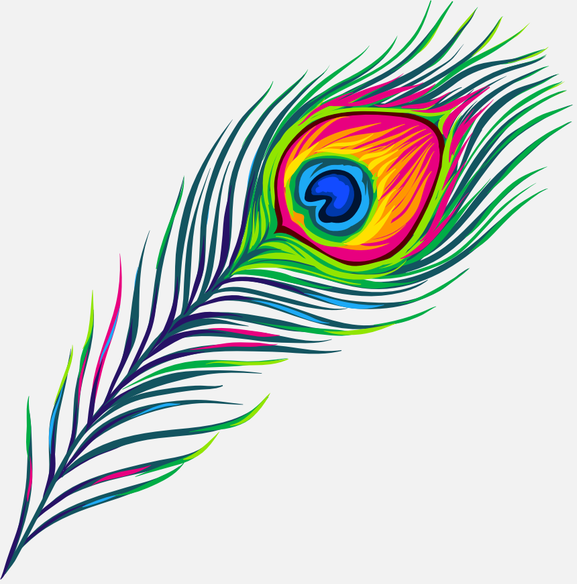


“Do everything you have to do, but not with ego, not with lust, not with envy but with love, compassion, humility, and devotion.” – Shree Krishna

ADVANCED C WORKSHOP

```c
#include <stdio.h>

struct advanced_C_workshop
{
    unsigned int year;
    unsigned int lecture;
};

int main()
{
    struct advanced_C_workshop by_IT_club = {2023, 1};
    return 0;
}
```


By:

Sudip Tiwari

Sandhya Baral

TABLE OF CONTENT

X86 AND X64 ARCHITECTURE

Size of a char ?

Size of a char ?

1 byte

Size of a short int ?

Size of a short int ?

2 bytes

Size of an integer ?

Size of an integer ?

4 bytes

-2,147,483,648 to 2,147,483,647.

Size of a float ?

Size of a float ?

4 bytes

- 1 bit is allocated for the sign.
- 8 bits are allocated for the exponent.
- 23 bits are allocated for the significand (also called mantissa).

Size of a char pointer?

Size of a char pointer?

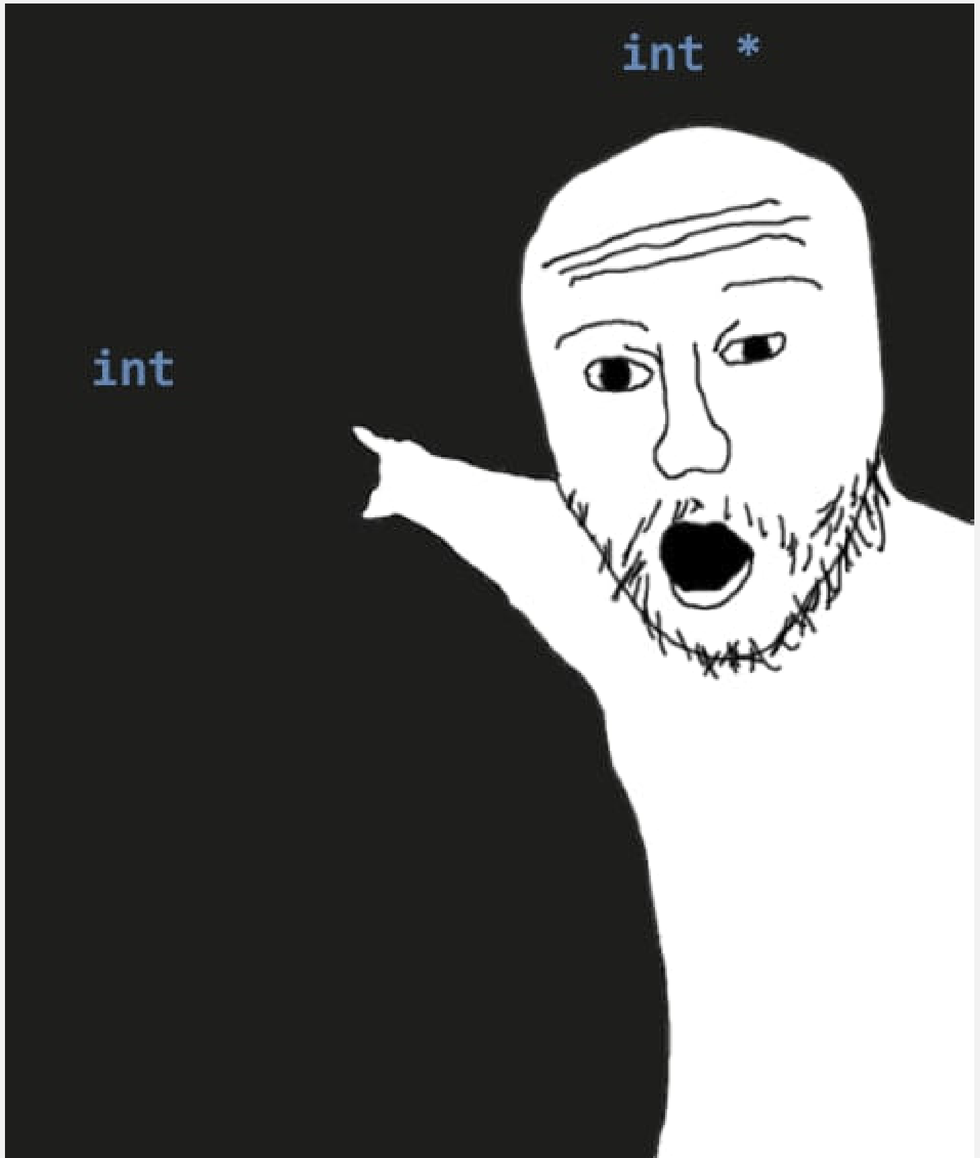
Size of an int pointer?

