitive data types (those defined by compiler). It is usually aligned to 8 byte boundary on 32 bit machines.

**Object File Alignment, Section Alignment, Page Alignment**

These are specific to operating system implementer, compiler writers and are beyond the scope of this article. Infact, I don't have much information.

**General Questions:**

1. Is alignment applied for stack?

Yes. The stack is also memory. The system programmer should load the stack pointer with a memory address that is properly aligned. Generally, the processor won't check stack alignment, it is the programmer's responsibility to ensure proper alignment of stack memory. Any misalignment will cause run time surprises.

For example, if the processor word length is 32 bit, stack pointer also should be aligned to be multiple of 4 bytes.

2. If *char* data is placed in a bank other bank 0, it will be placed on wrong data lines during memory read. How the processor handles *char* type?

Usually, the processor will recognize the data type based on instruction (e.g. LDRB on ARM processor). Depending on the bank it is stored, the processor shifts the byte onto least significant data lines.

3. When arguments passed on stack, are they subjected to alignment?

Yes. The compiler helps programmer in making proper alignment. For example, if a 16-bit value is pushed onto a 32-bit wide stack, the value is automatically padded with zeros out to 32 bits. Consider the following program.

```
void argument_alignment_check( char c1, char c2 )
{
   // Considering downward stack
   // (on upward stack the output will be negative)
   printf("Displacement %d\n", (int)&c2 - (int)&c1);
}
```

The output will be 4 on a 32 bit machine. It is because each character occupies 4 bytes due to alignment requirements.

4. What will happen if we try to access a misaligned data?

It depends on processor architecture. If the access is misaligned, the processor automatically issues sufficient memory read cycles and packs the data properly onto the data bus. The penalty is on performance. Where as few processors will not have last two address lines, which means there is no-way to access odd byte boundary. Every data access must be aligned (4 bytes) properly. A misaligned access is critical exception on such processors. If the exception is ignored, read data will be incorrect and hence the results.

5. Is there any way to query alignment requirements of a data type.

Yes. Compilers provide non standard extensions for such needs. For example, __alignof() in Visual Studio helps in getting the alignment requirements of data type. Read MSDN for details.

6. When memory reading is efficient in reading 4 bytes at a time on 32 bit machine, why should a **double** type be aligned on 8 byte boundary?

It is important to note that most of the processors will have math co-processor, called Floating Point Unit (FPU). Any floating point operation in the code will be translated into FPU instructions. The main processor is nothing to do with floating point execution. All this will be done behind the scenes.

As per standard, double type will occupy 8 bytes. And, every floating point operation performed in FPU will be of 64 bit length. Even float types will be promoted to 64 bit prior to execution.

The 64 bit length of FPU registers forces double type to be allocated on 8 byte boundary. I am assuming (I don't have concrete information) in case of FPU operations, data fetch might be different, I mean the data bus, since it goes to FPU. Hence, the address decoding will be different for double types (which is expected to be on 8 byte boundary). It means, *the address decoding circuits of floating point unit will not have last 3 pins*.

**Answers:**

```
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
```

**Update: 1-May-2013**

It is observed that on latest processors we are getting size of struct_c as 16 bytes. I yet to read relevant documentation. I will update once I got proper information (written to few experts in hardware).

On older processors (AMD Athlon X2) using same set of tools (GCC 4.7) I got struct_c size as 24 bytes. The size depends on how memory banking organized at the hardware level.

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

**You may also like following posts**

1. Implement Your Own sizeof
2. Little and Big Endian Mystery
3. C Language | Set 1
4. Data type of character constants in C and C++
5. The OFFSETOF() macro

## Facebook Comments

3 comments

Add a comment...

**Anitesh Kumar Bhatt** · Sr. Systems Engineer at Siemens

Thank you for the article. I have one observation, and am a nbit confused abou

void CheckAlignment(int c1, char c2);.

In the implementation if I do :
void CheckAlignment(int c1, char c2).
{
cout<<"Padding: "<<(&c1 - (int*)(&c2))<<endl;.
return;
}

I get result : 1.

