B. Tech. I CSE (Sem-2)
Data Structures
CS102

Data Structures
Introduction

Dr. Rupa G. Mehta
Dr. Dipti P. Rana
Department of Computer Science and Engineering
SVNIT, Surat

Brief Syllabus

- Review and Introduction
- Linear Lists
- Stacks
- Queues
- Sorting and Searching
- Trees
  - Multi way Trees
- Graph algorithms
- Algorithm analysis

Course Outcome

CO1 recognize the **need of different data structures** and understand **its characteristics**.

CO2 **apply different data structures for given problems**.

CO3 **design and analyze different data structures**, sorting and searching techniques.

CO4 **evaluate data structure operations** theoretically and experimentally.

CO5 **give solution for complex engineering problems**.

Text Books

1. Trembley & Sorenson: "An Introduction to Data Structures with Applications", 2/E, TMH, 1991.
2. Tanenbaum & Augenstein: "Data Structures using C and C++", 2/E, Pearson, 2007.
3. Horowitz and Sahani: "Fundamentals of Data Structures in C", 2/E, Silicon Press, 2007.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest: "Introduction to Algorithms",3/E, MIT Press, 2009.
5. Robert L. Kruse, C. L. Tondo and Brence Leung: "Data Structures and Program Design in C", 2/E, Pearson Education, 2001.

Data Structure: Introduction

- Any form of information processing or communication requires that data must be stored in and accessed from either main or secondary memory
  - There are two questions:
    - What do we want to do?
    - How can we do it?

- Data structures
  - Concrete methods for organizing (insert/sort and insert) and accessing (display, modify, delete, search) data in the computer
  - A representation of the logical relationship existing between individual elements of data

- Abstract Data Types
  - Models of the storage and access of information

Core Operations

- Data Structures will have 3 core operations
  - A Way to add data
  - A way to remove data
  - A way to access data
- Details of these operations depend on the data structure
  - List
  - Add at the location (begin/end/a particular position)
  - Access by location (begin/end/a particular position)
  - Remove by location (begin/end/a particular position)
- More operations added depending on what data structure is designed to do

Data Types Vs Data Structures

DATA TYPES	DATA STRUCTURES
Kind or form of a variable which is being used throughout the program with assigned values	The collection of different kinds of data. That entire data can be represented using an object and can be used throughout the entire program
Implementation through data types is a form of abstract implementation	Implementation through data structures is called concrete implementation
Can hold values and not data, so it is data less	Can hold different kind and types of data within one single object
Values can directly be assigned to the data type variables	The data is assigned to the data structure object using some set of algorithms and operations like push, pop and so on.
No problem of time complexity	Time complexity comes into play when working with data structures
Examples: int, float, double	Examples: stacks, queues, tree

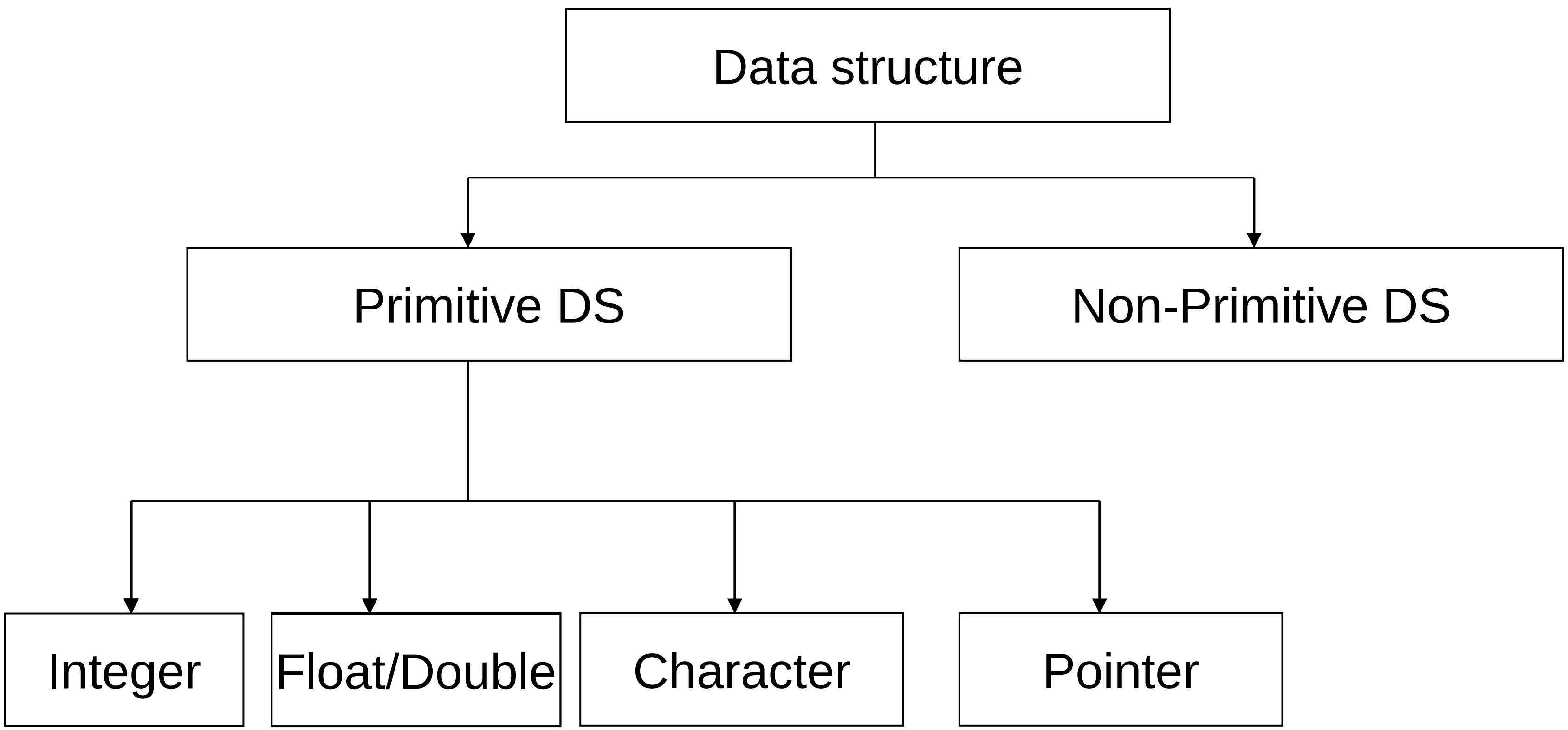
Data Structure Classification

1. Primitive / Non-primitive
   – Basic Data Structures available / Derived from Primitive Data Structures
2. Linear / Non-Linear
   – Maintain a Linear relationship between element
3. Sequential / Non-Sequential
   – Based on an access of data
4. Homogeneous / Heterogeneous
   – Elements are of the same type / Different types
5. Static / Dynamic
   – Memory is allocated at the time of compilation / run-time