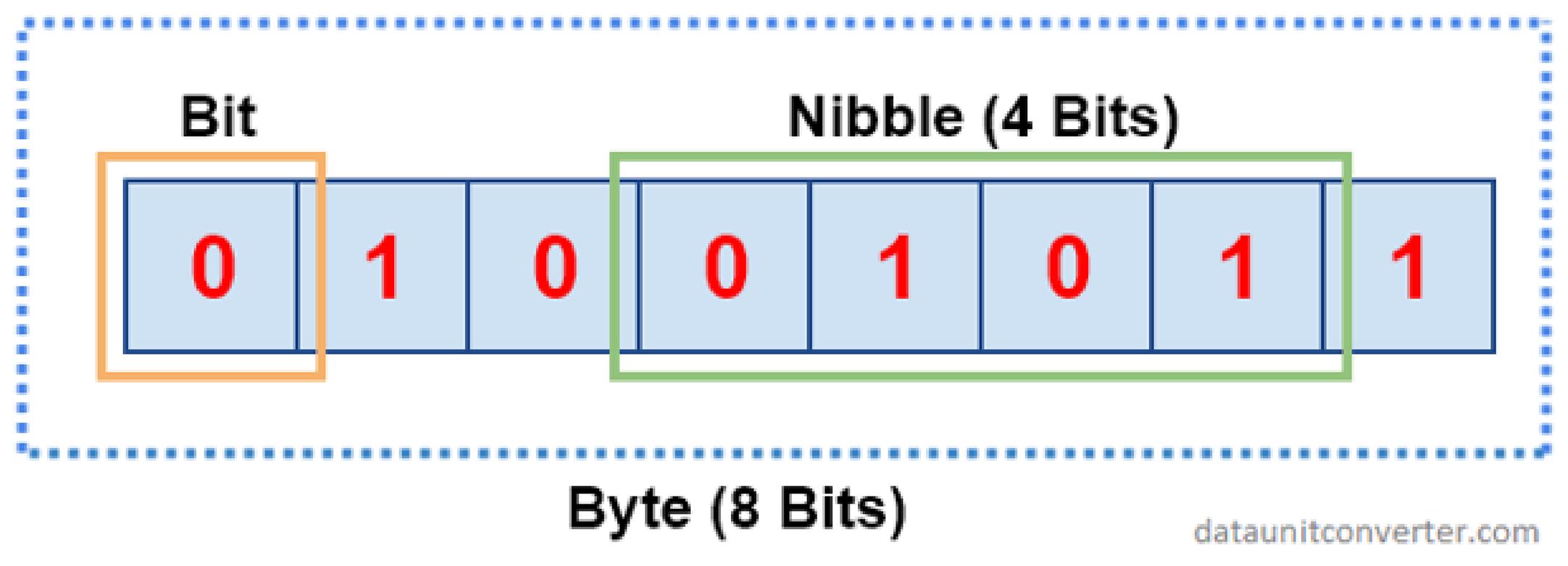
Size of a char pointer?

Size of an int pointer?

Size of a float pointer?

Let’s talk about Pointers




Bit
Nibble (4 Bits)
0
1
0
0
1
0
1
1
Byte (8 Bits)
dataunitconverter.com

What does it mean when we say a char is 1 byte ?

What does it mean when we say a char is 1 byte ?

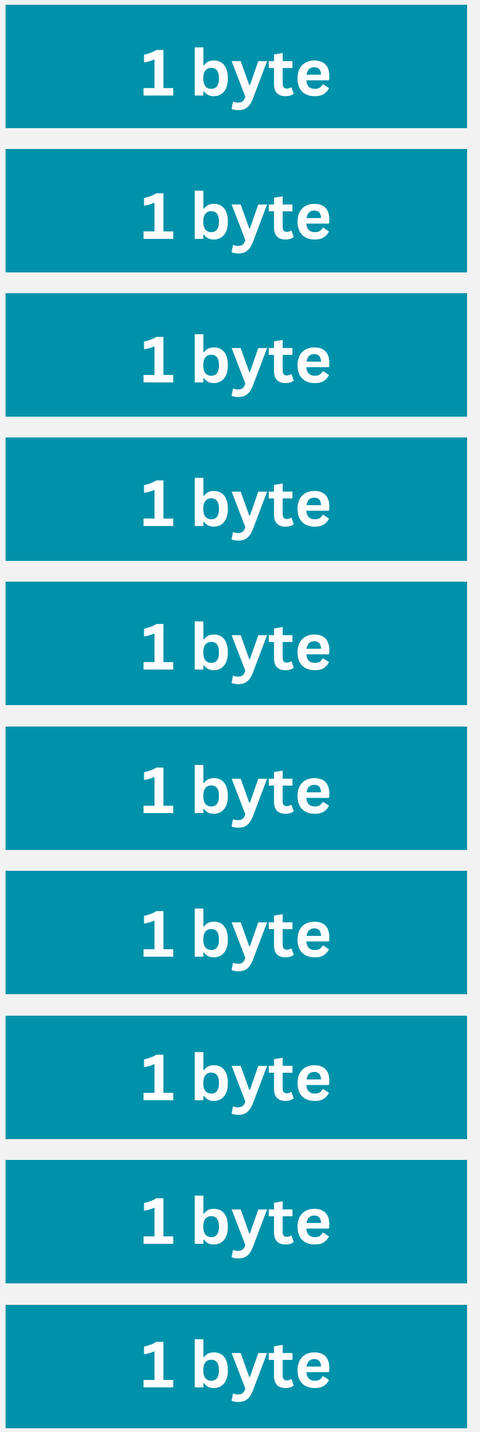
Explanation:

How many bits can a computer access/manipulate at a single time ?

- Serial & Parallel Communication
- How can we make a computer faster?
- Data bus length (Parallel)

The Memory Stack :

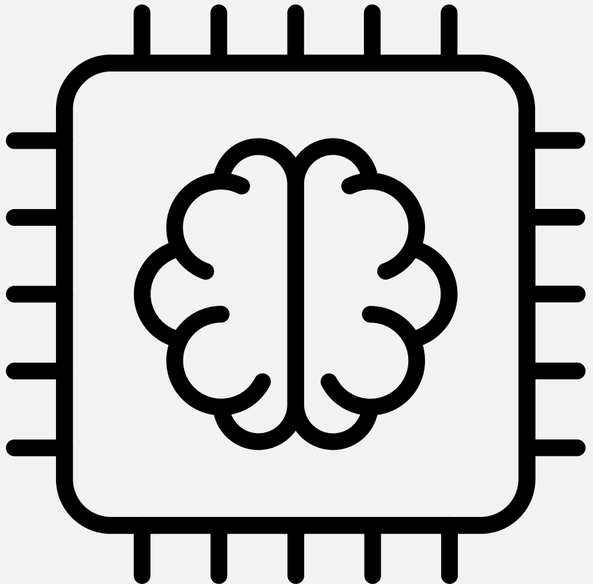
- What does it mean to write a program ?
- How is the program executed by the microprocessor ?
- **Instruction length of various instructions**
- What is logical for designers of computer ?



How many bits can a computer access/manipulate at a single time ?

Answer: Let’s not talk about bits, but about bytes.

Why not fetch chunk of bytes at a single time ?



