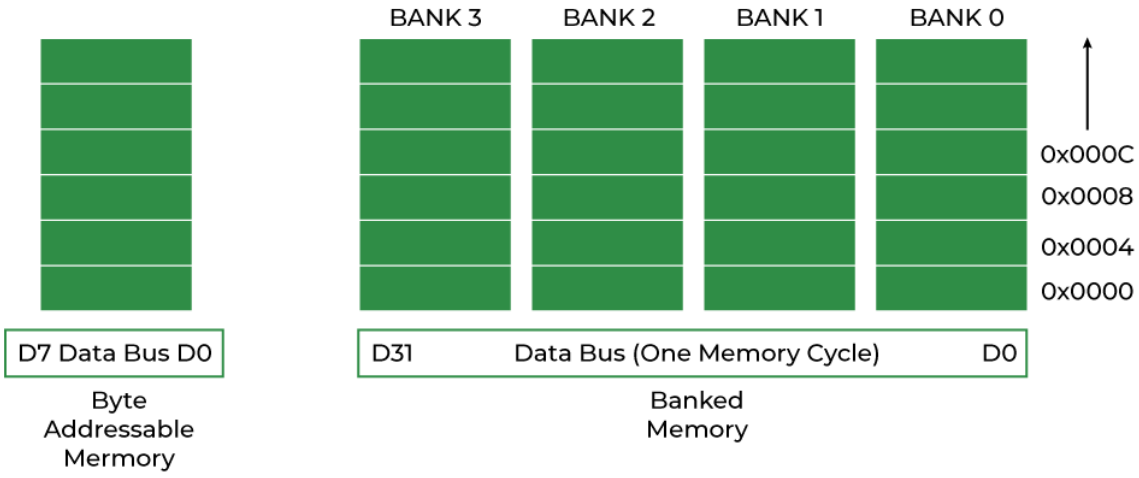
Structure Member Alignment, Padding and

In C, the structures are used as data packs. They don't provide any data encapsulation or data hiding features.

In this article, we will discuss the property of structure padding in C along with data alignment and structure packing.

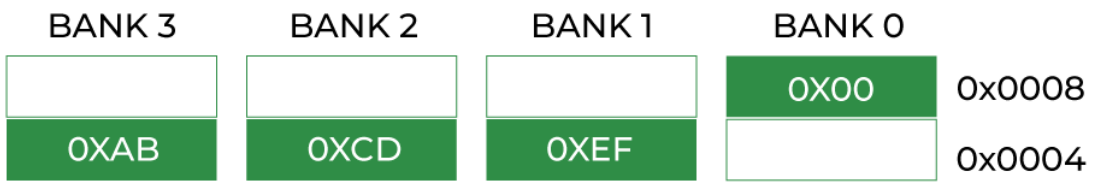
Data Alignment in Memory

Every data type in C will have alignment requirements (in fact 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 a 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 an integer in one memory cycle. To take such advantage, the memory will be arranged as a 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 a multiple of 4), the processor needs only one memory cycle to read the entire integer. Whereas, if the integer is allocated at an address other than a multiple of 4, it spans across two rows of the banks as shown in the below figure. Such an integer requires two memory read cycles to fetch the data.



Layout of misaligned data (0X01ABCDEF)

A variable's ***data alignment*** deals with the way the data is stored in these banks. For example, the natural alignment of ***int*** on a 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 a ***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 an 8-byte boundary on a 32-bit machine and requires two memory read cycles. On a 64-bit machine, based on a number of banks, a **double** variable will be allocated on the 8-byte boundary and requires only one memory read cycle.

Structure Padding in C

Structure padding is the addition of some empty bytes of memory in the structure to naturally align the data members in the memory. It is done to minimize the CPU read cycles to retrieve different data members in the structure.

Try to calculate the size of the following structures:

C

```c
// 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;
```

Calculating the size of each structure by directly adding the size of all the members, we get:

- **Size of Structure A** = Size of (char + short int) = 1 + 2 = **3.**
- **Size of Structure B** = Size of (short int + char + int) = 2 + 1 + 4 = **7.**
- **Size of Structure C** = Size of (char + double + int) = 1 + 8 + 4 = **13.**
- **Size of Structure A** = Size of (double + int + char) = 8 + 4 + 1= **13.**

Now let's confirm the size of these structures using the given C Program:

C

```c
// C Program to demonstrate the structure padding property
#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) = %lu\n", sizeof(structa_t));
    printf("sizeof(structb_t) = %lu\n", sizeof(structb_t));
    printf("sizeof(structc_t) = %lu\n", sizeof(structc_t));
    printf("sizeof(structd_t) = %lu\n", sizeof(structd_t));

    return 0;
}
```

Output

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

As we can see, the size of the structures is different from those we calculated.

> ***This is because of the alignment requirements of various data types, every member of the structure should be naturally aligned. The members of the structure are allocated sequentially in increasing order.***

Let us analyze each struct declared in the above program. For the sake of convenience, assume every structure type variable is allocated on a 4-byte boundary (say 0x0000), i.e. the base address of the structure is multiple of 4 (need not necessarily always, see an 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 bytes aligned. If 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 is required between short int and char, in 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 the next int member 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) is 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 the index 1 element of the 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 conflicts with the alignment requirements of double. As we mentioned at the top, the alignment requirement of double is 8 bytes.

In order to avoid such misalignment, **the compiler introduces alignment requirements 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 the largest inner structure will be the alignment of an immediate larger structure.

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

Structure D

In a similar way, the size of the structure D is :

```
sizeof(double) + sizeof(int) + sizeof(char) + padding(3) = 8 + 4 + 1 +
3 = 16 bytes
```

How to Reduce Structure 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?