Primitive Data Structure

- Basic data structures that are already defined in the language and directly operate upon the machine instructions
- Different representations on different computers
- Used to store only a single value
- The primitive data structures in C (also known as primitive data types) include int, char, float, double, and pointers

Data Structure Classification



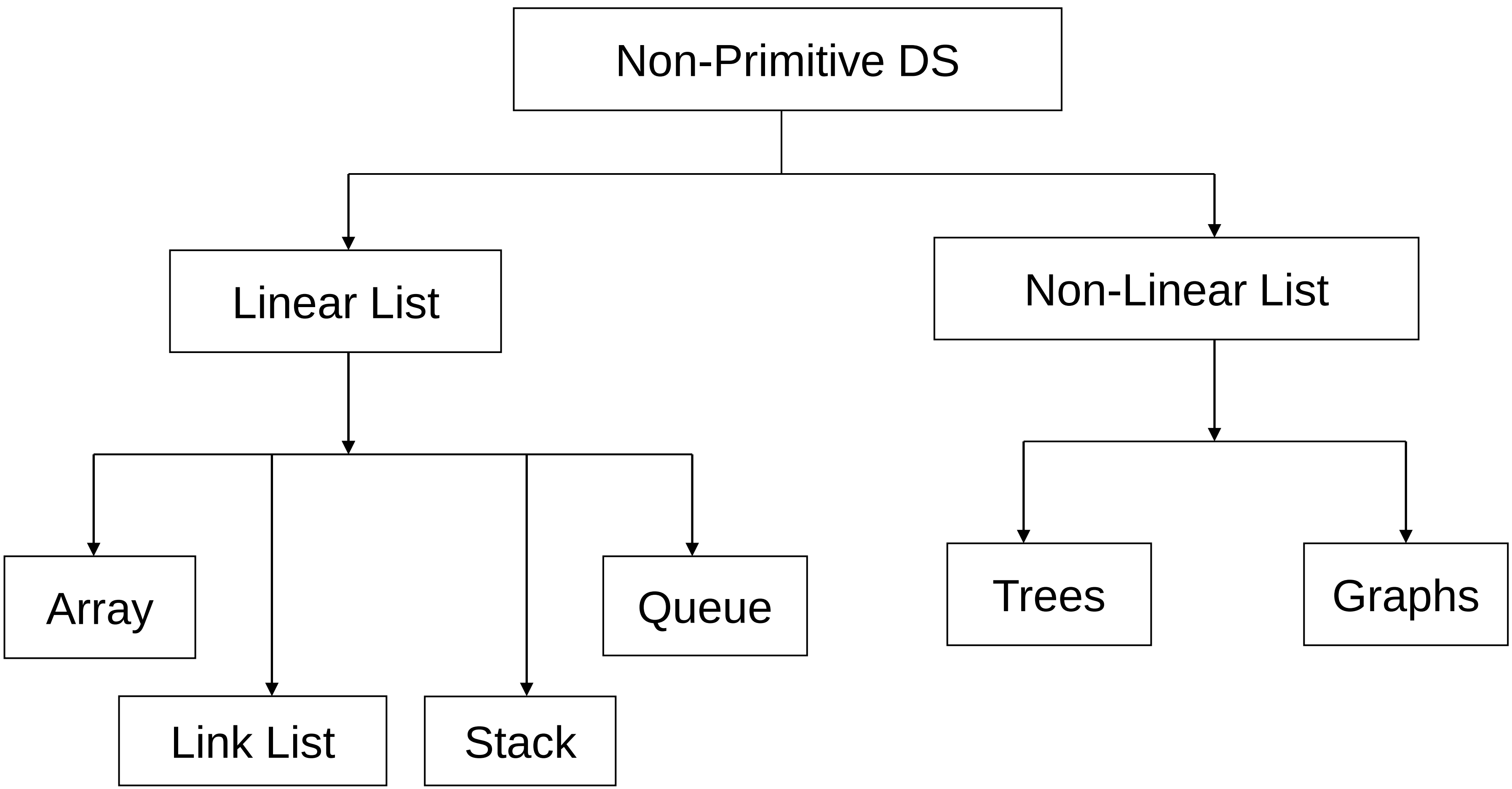
Primitive Data Structure

Type	Typical Bit Width	Typical Range
char	1 byte	-127 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-127 to 127
int	4 bytes	-2147483648 to 2147483647
unsigned int	4 bytes	0 to 4294967295
signed int	4 bytes	-2147483648 to 2147483647
short int	2 bytes	-32768 to 32767
unsigned short int	2 bytes	0 to 65,535
signed short int	2 bytes	-32768 to 32767
long int	8 bytes	-2,147,483,648 to 2,147,483,647
signed long int	8 bytes	same as long int
unsigned long int	8 bytes	0 to 4,294,967,295
long long int	8 bytes	-(2^63) to (2^63)-1
unsigned long long int	8 bytes	0 to 18,446,744,073,709,551,615
float	4 bytes	
double	8 bytes	
long double	12 bytes	
wchar_t	2 or 4 bytes	1 wide character

NonPrimitive Data Structure

- AKA derived data structures
  - As they are derived from primitive ones
- More complicated data structures
- Group of same or different data items with relationship between each data item
- A large number of values can be stored using the non-primitive data structures
- The data stored can also be manipulated using various operations like insertion, deletion, searching, sorting, etc.
- E.g. Arrays, Lists, Stacks, Queues, Trees, Graphs,

Data Structure Classification



Linear and Non-Linear

- **Linear Data Structures**
  - Data are accessed in a sequential manner
    - The elements can be stored in these data structures in any order
  - Eg. Array, Linked List, Stack, Queue, Hashing, etc.
- **Non-Linear Data Structure**
  - Elements are not stored in linear manner
  - The data elements have hierarchical relationship which involves the relationship between the child, parent, and grandparent
  - E.g. Trees, Binary Search Trees, Graphs, Heaps, Tries, Segment Tree etc.