And, if I do:
void CheckAlignment(int c1, char c2).
... See more

Reply · Like · 29 April at 02:14

**Venkata Ramana Sanaka** · Senior System Software Engineer a

Refresh your understanding about pointer arithmetic.

Let us consider first case,

c1 is int, hence &c1 yields 'const int *'
c2 is char, so &c2 yields 'const char *'

When you do pointer arithmetic, the difference is divided by sizeof
is valid.

The result of difference will be divided by sizeof(int), so the output

Consider second case,

When we cast the pointer to char *, they both still point the same

... See more

Reply · 2 · Like · 1 May at 07:10

**Mervin Jons** · Works at Global Edge Software Ltd

Its wrong ans wil be 4 , 8 , 16 , 16 b'coz it will take the longest data type size a

Reply · 1 · Like · 26 January at 19:52

**Venkata Ramana Sanaka** · Senior System Software Engineer a

I will check. I am also getting 16 bytes on moderen processors. I n

My test environment.
Intel i7 64 bit. Code tested on 32 bit Ubuntu Linux, GCC 4.7 runnir

Reply · Like · 1 May at 07:27

**Venkata Ramana Sanaka** · Senior System Software Engineer a

On AMD64 + Ubuntu 12.10, GCC 4.7, it is the output ..

venki@siri-linux:~/pw/siri$ g++ --std=c++11 -O0 -o sample struct
structure.cpp: In function 'int main()':
structure.cpp:44:56: warning: format '%d' expects argument of ty

```
structure.cpp:45:56: warning: format '%d' expects argument of ty
structure.cpp:46:56: warning: format '%d' expects argument of ty
structure.cpp:47:56: warning: format '%d' expects argument of ty
venki@siri-linux:~/pw/siri$ ./sample
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
venki@siri-linux:~/pw/siri$ uname -a
Linux siri-linux 3.5.0-17-generic #28-Ubuntu SMP Tue Oct 9 19:31:
```

So, it depends on processor architecture and memory organization

Reply · Like · 1 May at 09:20

Facebook social plugin

## 58 comments so far

**RS** says:
April 3, 2013 at 10:15 PM

For me Also size of sizeof(structc_t) = 16 and not 24

Using built-in specs.
COLLECT_GCC=g++
COLLECT_LTO_WRAPPER=/usr/lib/gcc/i686-linux-gnu/4.6/lto-wrapper
Target: i686-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu/Linaro 4.6.3-1ubuntu5' --with-bugurl=file:///usr/share/doc/gcc-4.6/README.Bugs --enable-languages=c,c++,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-4.6 --enable-shared --enable-linker-build-id --with-system-zlib --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --with-gxx-include-dir=/usr/include/c++/4.6 --libdir=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --enable-gnu-unique-object --enable-plugin --enable-objc-gc --enable-targets=all --disable-werror --with-arch-32=i686 --with-tune=generic --enable-checking=release --build=i686-linux-gnu --host=i686-linux-gnu --target=i686-linux-gnu
Thread model: posix
gcc version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)

```
/* Paste your code here (You may delete these lines if not writing code) */
```

Reply

**Venki** says:
May 1, 2013 at 11:09 PM

Please read the update.

Reply

**Fuzzy** says:
March 21, 2013 at 1:47 PM

In case of structure structc_t for character variable you have added a padding of 7 byte though word length is 4 byte. If I think that word length is 8 byte then why you have not added a padding to the integer variable? I have tested the structc_t in my machine (32bit) and it is giving 16 byte. Am I wrong or right?

Reply

**Venki** says:
April 2, 2013 at 10:07 PM

Read the explanation in blue color. Every structure also have alignment requirement.

Please post your compiler flags, processor details, system specs and compiler used.

Reply

**Willa** says:
December 12, 2012 at 8:38 PM

I couldn't refrain from commenting. Well written!

Reply

**Rahul** says:
September 17, 2012 at 7:14 PM

Can you also mention the compiler you used for the particular test cases because as far as I know, the byte alignment will occur in the order of 4 bytes only so for third case i.e. typedef struct structc_tag
{
char c;
double d;
int s;
} structc_t;

16 has to be the total size.
Already tested on two standard compilers --> g++ and Clang++

Reply

**anksanu** says:
November 5, 2012 at 10:49 PM

All the above code is tested on 64-bit system.....