1 byte

1 byte

1 byte

1 byte

1 byte

1 byte

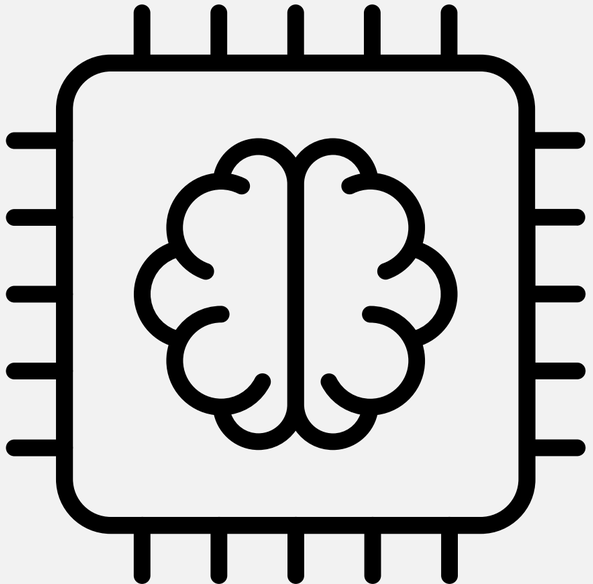
1 byte

1 byte

1 byte

1 byte

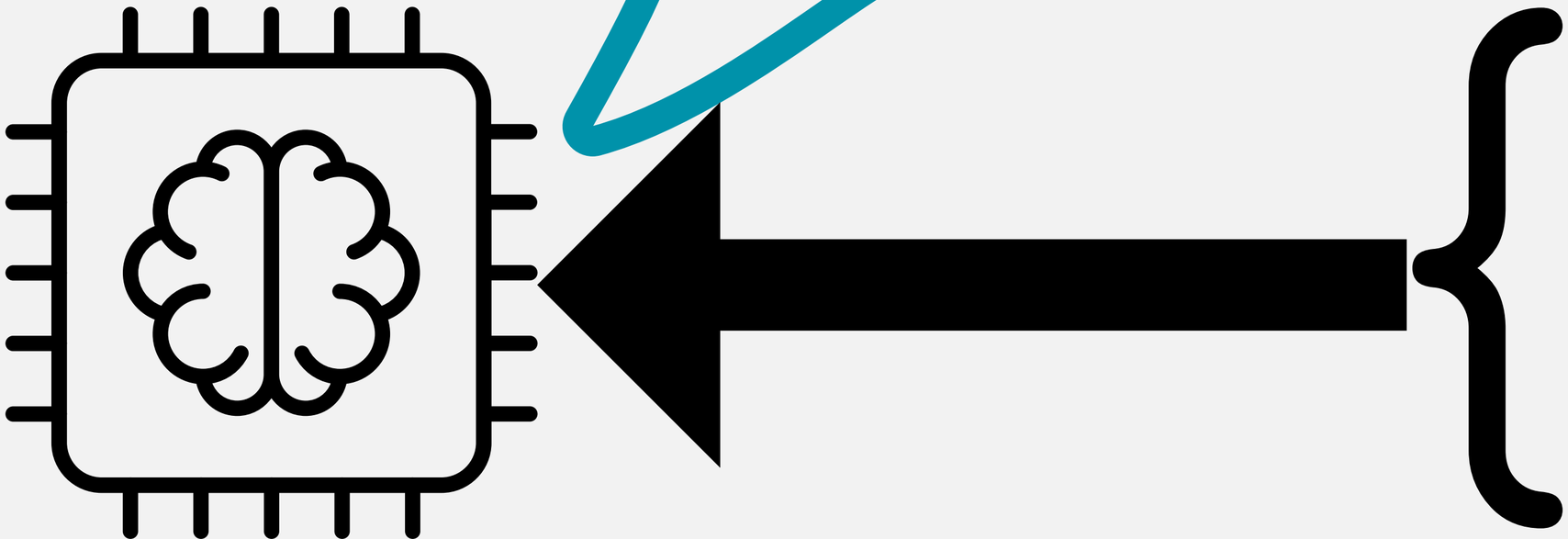
I will fetch 4 bytes
at a single time



1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte

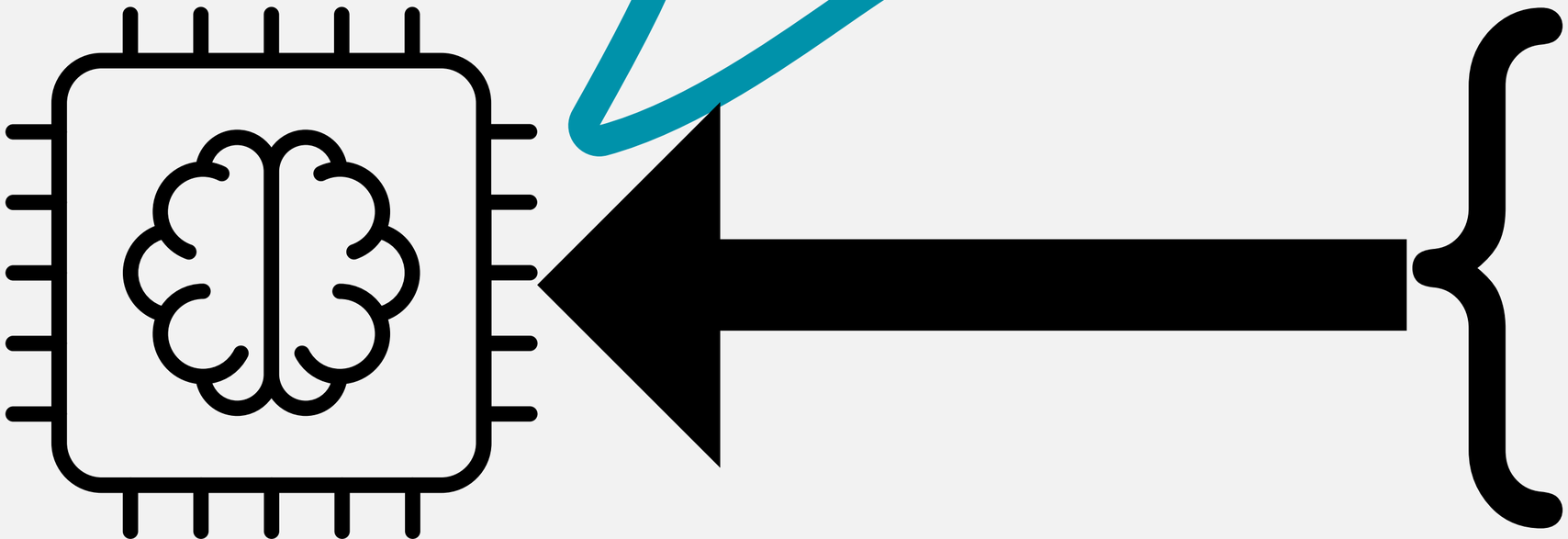
4 bytes = 4 * 8 bits
= 32-bits

They call me a 32-bit system.