Sequential and Non-Sequential

- **Sequential Data Structure**
  - Storage of data is contiguous
  - E.g. Array
- **Non-Sequential Data Structure**
  - Storage of data is Non contiguous
  - E.g. Linked List

Homogenous and Non-Homogeneous

- **Homogenous Data structure**
  - All the elements are of same type
  - E.g. Array
- **Non- Homogenous Data structure**
  - The elements may or may not be of the same type
  - E.g. Structures

Primitive Data Structures

Integer

- Integer
  - Size 4 Byte
  - Signed Integer and Unsigned Integer

Signed and Unsigned Int

- Signed Int
  - Left most bit (MSB) is used to denote the sign (negative or Positive)
  - The rest of the bits are then used to denote the value normally
  - MSB is used to denote whether it's positive (with a 0) or negative (with a 1)
- A Sign bit of 0 denotes that the number is a *non-negative*
  - May be equal to the decimal zero or a positive number.
- The negative number is stored in 2’s Complement notation

Example

Assuming 8 bit storage :

$01001101_2 = +77_{10}$

(-/+)	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	0	1	1	0	1

Assuming 4 bit storage :

$0101_2 = +5_{10}$

2^3	2^2	2^1	2^0
0	1	0	1

Assuming 4 bit storage :

$1001_2 = -7_{10}$

(-/+)	2^2	2^1	2^0
1	0	0	1

Assuming 4 bit storage :

$0000_2 = +0_{10}$

(-/+)	2^2	2^1	2^0
0	0	0	0

Assuming 4 bit storage :

$1000_2 = -8_{10}$

(-/+)	2^2	2^1	2^0
0	0	0	0

2’s complement

Two’s complement binary	Decimal
0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

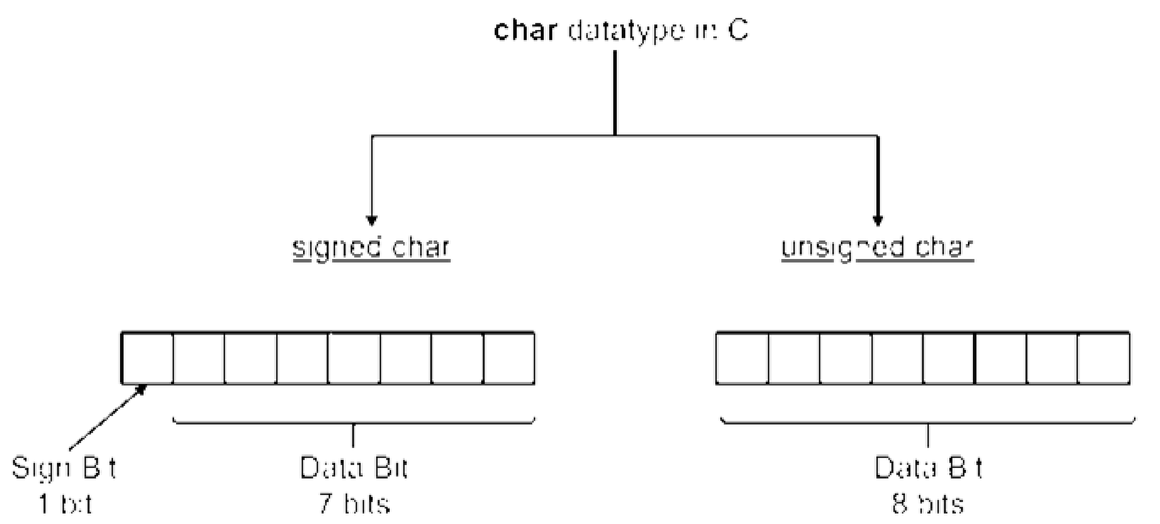
Range of Integer

- For 32 bit (4 byte integer)

- Signed : $-2^{31} – 0 – (2^{31}-1)$
- Unsigned: $0 – (2^{32}-1)$

Character

- **char** is the most basic data type in C.
- Stores a single character and requires a **single byte of memory** in almost all compilers.
- Can be divided into 2 types:
  - signed char
  - unsigned char