Reply

**Xi** says:
February 25, 2013 at 9:20 PM

I also test onon 64-bit system, this is my answer.
I am using g++-4.6

```
sizeof(structa_t) = 4
sizeof(structb_t) = 8
sizeof(structc_t) = 24
sizeof(structd_t) = 16
```

Reply

**learner** says:
June 15, 2012 at 8:10 PM

Hi,
I don't know where i'm going wrong. I think the o/p should be 32. But i'm getting 28 here in this case.Please explain me.

```
#include <stdio.h>
struct a
{
        char t;      //1 byte+7 padding byte
        double d;    //8 bytes
        short s;     //2 bytes + 2 padding bytes
        char arr[12];//12 bytes 8+8+4+12=32.
};
int main(void)
{
        printf("%d",sizeof(struct a));
        return 0;
}
```

Reply

**learner** says:
June 15, 2012 at 8:14 PM

Ps:-In 32 bit system.

Reply

**Yatal** says:
August 11, 2012 at 4:19 PM

28 byte only
memory block 1 sizeof(char)+3 padding
memory block 2 sizeof(double) <-- first fatch
memory block 3 sizeof(double) <-- secound fatch
memory block 4 sizeof(short) + two byte char array
memory block 5 4
memory block 6 4
memory block 7 2 + 2 padding

total = 4*7

= 28
Yatal Singh Rathod

Reply

**Bhanu Kishore G** says:
March 5, 2013 at 5:51 PM

Yes. It should be 28 only, because there is no need for 7 byte padding after first character. In main there is only one instance of struct a, unless we have array of struct a there is no need to start double address at multiple of 8

Correct me if i am wrong.

Reply

**Venki** says:
July 8, 2012 at 12:03 AM

I don't think 28 will be the output. Please check again. You should get 32 as output. Also, verify that your compiler setting are not optimized to switch off the alignment requirements.

I guess some compilers have limitation of arrays inside structures. Usually in that that case, it boils down to pointer. Check your compiler documentation as well.

Reply

**Amit** says:
July 17, 2012 at 12:04 PM

I executed this code on linux ( 32 bit archi
tecture) with gcc compiler and got 28 as the answer. After reading the wiki article( http://en.wikipedia.org/wiki/Data_structure_alignment ) it was clear to me why it is 28.
There are three main flaws in your understanding of how the memory layout will be:
- On linux double is 4 byte aligned while on windows it is 8 byte aligned. Since we are running it on linux so we should use 4 byte alignment for double
- The char array arr[12] will start immediately after short, there won't be any padding. char has a 1 byte alignment so why should there be a padding.
- The total size of the structure has to be a multiple of largest alignment of it's members. Since we have a double so the total size of structure should be a multiple of 4

```
struct a {
char t;      //1 byte+3 padding byte
double d;    //8 bytes
short s;     //2 bytes
char arr[12];//12 bytes + 2 bytes to make structure size a multiple of 4, tot
 };
```

Reply

**Nishant Kumar** says:
July 22, 2012 at 8:28 PM

I am getting 32 on my 32-bit windows based gcc compiler.

Reply

**Venki** says:
May 12, 2012 at 10:30 PM

@Avi,

I am not sure why first 8 bytes to be left open. May be for some bookkeeping activity. Recommended to consult processor, compiler and your application documentation for correct alignment information.

In your requirement you said, "except char, every other data type is 4 or 8 byte aligned". Usually different data types (primitive) will have different alignment requirement. The above will not be the case. Better get your requirement precisely.

To find the size of structure on 8 byte, do this simple math. Assume the array base address starts on 8 byte boundary, make sure every element is ensured to start on it's alignment. I would recommend to do the sample exercises given in the post. If you are not clear, let me know.