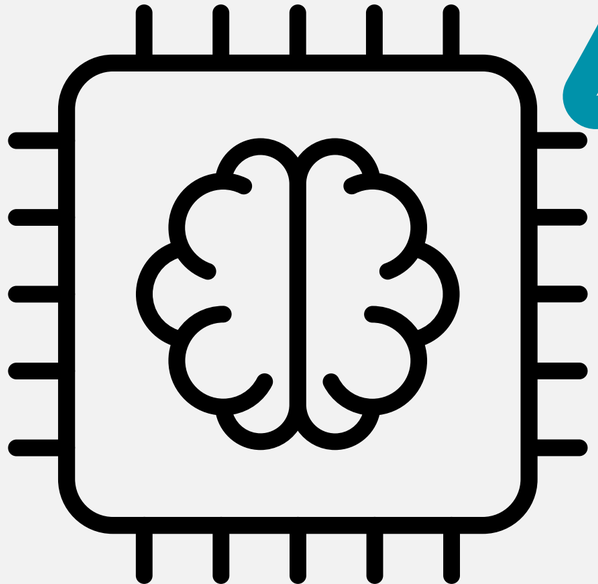
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte

Since I maintain backward compatibility with my famous great grandfather 8086, they call me x86.



1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte

I will fetch 8 bytes
at a single time

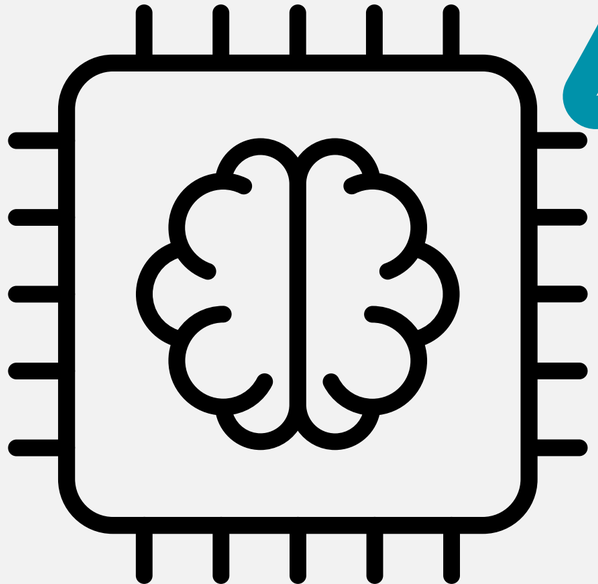


1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte

8 bytes = 8 * 8 bits

= 64 bits

They call me a 64-bit processor.



1 byte

1 byte

1 byte

1 byte

1 byte

1 byte

1 byte

1 byte

1 byte

1 byte

Word Length :

- Collection of bits that can be addressed, transferred or manipulated as a single unit
- Specifically related to the data bus width

Okay cool fact, but what is the size of pointer ?

- Data bus width
- Address bus width

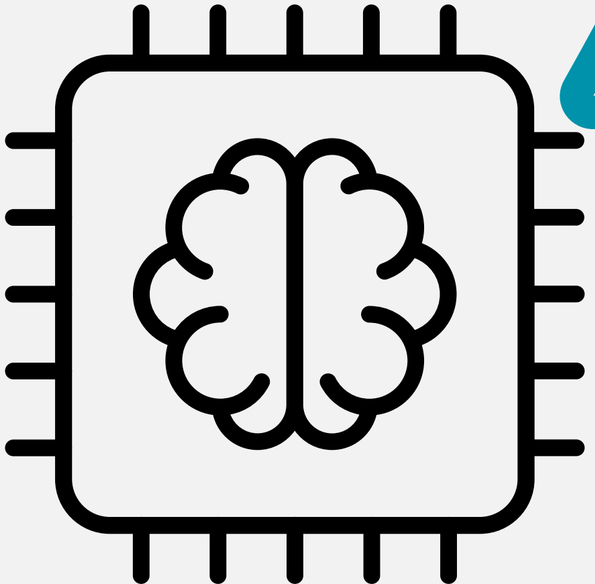
x86	x64
32-bit	64-bit
32-bit (4 GB memory)	36-bit (Intel i7) (64 GB memory)

18,446,744,073,709,551,616 bytes

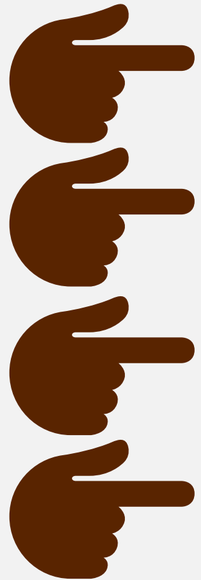
In principle, a 64-bit microprocessor can address 16 EiB ($16 \times 1024^6 = 2^{64}$ = 18,446,744,073,709,551,616 bytes, or about 18.4 exabytes) of memory.

If I have 2-bit address bus width:

00 Location 1
01 Location 2
10 Location 3
11 Location 4



I can point to 4 different memory location

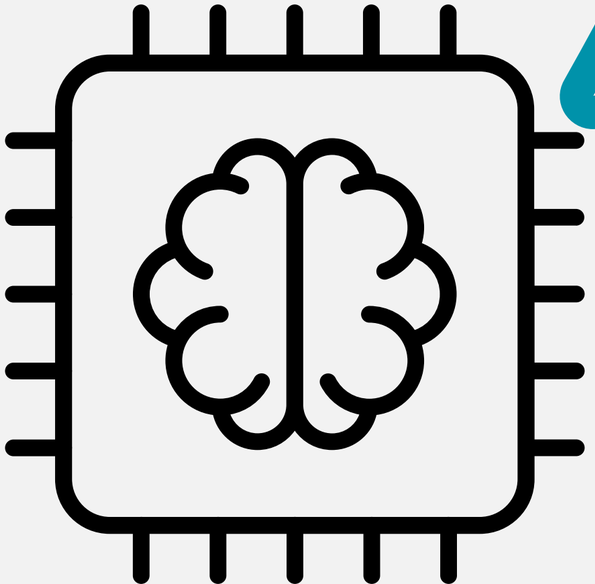


1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte

If I am 8-bit processor:

(I also have 8 bit address bus width)

I can have a memory of
size of 256 Bytes.