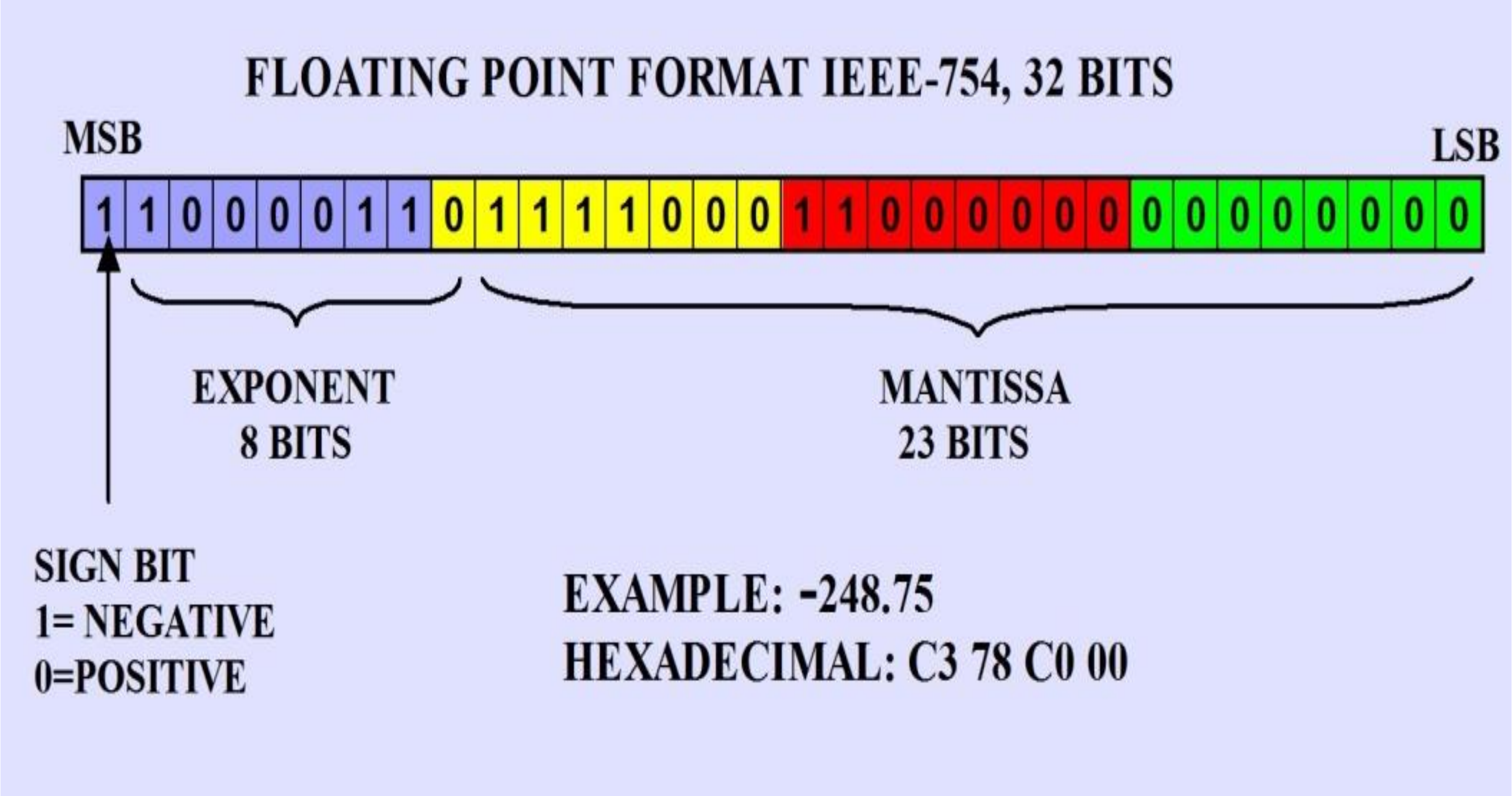


Signed and Unsigned char

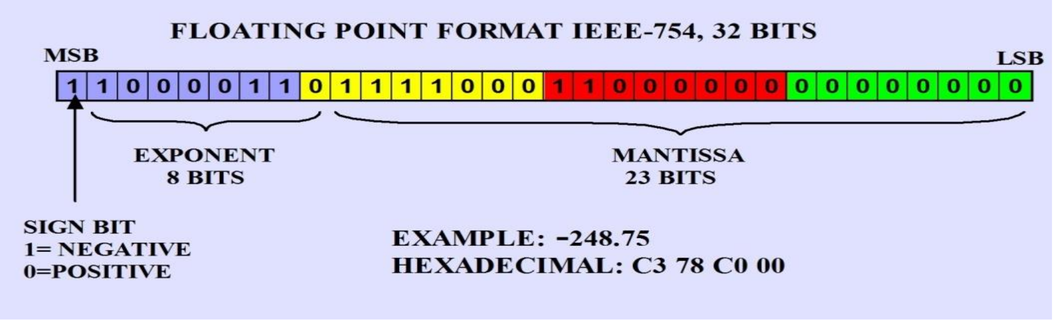
- **Example**
- **signed char c=97**
  - ASCII value 97 will be converted to a character value, i.e. ‘a’ and it will be inserted
  - Range -128 to 127

- **unsigned char = -1 (Initializing with signed value)**
  - ASCII value -1 will be first converted to 255
  - This value will be converted to a character value, i.e. ‘ÿ’ and it will be inserted
  - Range 0 to 255

Floating-point number

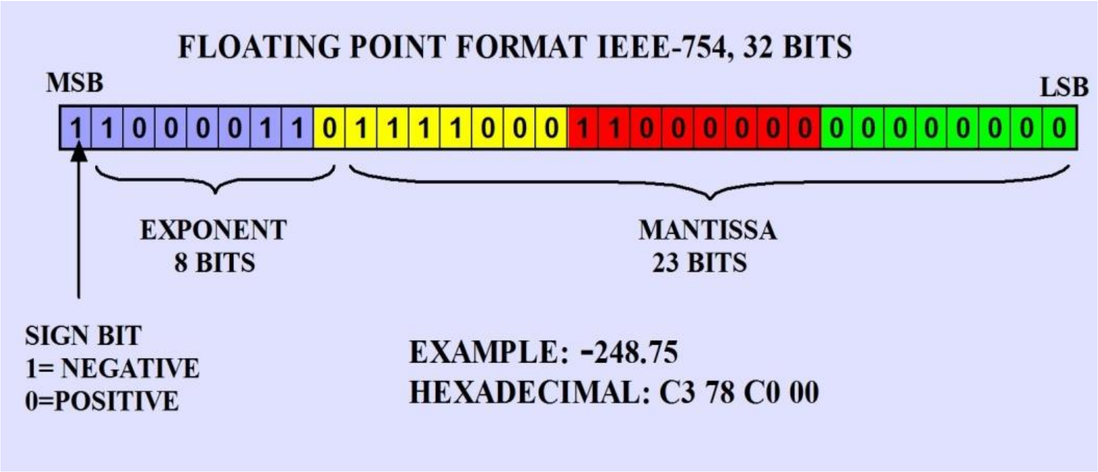


Floating Point Storage



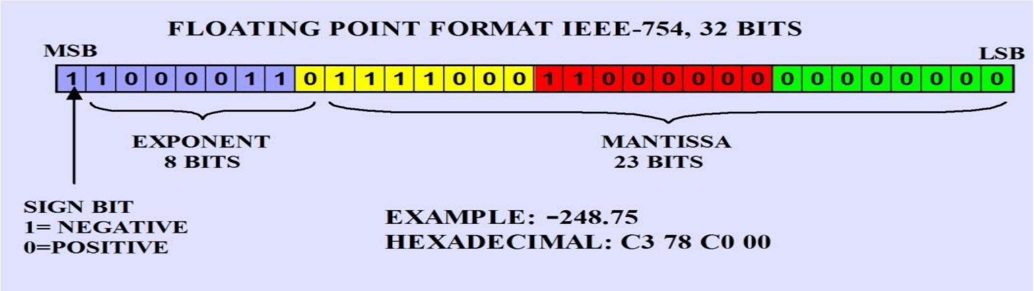
- A typical single-precision 32-bit floating-point memory layout
  - Sign (1 bit)
  - Exponent (biased exponent) (8 bits)
  - Significand(Mantissa) (Normalized Mantissa) (23 bits)
- Sign
  - The high-order bit indicates a sign.
  - 0 indicates a positive value, 1 indicates negative.

Floating Point Storage



- Exponent
  - The next 8 bits are used for the exponent which can be positive or negative,
  - But instead of reserving another sign bit they are encoded by adding *bias=127*
  - Bias : 2^{k-1} -1 where 'k' is the number of bits in exponent field.