Reply

**avi** says:
May 10, 2012 at 10:53 PM

Hi Venki,
What would be the size of following structure, 40 byte?
struct
{
char branch[4];
long log;

char alpha[2];
short code;
short err;
char time1[8];
char time2[8];
char time3[8];
short length;
};

I also got a set of guidelines:
Items of type char or unsignedchar, or arrays containing items
if these types, are byte aligned.
Structures are word aligned.
All other types of structure members are word alligned.

Reply

**avi** says:
May 10, 2012 at 10:37 PM

Hi Venki,
what would be the size of the following structure : 40?
struct sample1
{
char branch[4];
long log;
char alpha[2];
short code;
short err;
char time1[8];
char time2[8];
char time3[8];
short length;
}
====I also got a set of guidelines==========
---> Items of type char or unsignedchar, or arrays
containing items if these types, are byte alligned.

---> Structures are word alligned.

--->All other types of structure members are word alligned.
==========

Reply

**Venki** says:
May 11, 2012 at 11:25 AM

Let us see the constraints first.

1. char, unsigned char or it's aggregate are byte aligned.
2. Structures and it's members (except char) are word aligned.

Assuming word size as 4 bytes. Let us analyze the above structure.

On total we need 48 bytes, as struct also needs to be word aligned.

```
struct sample1
{
    char branch[4]; // 4 bytes
    long log;       // 4 bytes
    char alpha[2];  // 2 bytes
                    // 2 bytes padding, as short is word word aligned
    short code;     // 2 bytes
                    // 2 bytes padding, as next short to be word aligned
    short err;      // 2 bytes
    char time1[8];  // 8 bytes
    char time2[8];  // 8 bytes
    char time3[8];  // 8 bytes
                    // 2 byte padding
    short length;   // 2 bytes
                    // 2 byte padding
}
```

Reply

**avi** says:
May 11, 2012 at 8:54 PM

Thank You Venki. But why you didn't add 4 bytes padding before log is not clear to me.

avi

Reply

**Venki** says:
May 12, 2012 at 9:47 AM

The rule of thumb is, given that the base address being aligned properly, does all elements aligned naturally? If not introduce padding.

In the above case, assume that the structure base address is 4 byte (as it is given so) aligned. Then the element 'branch' is 4 byte long, and next element 'log' which is type long (assumed 4 bytes), should start on 4 byte boundary. It is satisfied, so no padding is needed.

Reply

**avi** says:
May 12, 2012 at 7:43 PM

Back again Venki.
As my concept getting clear I find there's 1 more constraint which I ignored earlier.
It is said to leave first 8 bytes of the buffer and start mapping the structure from 9th byte of the buffer.
Does it indicate the word size is 8 bytes here. If it is so then after alignment structure would be like this ..
word size assumed 8 bytes

```
struct sample1
{
char branch[4]; // 4 bytes
// 4 bytes padding, as long is word aligned
long log; // 4 bytes

char alpha[2]; // 2 bytes
// 6 bytes padding, as short is word word aligned
short code; // 2 bytes
// 6 bytes padding, as next short to be word aligned
short err; // 2 bytes

char time1[8]; // 8 bytes
char time2[8]; // 8 bytes
char time3[8]; // 8 bytes
// 6 bytes
short length; // 2 bytes

}
```

Is my assumption about word size and structure alignment correct now?

**meer** says:
March 7, 2013 at 2:51 PM

@venki,
i have checked the address of branch n log, it shows if branch starts at 20th location, then log starts at 28th location. that means 4 bytes are padded after branch

**Rahul** says:
June 23, 2012 at 5:19 PM

Hi Venki,

I am little confused here.

Why does 2 bytes of padding are added here, when next member 'short code' is also 2 bytes long.

```
char alpha[2]; // 2 bytes
// 2 bytes padding, as short is word word aligned
```

If its word word aligned , then in strcuta_t after char c , shouldn't 3 bytes be added for padding, why Only 1 byte is added over there...?

```
/* Paste your code here (You may delete these lines if not writing
```

Reply

**praveen** says:
April 27, 2012 at 4:19 PM

i learnt many things from this artical...Thanx a lot..:)

```
/* Paste your code here (You may delete these lines if not writing code) */
```