1 byte

1 byte

1 byte

1 byte

1 byte

1 byte

1 byte

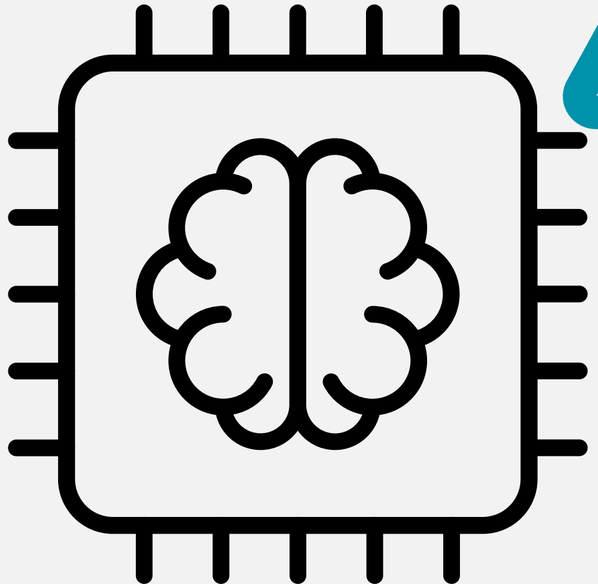
1 byte

1 byte

1 byte

If I am 8-bit processor:
(I also have 8 bit address bus width)

To locate 256 different locations,
I need log2(256) = 8 bit field.

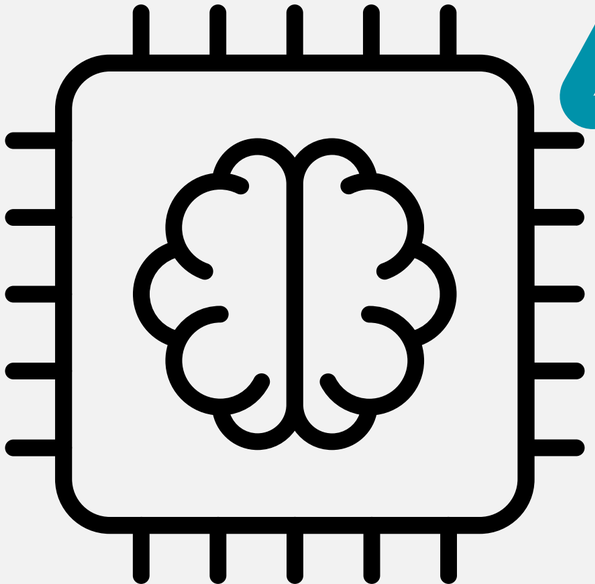


1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte

If I am 8-bit processor:

(I also have 8 bit address bus width)

8 bit field means 1 Byte field.
Hence, the size of a pointer in my system is 8-bit or 1 Byte.

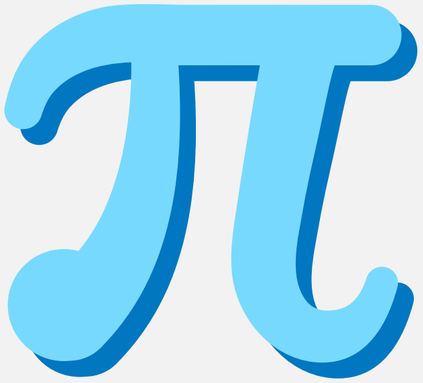


1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte

- Size of any pointer in 8-bit architecture : 8-bit or 1 Byte.
- Size of any pointer in 16-bit architecture : 16-bit or 2 Byte.
- Size of any pointer in 32-bit architecture : 32-bit or 4 Byte.
- Size of any pointer in 64-bit architecture : 64-bit or 8 Byte.

Size of any pointer in x-bit architecture : x-bit or (x/8) Byte.

Size of any pointer in x-bit architecture : x-bit or (x/8) Byte.



Size of any pointer in x-bit architecture : x-bit or (x/8) Byte.

Ans: Because pointers hold memory address, irrespective of type.

Lifetime of non-static variable:

- Be extra careful when returning pointers to non-static stack based variables since they might be dangling and we‘d be headed to uncharted territory.

Lifetime of non-static variable:

- On the contrary, returning string literals as char pointers is good because their lifetime is attached to program’s runtime because they are stored in read-only section of memory (i.e, They are immutable).

1_string_literals.c > ...

```c
#include <stdio.h>

// Function that returns a pointer to the first byte of the string
char *getFirstByteOfString(char *str) {
    return str;
}

int main() {
    char myString[] = "Hello, world!"; // Example string

    // Call the function and store the returned pointer
    char *firstByte = getFirstByteOfString(myString);

    // Print the pointer value and the character it points to
    printf("Pointer to the first byte: %p\n", (void *)firstByte); //
    to avoid any potential issues with pointer mismatching.

    printf("Character at that pointer: %c\n", *firstByte);

    return 0;
}
```

ENDIANNESS

Little Endian

- Lower Order Bytes in Lower Memory Address
- Higher Order Bytes in Higher Memory Address

0x123456

Address 000	56
Address 001	34
Address 010	12
Address 011	
Address 100	
Address 101	
Address 110	
Address 111	

Big Endian

- Lower Order Bytes in Higher Memory Address
- Higher Order Bytes in Lower Memory Address

0x123456

Address 000	12
Address 001	34
Address 010	56
Address 011	
Address 100	
Address 101	
Address 110	
Address 111	

Little Endian

- Lower Order Bytes in Lower Memory Address
- Higher Order Bytes in Higher Memory Address

Big Endian

- Lower Order Bytes in Higher Memory Address
- Higher Order Bytes in Lower Memory Address

```c
#include<stdio.h>


int main() {
    int a = 1193046;
    return 0;
}
```

(end of file)

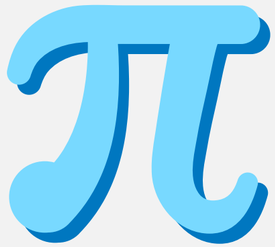
memory

0x7fffffffda2c	0x7fffffffda4b	8

address	hex
more	
0x7fffffffda2c	56 34 12 00 01 00 00 00

Which one do we prefer ?

- Most of the devices (x86, most AMD processors, etc.) are little-endian.
- Big-endian are dominant in networking protocols.