Floating Point Storage



- Significand
  - The remaining 23-bits used for the significand (AKA mantissa)
  - Each bit represents a negative power of 2 counting from the left

Floating value 3.14 storage

- **3 = 0011 in binary**
- **The rest, 0.14**
- 0.14 x 2 = 0.28, 0
- 0.28 x 2 = 0.56, 00
- 0.56 x 2 = 1.12, 001
- 0.12 x 2 = 0.24, 0010
- 0.24 x 2 = 0.48, 00100
- 0.48 x 2 = 0.96, 001000
- 0.96 x 2 = 1.92, 0010001
- 0.92 x 2 = 1.84, 00100011
- 0.84 x 2 = 1.68, 001000111
- And so on . . .

Floating value 3.14 storage

- 3.14 = 11.001000111... In binary

- Shift it (normalize it) to adjust Exponent
  - **1.1001000111 →+1.57 * 2^1**
  - **Exponent = +1**

- Biased Exponent = 1 + 127 = 128
  - **0000 00001 + 0111 1111 = 1000 0000**

- Final Value
  - **0 1000 0000 1100 1000 111...**

- Forget the top 1 of the mantissa (which is always supposed to be 1, except for some special values, so it is not stored)

Memory organization for 65.125

- Converting to Binary form,
  - **65 = 1000001**
  - **0.125 = 001**

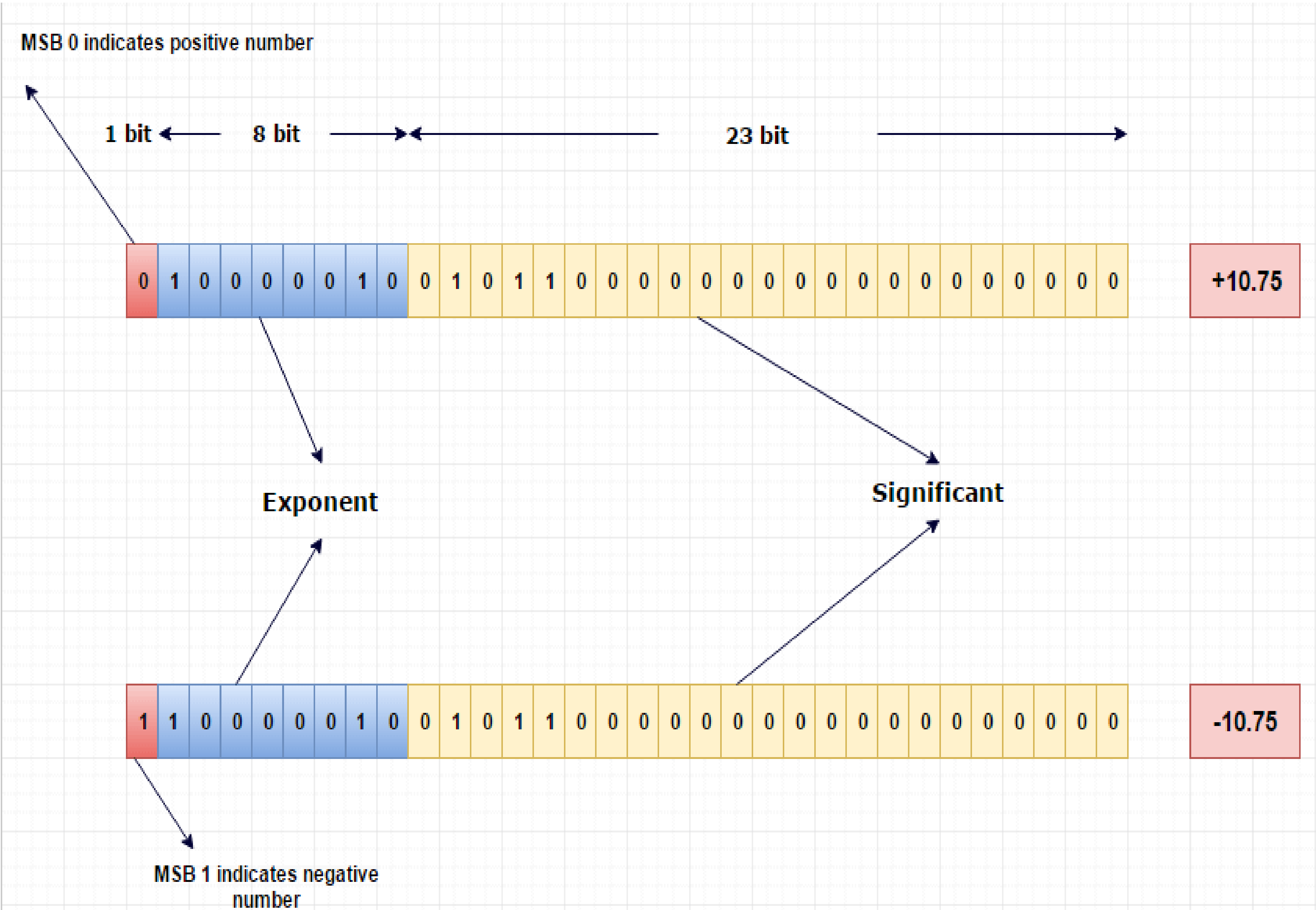
- **65.125 = 1000001.001 = 1.000001001 x 10^6**

- **Normalized Mantissa = 000001001**

- Biased exponent by adding the exponent to 127, = 127 + 6 = 133
- **Biased exponent = 10000101**

- And the **signed bit is 0 (positive)**

- 65.125 will be stored as
- **0 10000101 00000100100000000000000**


MSB 0 indicates positive number
1 bit
8 bit
23 bit
0 1 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
+10.75
Exponent
Significant
1 1 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
-10.75
MSB 1 indicates negative number

