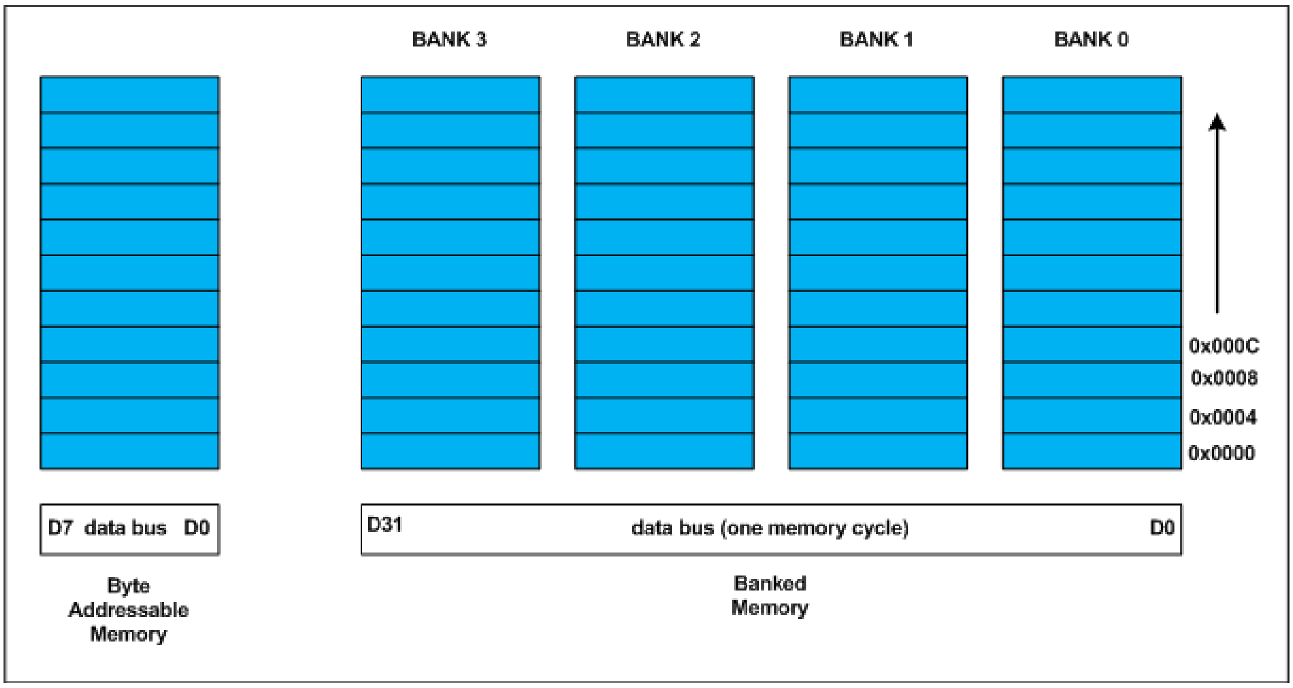
Structure Member Alignment, Padding and Data Packing

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

Predict the output of following program.

```c
#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;
}
```

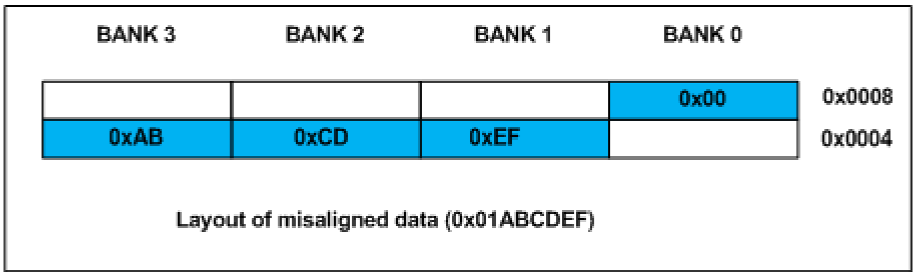
Every data type in C/C++ will have alignment requirement (infact it is mandated by processor architecture, not by language). A processor will have processing word length as that of data bus size. On a 32 bit machine, the processing word size will be 4 bytes.