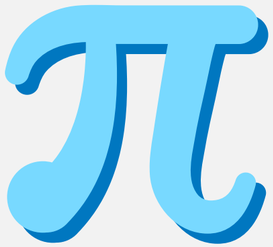



Why?
Why?
Why?
Oh, that's why.

Which one do we prefer ?

- Arithmetic operations (like addition, subtraction and multiplication) start with the least significant bytes and carry their results into the operations for bytes of increasing significance.

- For Division and Comparison, we operate on the most significant bytes first.



BITFIELDS

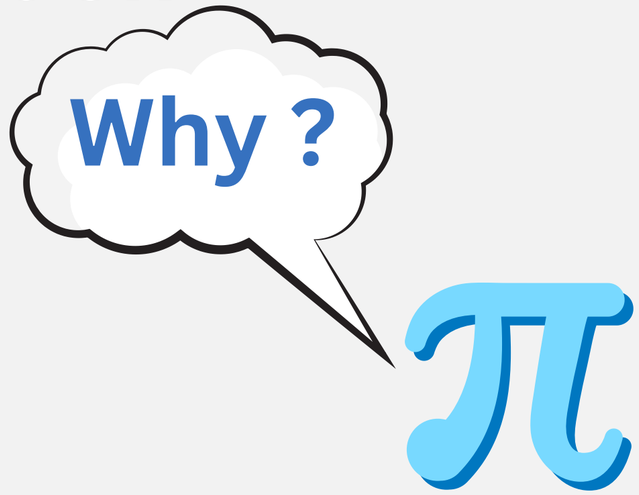
Why use bitfields ?

- Reduce memory consumption when a program requires a number of integer variables which always will have low values
- In some cases when a bunch of flags need to be stored/interrogated.

Examples as code :

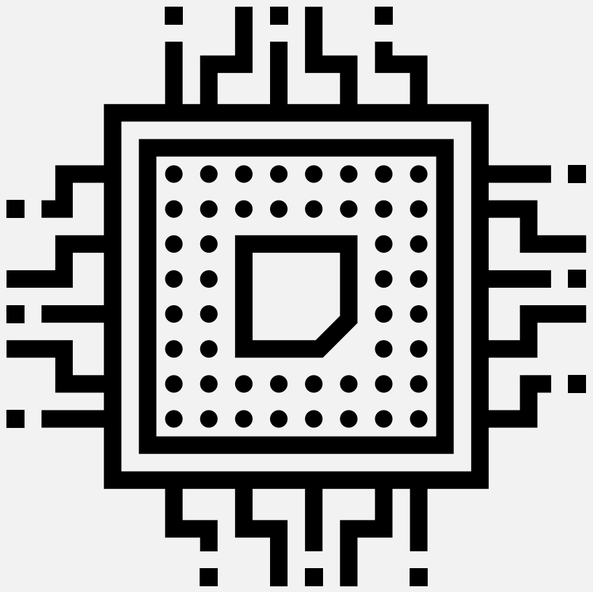
PADDING

- char can start on any byte address.
- 2-byte shorts must start on an even address
- 4-byte ints or floats must start on address divisible by 4
- 8-bytes long or doubles must start on an address divisible by 8



I am 64-bit processor.

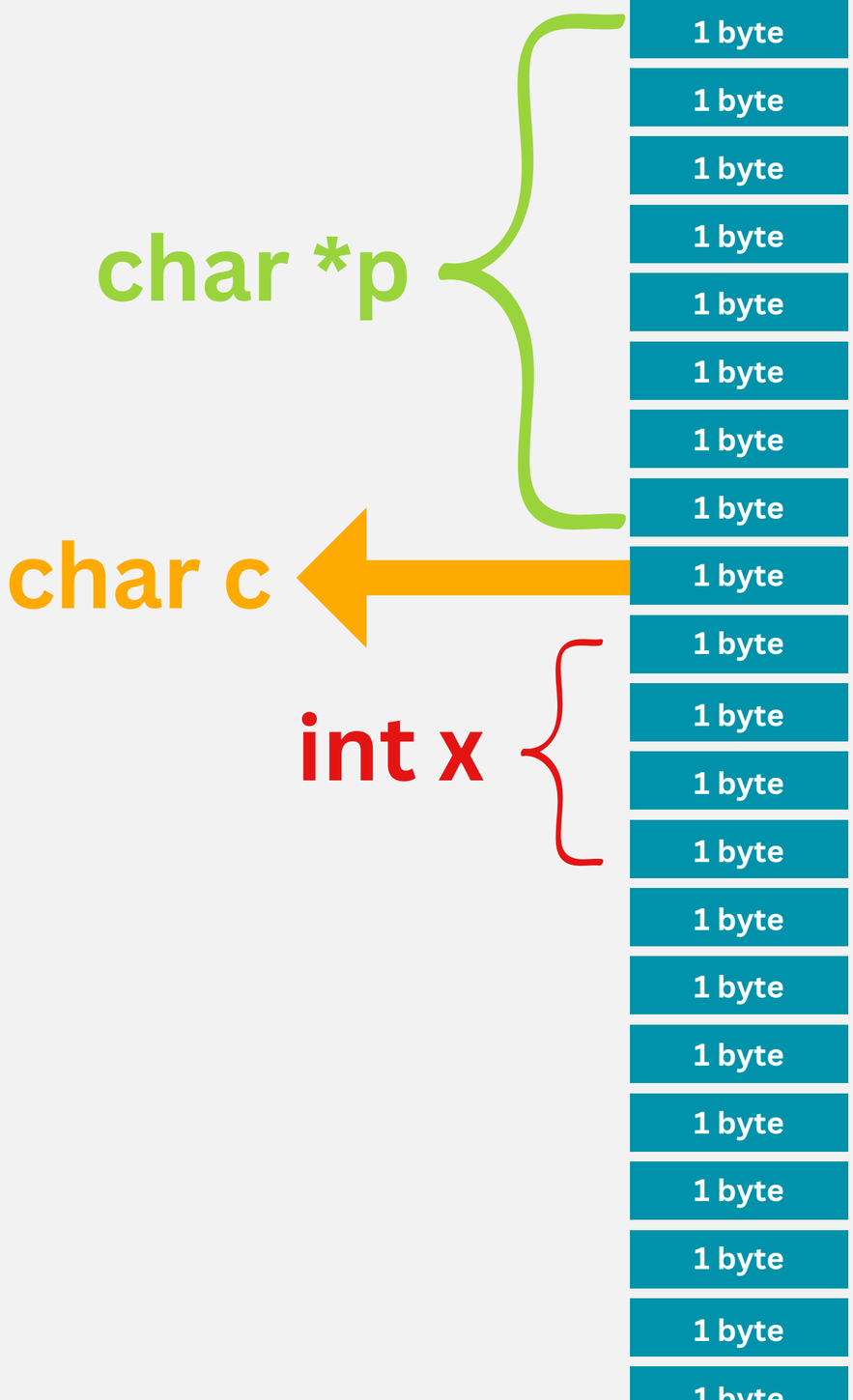
I fetch 8 bytes data
at a time.