Floating Storage for 0.1

- Binary conversion
  - 0.1 *2 0.2   0
  - 0.2 *2 0.4 0
  - 0.4 *2 0.8   0
  - 0.8*2 0.6   1
  - 0.6*2 0.2   1
  - 0.2 *2 0.4 0
  - 0.4 *2 0.8   0
  - 0.8*2 0.6   1
  - 0.6*2 0.2   1
  - **0.000110011……**
  - **1.10011…. * 2^-4**
- Mantissa = 10011….
- Exp = -4+127 = 123 (01111011)

s eeeeeeee mmmmmmmmmmmmmmmmmmmmmmm

0 01111011   10011001100110011001101

Which is not exactly same as 0.1

Advantages of Floating Point Representation

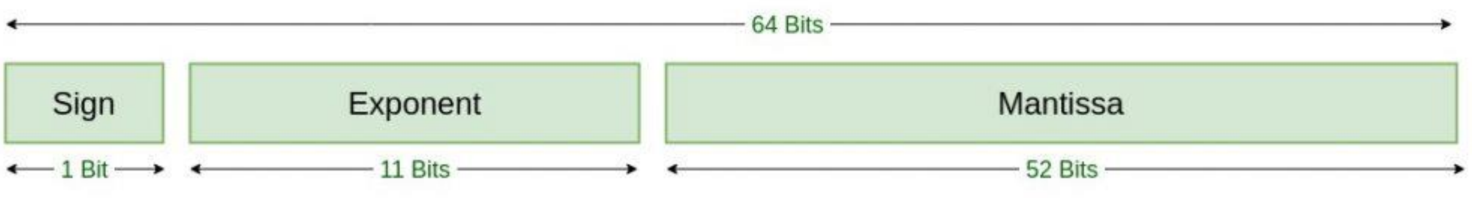
- **Handles very large numbers** – Floating point representation can manage extremely big numbers, making it ideal for computations involving large values.
- **Handles very small numbers** – It is also great at dealing with very tiny numbers, which can be crucial in precise calculations.
- **Supports fractional values** – It is capable of supporting fractional values, which allows for accurate representation of numbers between whole integers.
- **Allows mathematical operations** – This system is suited for mathematical operations, as it maintains precision and can handle a wide range of numbers.
- **Efficient for scientific calculations** – It is highly effective for scientific calculations, where numbers can range from very small to very large, and precision is key.

Disadvantages of Floating Point Representation

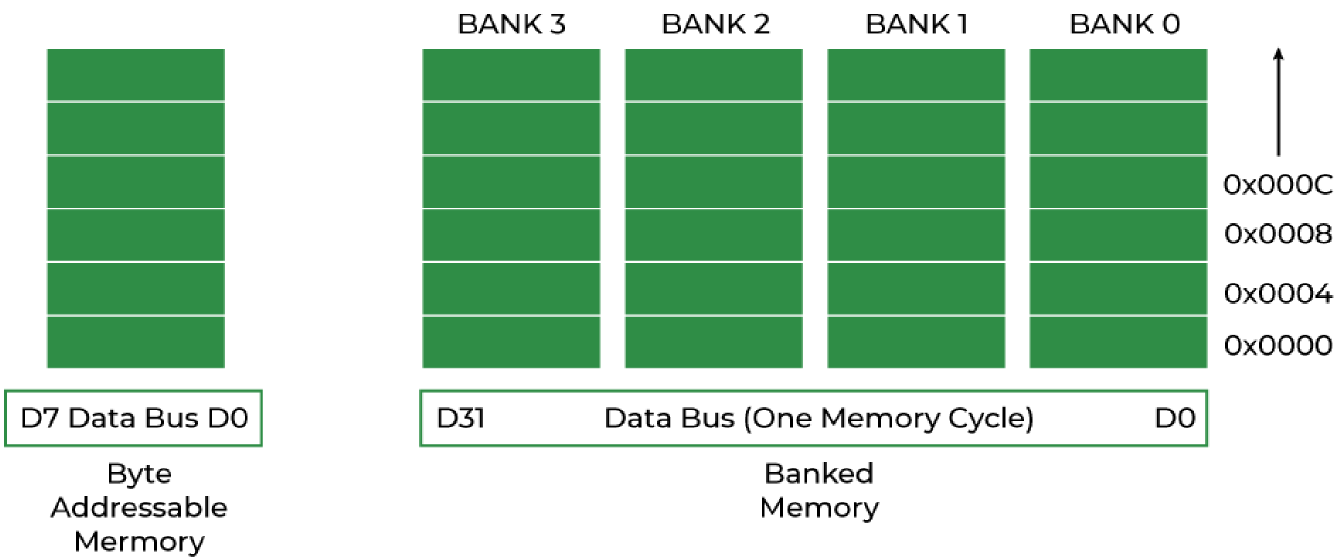
- **Can lead to rounding errors** – Floating point representation can sometimes cause rounding errors. This is because it approximates real numbers, which can lead to minor inaccuracies.
- **Limited precision** – A significant limitation of this method is its restricted precision. It cannot represent all real numbers accurately, especially very large or small ones.
- **Not suitable for exact values** – When exact values are required, floating point representation might not be the best choice. It’s not designed to perfectly represent all numbers, which can lead to inaccuracies.
- **Can cause overflow or underflow** – Overflow or underflow issues can arise with floating point representation. This happens when the numbers are too large or too small to be stored in the available space.
- **Difficult to compare values** – Comparing values can be challenging with floating point representation. Small differences between numbers might not be recognized due to the approximation involved.

Memory organization for Double data type

- The size of the double is 64-bit, out of which:
  - **The most significant bit (MSB)** is used to store the **sign** of the number.
  - The next **11 bits** are used to store the **exponent.**
  - The remaining **52 bits** are used to store the **mantissa.**
-

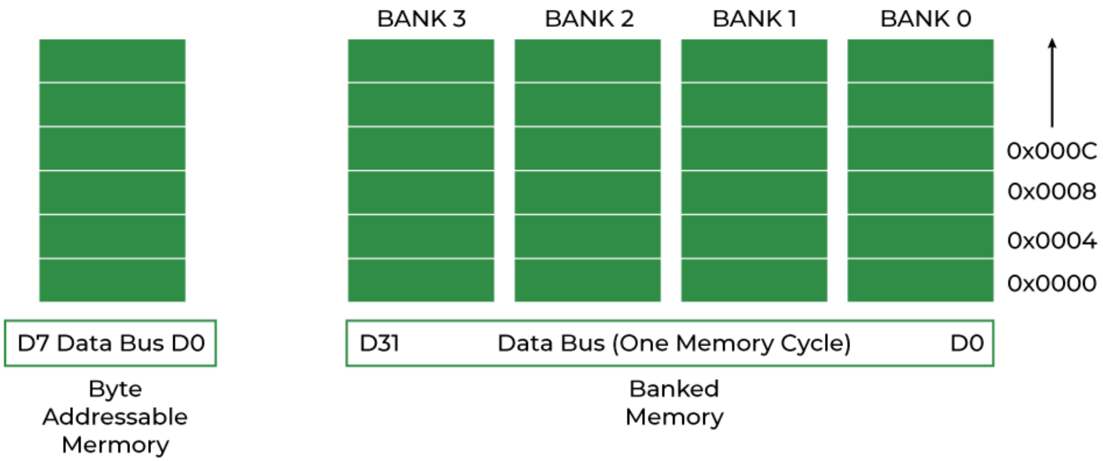


Memory alignment for 32 bit processor (4 Bytes)