Sometimes it is mandatory to avoid padded bytes among the members of the 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 but there will be a hit on performance.

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

In GCC, we can use the following code for structure packing:

```
#pragma pack(1)
```

or

```
struct name {
    ...
}__attribute__((packed));
```

Example of Structure Packing

C

```
// C Program to demonstrate the structure packing
#include <stdio.h>
#pragma pack(1)

// structure A
typedef struct structa_tag {
    char c;
    short int s;
} structa_t;

// structure B
```

```c
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) = %lu\n", sizeof(structa_t));
    printf("sizeof(structb_t) = %lu\n", sizeof(structb_t));
    printf("sizeof(structc_t) = %lu\n", sizeof(structc_t));
    printf("sizeof(structd_t) = %lu\n", sizeof(structd_t));

    return 0;
}
```

Output

```
sizeof(structa_t) = 3
sizeof(structb_t) = 7
sizeof(structc_t) = 13
sizeof(structd_t) = 13
```

FAQs on Structure Padding in C

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, the stack pointer also should be aligned to be a multiple of 4 bytes.

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

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

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

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

C

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

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

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

It depends on the 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. Whereas few processors will not have the last two address lines, which means there is no way to access the odd byte boundary. Every data access must be aligned (4 bytes) properly. Misaligned

access is a 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 the 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 a 32-bit machine, why should a double type be aligned on an 8-byte boundary?

It is important to note that most of the processors will have a 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, the 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 bits prior to execution.

The 64-bit length of FPU registers forces double type to be allocated on an 8-byte boundary. I am assuming (I don't have concrete information) in the 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 are expected to be on an 8-byte boundary). It means *the address decoding circuits of the floating point unit will not have the last 3 pins*.

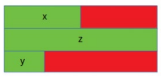
This article is contributed by **Venki**. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

Last Updated : 10 Apr, 2023


Similar Reads

Can We Access Private Data Members of a Class without using a Member or a Friend Function in C++?

How to avoid Structure Padding in C?



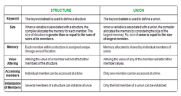
Is sizeof for a struct equal to the sum of sizeof of each member?

Count the number of objects using Static member function

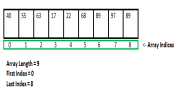
Difference between fundamental data types and derived data types

How to Convert Data From SQL to C Data Types?

Anonymous Union and Structure in C



Difference Between Structure and Union in C



Difference between Structure and Array in C

Previous
C typedef

Next
Flexible Array Members in a structure in C

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Expert

Easy | Normal | Medium | Hard | Expert

Improved By :
stephenblaked,
simmytarika5,
surindertarika1234,
abhishekcpp

Article Tags :
C-Struct-Union-Enum,
C Language

Report Issue

Join our Community

Courses

course-img

103k+ interested Geeks

Master C Programming with Data Structures

Explore

course-img

GeeksforGeeks

A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org

Job-A-Thon For Experienced

GfG Weekly Contest

Offline Classes (Delhi/NCR)

DSA in JAVA/C++

Master System Design

Master CP

Languages

Python

Java

C++

PHP

GoLang

SQL

R Language

Android Tutorial

DSA Concepts

Data Structures

Arrays

Strings

Linked List

Algorithms

Searching

Sorting

Mathematical

Dynamic Programming

DSA Roadmaps

DSA for Beginners

Basic DSA Coding

Problems

Complete Roadmap To Learn DSA

DSA for FrontEnd Developers

DSA with JavaScript

Top 100 DSA Interview Problems

Web Development

HTML

CSS

JavaScript

Bootstrap

ReactJS

AngularJS

NodeJS

Computer Science

GATE CS Notes

Operating Systems

Computer Network

Database Management System

Software Engineering

Digital Logic Design

Engineering Maths

Python

Python Programming Examples

Django Tutorial

Python Projects

Python Tkinter

OpenCV Python Tutorial

Python Interview Question

Data Science & ML

Data Science With Python

Data Science For Beginner

Machine Learning Tutorial

Maths For Machine Learning

Pandas Tutorial

NumPy Tutorial

NLP Tutorial

Deep Learning Tutorial

DevOps

Git

AWS

Docker

Kubernetes

Azure

GCP

Competitive Programming

Top DSA for CP

Top 50 Tree Problems

Top 50 Graph Problems

Top 50 Array Problems

Top 50 String Problems

Top 50 DP Problems

Top 15 Websites for CP

System Design

What is System Design

Monolithic and Distributed SD

Scalability in SD

Databases in SD

High Level Design or HLD

Low Level Design or LLD

Top SD Interview Questions

Interview Corner

Company Wise Preparation

Preparation for SDE

Experienced Interviews

Internship Interviews

Competitive Programming

Aptitude Preparation

GfG School

CBSE Notes for

Class 8

CBSE Notes for Class 9

CBSE Notes for Class 10

CBSE Notes for Class 11

CBSE Notes for Class 12

English Grammar

Commerce

Accountancy

Business Studies

Economics

Management

Income Tax

Finance

UPSC

Polity Notes

Geography Notes

History Notes

Science and Technology Notes

Economics Notes

Important Topics in Ethics

UPSC Previous Year Papers

SSC/ BANKING

SSC CGL Syllabus

SBI PO Syllabus

SBI Clerk Syllabus

IBPS PO Syllabus

IBPS Clerk Syllabus

Aptitude Questions

SSC CGL Practice Papers

Write & Earn

Write an Article

Improve an Article

Pick Topics to Write

Write Interview Experience

Internships