Historically memory is byte addressable and arranged sequentially. If the memory is arranged as single bank of one byte width, the processor needs to issue 4 memory read cycles to fetch an integer. It is more economical to read all 4 bytes of integer in one memory cycle. To take such advantage, the memory will be arranged as group of 4 banks as shown in the above figure.

The memory addressing still be sequential. If bank 0 occupies an address X, bank 1, bank 2 and bank 3 will be at (X + 1), (X + 2) and (X + 3) addresses. If an integer of 4 bytes is allocated on X address (X is multiple of 4), the processor needs only one memory cycle to read entire integer.

Where as, if the integer is allocated at an address other than multiple of 4, it spans across two rows of the banks as shown in the below figure. Such an integer requires two memory read cycle to fetch the data.



A variable's ***data alignment*** deals with the way the data stored in these banks. For example, the natural alignment of ***int*** on 32-bit machine is 4 bytes. When a data type is naturally aligned, the CPU fetches it in minimum read cycles.

Similarly, the natural alignment of ***short int*** is 2 bytes. It means, a ***short int*** can be stored in bank 0 – bank 1 pair or bank 2 – bank 3 pair. A ***double*** requires 8 bytes, and occupies two rows in the memory banks. Any misalignment of ***double*** will force more than two read cycles to fetch ***double*** data.

Note that a **double** variable will be allocated on 8 byte boundary on 32 bit machine and requires two memory read cycles. On a 64 bit machine, based on number of banks, **double** variable will be allocated on 8 byte boundary and requires only one memory read cycle.

Structure Padding:

In C/C++ a structures are used as data pack. It doesn't provide any data encapsulation or data hiding features (C++ case is an exception due to its semantic similarity with classes).

Because of the alignment requirements of various data types, every member of structure should be naturally aligned. The members of structure allocated sequentially increasing order. Let us analyze each struct declared in the above program.

Output of Above Program:

For the sake of convenience, assume every structure type variable is allocated on 4 byte boundary (say 0x0000), i.e. the base address of structure is multiple of 4 (need not necessary always, see explanation of structc_t).

structure A

The *structa_t* first element is *char* which is one byte aligned, followed by *short int*. short int is 2 byte aligned. If the the short int element is immediately allocated after the char element, it will start at an odd address boundary. The compiler will insert a padding byte after the char to ensure short int will have an address multiple of 2 (i.e. 2 byte aligned). The total size of structa_t will be sizeof(char) + 1 (padding) + sizeof(short), 1 + 1 + 2 = 4 bytes.

structure B

The first member of *structb_t* is short int followed by char. Since char can be on any byte boundary no padding required in between short int and char, on total they occupy 3 bytes. The next member is int. If the int is allocated immediately, it will start at an odd byte boundary. We need 1 byte padding after the char member to make the address of next int member is 4 byte aligned. On total, the *structb_t* requires 2 + 1 + 1 (padding) + 4 = 8 bytes.

structure C – Every structure will also have alignment requirements

Applying same analysis, *structc_t* needs sizeof(char) + 7 byte padding + sizeof(double) + sizeof(int) = 1 + 7 + 8 + 4 = 20 bytes. However, the sizeof(structc_t) will be 24 bytes. It is because, along with structure members, structure type variables will also have natural alignment. Let us understand it by an example. Say, we declared an array of structc_t as shown below

```
structc_t structc_array[3];
```

Assume, the base address of *structc_array* is 0x0000 for easy calculations. If the structc_t occupies 20 (0x14) bytes as we calculated, the second structc_t array element (indexed at 1) will be at 0x0000 + 0x0014 = 0x0014. It is the start address of index 1 element of array. The double member of this structc_t will be allocated on 0x0014 + 0x1 + 0x7 = 0x001C (decimal 28) which is not multiple of 8 and conflicting with the alignment requirements of double. As we mentioned on the top, the alignment requirement of double is 8 bytes.

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

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

structure D – How to Reduce Padding?

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

What is structure packing?

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

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

Pointer Mishaps:

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

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

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

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

A note on malloc() returned pointer

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

Object File Alignment, Section Alignment, Page Alignment

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

General Questions:

1. Is alignment applied for stack?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Answers:

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