- 4 cycles for memory read (One Byte)
- 1 cycle for memory read (4 bytes)

Memory alignment for 32 bit processor (4 Bytes)



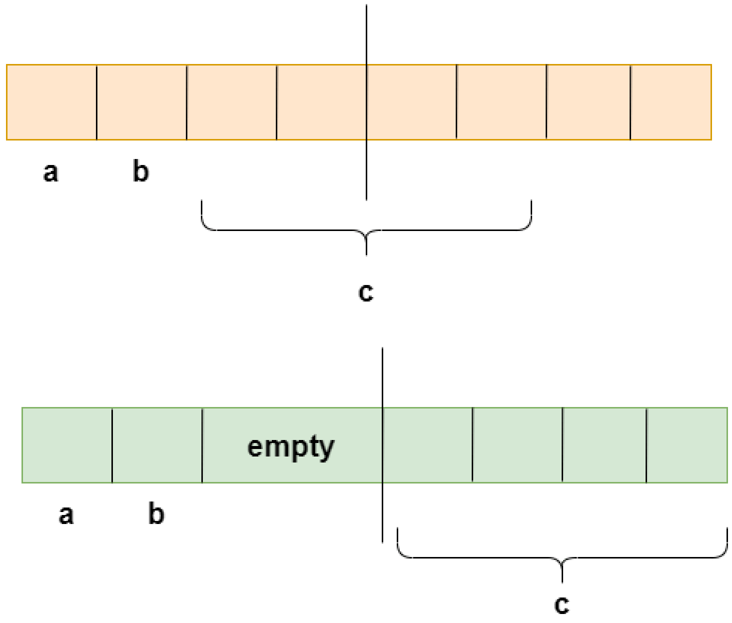
- 4 cycles for memory read (One Byte)
- 1 cyle for memory read (4 bytes)

Layout of misaligned data (0X01ABCDEF)

Structure padding in C

```
struct student
{
   char a;
   char b;
   int c;
} stud1;
```

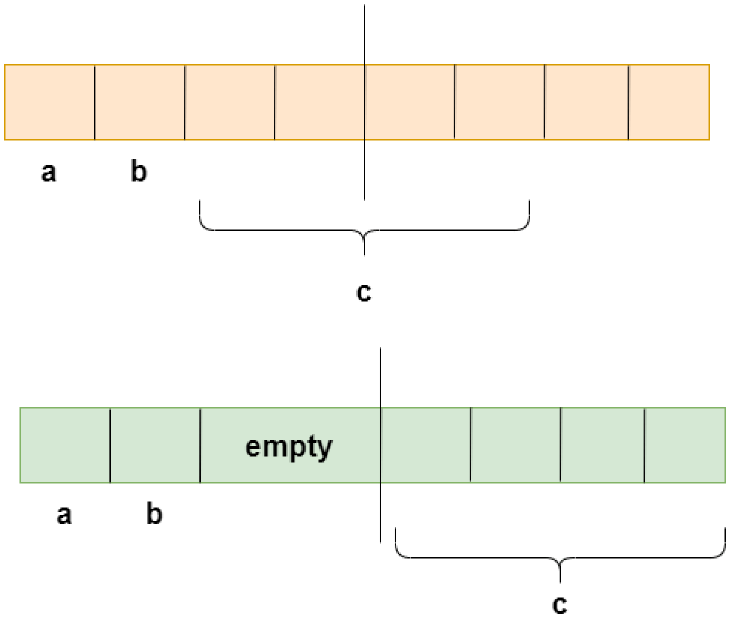
Sizeof(stud) ????



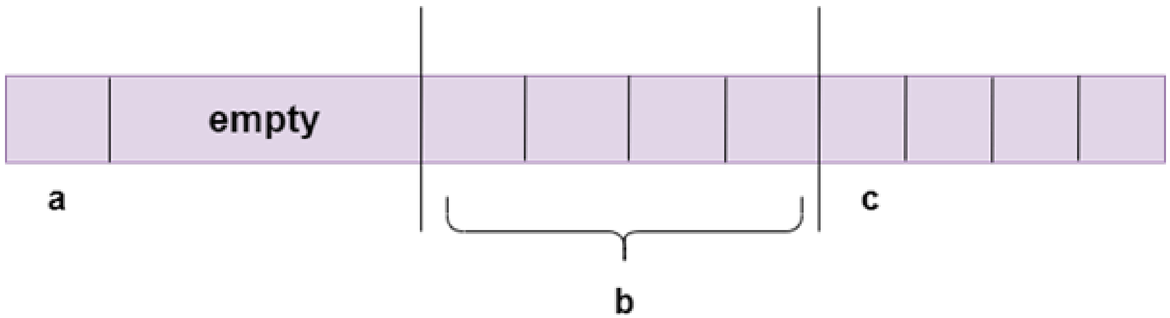
Structure padding in C

```
struct student
{
   char a;
   char b;
   int c;
} stud1;
```

Sizeof(stud) **8** bytes



Structure padding in C (Change the sequence)

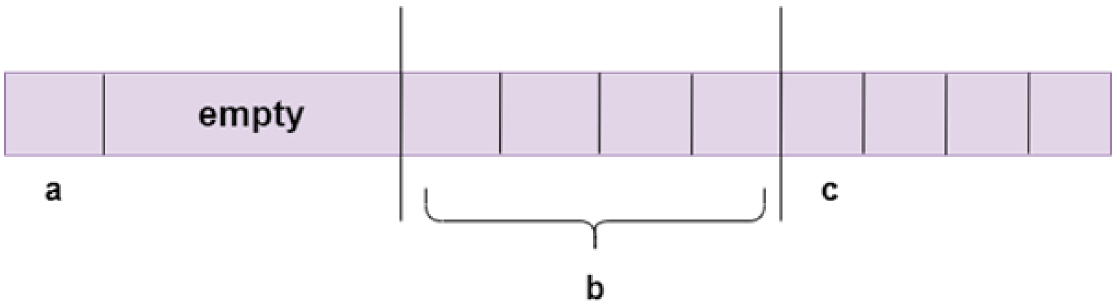



```
struct student
{
   char a;
   int b;
   char c;
  } stud1;
```

Sizeof(stud) =?????