Reply

**c_learner** says:
April 14, 2012 at 5:51 PM

Hi Venki,

As per the below program for "struct c", I am getting out as 16 instead of 24. Can you please help?

```
#include

struct c
{
char a;
double b;
int c;
};

int main()
{
printf("Sizeof double %d int %d\n", sizeof(double), sizeof(int));
printf("Sizeof struct_c %d\n", sizeof(struct c));
}
```

```
[user@machine ~]# ./a.out
Sizeof double 8 int 4
Sizeof struct_c 16

[user@machine ~]# uname -r
2.6.9-22.EL

[user@machine ~]# gcc --version
gcc (GCC) 3.4.4 20050721 (Red Hat 3.4.4-2)
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

[user@machine4 ~]# cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 15
model : 2
model name : Intel(R) Pentium(R) 4 CPU 2.80GHz
stepping : 9
cpu MHz : 2793.004
cache size : 512 KB
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 2
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
sse2 ss ht tm pbe cid xtpr
bogomips : 5521.40
```

Reply

**Venki** says:
April 15, 2012 at 6:19 PM

It is interesting, it seems you are using older compiler on P4. Although I didn't understand all of the HW specs here, let us do the following experiment to rule any optimization.

Declare an array of "struct c", use atleast one of it's elements or pass it to a function (by pointer - I mean the address of array to be passed, not value) defined in another file. Let me know the result. Make sure to turn off the optimization.

Reply

**c_learner** says:
April 14, 2012 at 5:41 PM

Hi Venki,

The below program is giving output as 16 instead of 24. Can you please explain the reason. I am pasting the gcc version and processor information. If you need more info please give me the commands, I will collect and post it

```
[user@machine ~]# ./a.out
Sizeof double 8 int 4
Sizeof struct_c 16

[user@machine ~]# uname -r
2.6.9-22.EL

[user@machine ~]# gcc --version
gcc (GCC) 3.4.4 20050721 (Red Hat 3.4.4-2)
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

[user@machine4 ~]# cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 15
model : 2
model name : Intel(R) Pentium(R) 4 CPU 2.80GHz
stepping : 9
cpu MHz : 2793.004
cache size : 512 KB
fdiv_bug : no
hlt_bug : no
f00f_bug : no
coma_bug : no
fpu : yes
fpu_exception : yes
cpuid level : 2
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe cid xtpr
bogomips : 5521.40

```c
#include <stdio.h>

struct c
{
    char a;
    double b;
    int c;
};

int main()
{
    printf("Sizeof double %d int %d\n", sizeof(double), sizeof(int));
    printf("Sizeof struct_c %d\n", sizeof(struct c));
}
```

Reply

**codinglearner** says:
March 27, 2012 at 4:43 PM

```c
#include <stdio.h>
struct u
{
 union v
 {
  int i;
  int j;
 }a[10];
 int b[5];
 char d;
 float f ;
 }w;

int main()
{
 printf("%u",sizeof(w));
 return 0;
}
```

plzz xplain the result...???

Reply

**Venki** says:
April 15, 2012 at 1:27 AM

@codinglearner, Union and Struct follow same alignment principles. Alignment is for access data types, not for those used to aggregate them. See the following comments,