1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte
1 byte

padding.c > ...

```c
1  char *p;
2  char c;
3  int x;
```

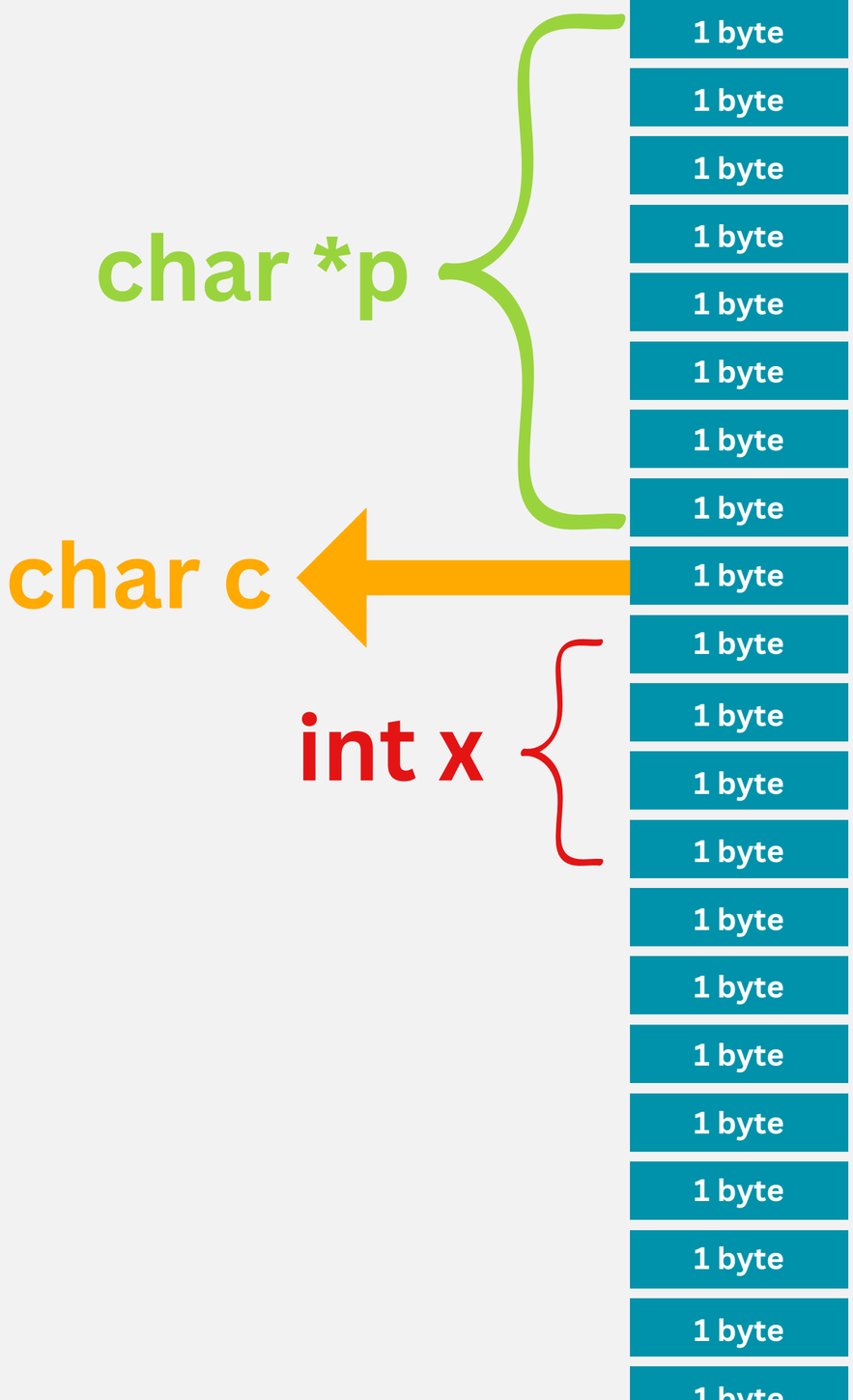


What we assume :