Structure padding in C (Change the sequence)

```
struct student
{
   char a;
   int c;
   char b;
  } stud1;
```



Sizeof(stud) =**<u>12 (Why not 6?)</u>**

Example for Padding

- Consider typical 32 bit machine
- char 1 byte
- short int 2 bytes
- int 4 bytes
- double 8 bytes

Consider 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;
```

sizeof(structa_t) = 4

- 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.**

Consider typical 32 bit machine

char	1 byte
short int	2 bytes
int	4 bytes
double	8 bytes

```
// structure B
typedef struct structb_tag {
    short int s;
    char c;
    int i;
} structb_t;
```

Result = 8

- 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,
  - **structb_t requires , 2 + 1 + 1 (padding) + 4 = 8 bytes**

Consider typical 32 bit machine

char	1 byte
short int	2 bytes
int	4 bytes
double	8 bytes

```
// structure C
typedef struct structc_tag {
    char c;
    double d;
    int s;
} structc_t;
```

<u>Result = 24</u>

- Expected : sizeof(char) + 7-byte padding + sizeof(double) + sizeof(int) = 1 + 7 + 8 + 4 + 4 = 24 bytes
- **<u>along with structure members, structure type variables will also have natural alignment.</u>**
- For structc_t s[3];
  - Consider address for s[0] = 00 , size of structc_t occupies 20 bytes
  - Addres of s[1] = 20
  - Address of s[1].d = 20+1 +7 = 28
  - It is not multiple of 8 (conflicted alignment requirements of double which is 8 bytes)

// structure A
typedef struct
structa_tag {
   char c;
   short int s;
} structa_t;
sizeof(structa_t) = 4
	// structure B
	typedef struct
	structb_tag {
	   short int s;
	   char c;
	   int i;
	} structb_t;
	Result = 8
		// structure C
		typedef struct
		structc_tag {
		   char c;
		   double d;
		   int s;
		} structc_t;
		Result = 24

- Compiler introduces **alignment requirements to every structure**
  - **It will be as that of the largest member of the structure.**
  - alignment of
    - structa_t = 2
    - structb_t = 4
    - structc_t = 8
  - **If we need nested structures, the size of the largest inner structure will be the alignment of an immediate larger structure**

```
// structure D
typedef struct structd_tag {
    double d;
    int s;
    char c;
} structd_t;
```

Answer = 16

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

avoid the structure padding in C

- **Using #pragma pack(1) directive:**
  - instructs the compiler to pack structure members with particular alignment.

- **Using attribute**

Using #pragma pack(1) directive

```
#include <stdio.h>
#pragma pack(1)
struct base
{
   int a;
   char b;
   double c;
};
int main()
{
  struct base var; // variable declaration of type base
  // Displaying the size of the structure base
  printf("The size of the var is : %d", sizeof(var));
return 0;
}
```

- Without
- #pragma pack(1)
- **Output : ?????**
- Without
- #pragma pack(1)
- **Output : ?????**

Using #pragma pack(1) directive

```
#include <stdio.h>
#pragma pack(1)
struct base
{
   int a;
   char b;
   double c;
};
int main()
{
  struct base var;
  printf("The size of the var is : %d", sizeof(var));
return 0;
}
```

- Without
- #pragma pack(1)
- **Output : 16**

- With
- #pragma pack(1)
- **Output : 13**

#pragma pack(2)

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

- *number*: where *number* is 1, 2, 4, 8, or 16.
- i.e. structure members are aligned on *number*-byte boundaries or on their natural alignment boundary, whichever is less.

pragma pack(1)

a1	a2	a3	a4
b1	c1	c2	c3
c4	c5	c6	c7
c8			

pragma pack(2)

a1	a2
a3	a4
b1	Padding
c1	c2
c3	c4
c5	c6
c7	c8

Using attribute

```
#include <stdio.h>
  struct base
{
    int a;
    char b;
    double c;
}__attribute__((packed));
int main()
{
  struct base var;
  printf("The size of the var is : %d", sizeof(var));

    return 0;
}
```

- Without
- __attribute__((packed))
- **Output : 16**

- With
- __attribute__((packed))
- **Output : 13**

Sizeof for flexible array member

- struct temp{ int a; int b[]; } f;

- Sizeof(f.a) =4
- Sizeof(f.b) … Error…. Not permitted
- Sizeof(f) = 4

- struct temp{int b[]; int a;} f;
- This structure declaration not allowed
- // Sizeof(f) …. Not allowed

Union

- User-defined data type in C
- Contain elements of the different data types
- All the members in the C union are stored in the same memory location
- The size of the union will always be equal to the size of the largest member of the array.
- All the less-sized elements can store the data in the same space without any overflow
- Only one member can store data at the given instance

```c
union Data
{  int i;
float f;
char str[15]; } data;
```

Sizeof(data) ?????

```
union Data
{  int i;
float f;
char str[15]; } data;
```

Sizeof(data) 16 (Why not 15?)

```
union Data
{  int i;
float f;
char str[13];  } data;
```

Sizeof(data) 16 (Why not 13?)

Structure vs Union

Structure	Union
• The size of the structure is equal to or greater than the total size of all of its members.	• The size of the union is the size of its largest member
• The structure can contain data in multiple members at the same time.	• Only one member can contain data at the same time.
• It is declared using the struct keyword.	• It is declared using the union keyword.

Formatted Output

type	code	typical literal	sample format strings	converted string values for output
int	d	512	"%14d"	" 512"
			"%-14d"	"512 "
double	f	1595.1680010754388	"%14.2f"	" 1595.17"
			"%.7f"	"1595.1680011"
	e		"%14.4e"	" 1.5952e+03"

- Check other data types

End of Introduction

Extra

- The header provides access not only to other .c files in the same program, but likewise to libraries that may be distributed in binary form
  - The relationship of one .c file to another is exactly the same as a library that depends on another

- Since a programming interface needs to be in text form no matter the format of the implementation, header files make sense as a separation of concerns.

- As others have mentioned, the program that resolves function calls and accesses between libraries and sources (translation units) is called the linker

- The linker does not work with headers
- It just makes a big table of all the names that are defined in all the translation units and libraries, then links those names to the lines of code that access them
- Archaic usage of C even allows for calling a function without any implementation declaration; it was just assumed that every undefined type was an int