```c
struct u
{
    union v
    {
        int i; // 4 bytes
        int j; // 4 bytes
    }a[10];     // Overall 4 * 10 = 40 bytes
    int b[5];  // 4 * 5 = 20 bytes
    char d;    // 1 byte followed by 3 byte padding
    float f ;  // 4 bytes
}w; // 40 + 20 + 1 + 3 + 4 = 68 on 32 bit machine
```

Reply

**Tarun** says:
November 29, 2011 at 9:45 AM

Hi

I have big confusion after reading through this article.
1. A 32 bit system,
has 32 data lines,
any given address in the system would point to a word = 32 bits
2. A 64 bit system,
has 64 data lines,

any given address in the system would point to a word = 64 bits

Now, you said memory is byte addressable. So does this discussion here pertains to 8 bit systems only??

Reply

**Venki** says:
November 29, 2011 at 12:45 PM

@Tarun, even prior to 32 bit systems, we had 16 bit machines, they too addressing memory at byte level. Note that there are processors which having data width of 16 bits, and yet address bus width of 24 bits. They still access memory byte wise.

Irrespective of address bus width and data bus width, many processors address memory as byte addressable for backward compatibility. Also, there are few processors (like in DSP or high speed industrial automation) that won't allow byte level access at all.

Reply

**Daya** says:
October 17, 2011 at 6:47 PM

Thanks a tonne bro. My life's better from this moment

Reply

**Krishs** says:
October 13, 2011 at 12:22 PM

Excellent article!!! clarified many doubts in mind. keep it up. Thanks!!!

Reply

**Venki** says:
September 23, 2011 at 1:20 PM

Another related article can be found here,

Reply

**Guest** says:
September 2, 2011 at 12:25 AM

Excellent article.

Would "long long" on 32bit machine have a 8byte alignment ? Should not be, as even 4byte alignment it correct (both would need 2 cycles).

Reply

**Venki** says:
September 22, 2011 at 7:14 PM

Good question. I depends on compiler the way it reads 8 byte variable. On my machine, I got **long long** size as 8 bytes and its alignment as 8 bytes.

In GCC compiler, we have few compiler extensions like

```
int __attribute__((vector_size(8))) vector_special_variable;
```

However the implementation of these extensions are compiler and processor dependent. For example few processors provide instruction for block read/write which compiler can make use.

Reply

**sam** says:
August 6, 2011 at 6:12 PM

explain me plz why there will be padding of 7 bytes after the first element of the structc_t.
i am getting confused as i am thinking that there should be padding of only 3 bytes after the char element, because after that 3 bytes padding the address for the double element will still be multiple of 8.
and one more thing i want to ask..

"double variable will be allocated on 8 byte boundary"
what does this thing means? i knw thats a silly one.. bt still plz let me knw..

Reply

**Venki** says:
August 29, 2011 at 10:35 PM

FAQ 6 clarifies your question. Also read the other comments.

Reply

**Gilco** says:
June 23, 2011 at 5:14 PM

Great explanations!!!

Can someone please explain how come for the following structure I wrote:

```
struct list_head
{
    struct list_head * next;
    struct list_head * prev;
};

struct myFriend
{
    char name[10];
    unsigned double weight;
    unsigned double height;
    struct list_head list;  //embedding the list component
};
```

I'm working on x86 so sizeof(address) is 4 bytes.
I got size of 28??

but I calculated on my own and got: 30!!

here is my calculations:
10*sizeof(char)+(2)padding+sizeof(unsigned double)+sizeof(unsigned double)+2*sizeof(address)+(2)padding = 30 Bytes

Thanks for the help guys!

Reply

**Venki** says:
August 3, 2011 at 10:22 PM

@Gilco, sorry for the delay, I missed to observe the comment.

Here it is...
What should be alignment of 'myFriend'? It should be multiple of largest element in that array. The largest element of 'myFriend' can be either **double** or **list_head** both takes 8 bytes each. The alignment of 'myFriend' should be 8 bytes, means any object of 'myFriend' should start on 8 byte boundary.

Now, how much should be the padding after *char* array? It is determined by the alignment requirements of next element which is 8 bytes. Hence after the *char* array there will be padding of 6 bytes. Overall

10 + 6 (padding) + 8 + 8 + 8 = 40 byte

I am surprised how you got 28 on computer and 30 on mind calculations.

Reply

**Venki** says:
August 3, 2011 at 10:31 PM

I think you left *unsigned double* (?) which is not allowed by compiler. A double can't be unsigned. May be the compiler is considering the 'weight' and 'height' members as *unsigned int*, hence the output as 28 (check yourself).

Reply

**neha2210** says:
March 1, 2013 at 2:32 PM

I think on Linux machine it will be 28 as the double is 4 byte aligned and not 8 byte aligned.

Reply

**Karthick** says:
September 22, 2011 at 11:32 AM

Here is my though on it.. The structure would have

name is a char[10] => internal implementation is char* => 4 not 10...

double => 8
double => 8
list_head* => 2*4 = 8

Total = 24...

Reply

**Venki** says:
September 22, 2011 at 7:00 PM

@Karthick, if char *name[10]* is stored as *char **, where will be the size of name stored? I don't think compiler can change attributes of identifiers.

User wants array semantics where as your suggestion (assumption) using pointer semantics.

Reply

**Venki** says:
June 10, 2011 at 10:28 PM

A related post on bit-fields
http://geeksforgeeks.org/forum/topic/c-structure-size-with-empty-bitfield

Reply

**sharat** says:
January 26, 2011 at 2:40 PM

Need more details for structc:

How did you arrive at padding 7 between Char and double(1st and 2nd parameter of the structure)

Is it because that double has to start at and address which is a multiple of 8(which is size of double) ?

If yes, then 7 is not always true. Consider the below example

if char a(first element) resides at 0x04(which is a valid assumption), then a padding of 7 would make double store at 0xc which is not a multiple of 8 . A padding of 3 would be appropriate here.

Please let me know if my understanding is right. or Am I missing something.

Thanks,
Sharat.

Reply

**Venki** says:
January 26, 2011 at 5:30 PM

@sharat, I miss one exception here. At the start of the article I told to assume every structure allocated at multiple of 4 bytes for easy.

It is not always true as explained in case of structc_t. Every structure type also will have alignment requirement. So in case, if the structure contains double as largest member, such structures will be allocated on 8 byte boundary, not on 4 byte boundary.

I will make required correction.

Reply

**sharat** says:
January 26, 2011 at 11:31 PM

Thanks for the clarifications..

Reply

**Sabya Sachi** says:
August 6, 2011 at 10:39 PM

gcc 4.5.2 gives the size to be 16. Is this compiler dependent???

Reply

**Venki** says:
September 22, 2011 at 7:16 PM

@Sabya Sachi, could you provide more details like processor, OS, what kind of GCC port (code blocks, mingw, etc...).

**sharat** says:
January 26, 2011 at 2:14 PM

Note that a double variable will be allocated on 8 byte boundary on 32 bit machine and requires two memory read cycles

Q) Why is it allocated on a 8 byte boundary, what is the issue with allocating on a 4 byte boundary ? it would still require two memread cycles to read a double.

Thanks in advance,
Sharat.

Reply

**Venki** says:
January 26, 2011 at 5:41 PM

@sharat, This is good question. It was asked by one of my colleague. I will update the necessary changes as FAQ.

It is important to note that every processor will have math co-processor (most of the processors), 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. Hence, every floating point operation performed in FPU will be 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, 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**.

Reply

**sk** says:
January 6, 2011 at 5:17 PM

really ...good stuff.

Reply

**jag** says:
January 5, 2011 at 11:48 AM

There is a possible typo at "structure C – Every structure will also have alignment requirements":
"structc_t needs sizeof(char) + 7 byte padding + sizeof(double) + sizeof(int)"
Struct C is defined as char+double+short so the size expected is 18

Reply

**Venki** says:
January 5, 2011 at 8:05 PM

@Jag, thanks for correction. I will update the post.

The analogy is still valid. We need 24 bytes for structc_t. There will be padding of six bytes at the end of structure to make structure size multiple of 8 bytes. Overall it looks,

**sizeof(char) + 7 (padding) + sizeof(double) + sizeof(short) + 6 (padding) = 1 + 7 + 8 + 2 + 6 = 24.**

I will make the data type of ***s*** as *int* instead of *short*, so that the explanation will be untouched.

Reply

**tej** says:
January 4, 2011 at 9:59 PM

I was looking for such article on Data Alignment. So thanks very much to share the same.

Reply

**Narendra** says:
January 3, 2011 at 3:27 PM

Great Job.

Thanks a lot

Reply

## Comment

Name (*Required*)

Email (*Required*)

Website URI

**Your Comment (Writing code? please paste your code between sourcecode tags)**

```
[sourcecode language="C"]
/* Paste your code here (You may delete these lines if not writing
code) */
[/sourcecode]
```

☒ Notify me of follow up comments via e-mail. You can also subscribe without commenting.


Have Your Say