char *p; // 8 bytes
char c; // 1 byte
int x; // 4 bytes

padding.c > ...

```c
1  char *p;
2  char c;
3  int x;
```

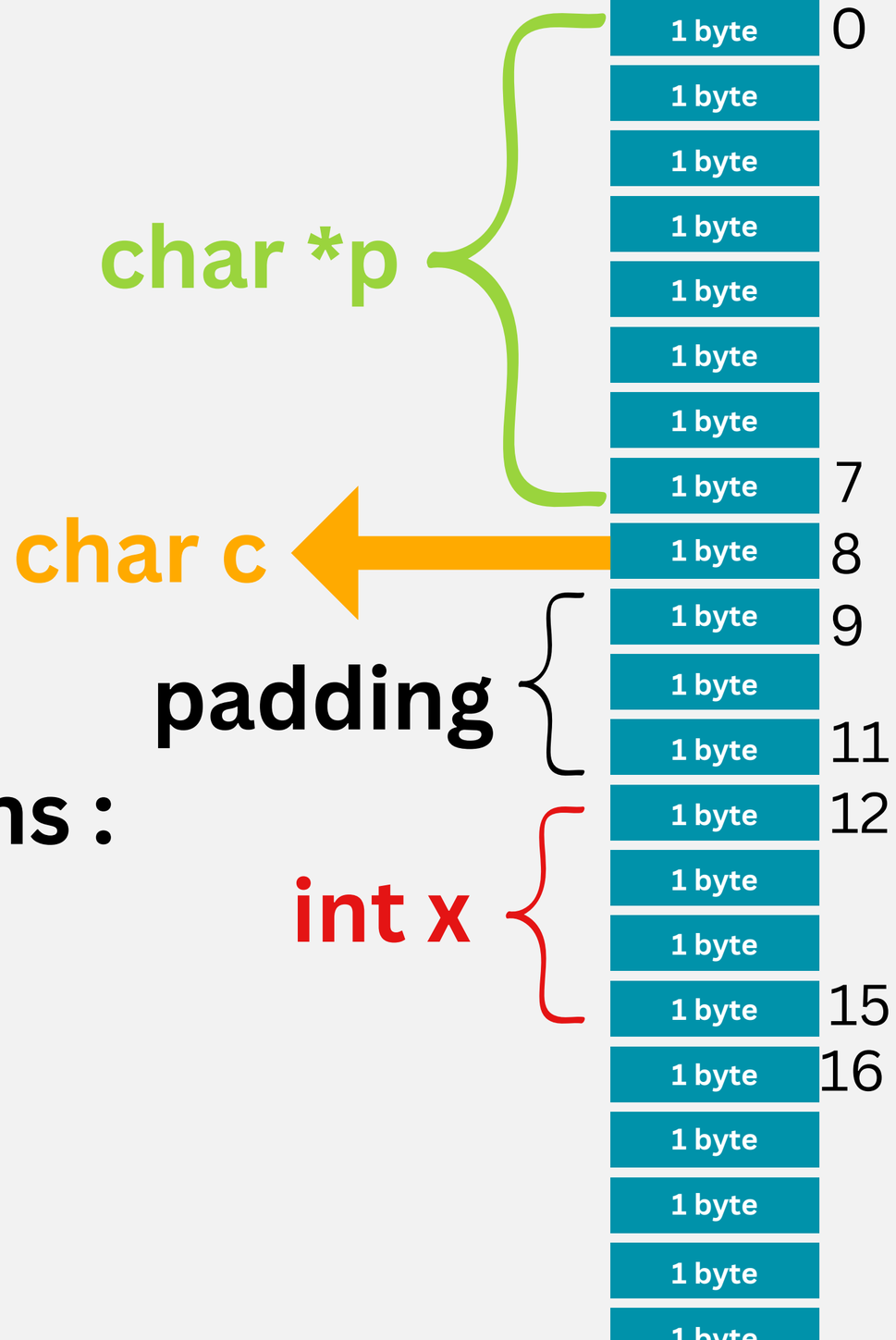


What we assume :

char *p; // 8 bytes
char c; // 1 byte
int x; // 4 bytes

padding.c > ...

```c
1  char *p;
2  char c;
3  int x;
```

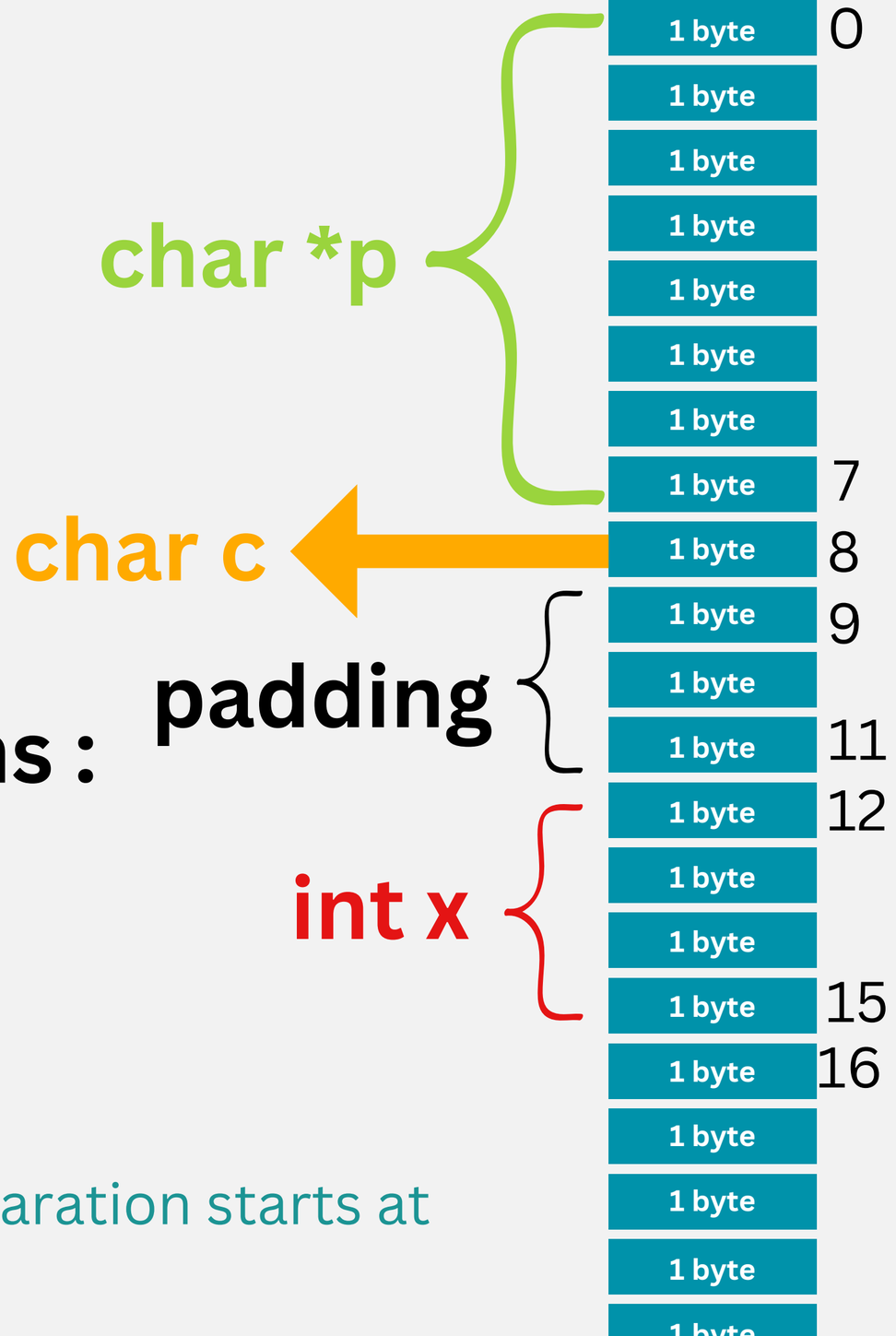


What actually happens :

```
char *p;        // 8 bytes
char c;         // 1 byte
char pad[3];    // 3 bytes
int x;          // 4 bytes
```

padding.c > ...

```c
1  char *p;
2  char c;
3  int x;
```



What actually happens :

```
char *p;          // 8 bytes
char c;           // 1 byte
char pad[3];      // 3 bytes
int x;            // 4 bytes
```

Here, padding of size 3 is always fixed since declaration starts at strictest type.

padding.c > pad1

```c
1  char c;
2  char *p;
3  int x;
```

What can we say about M and N ?

```c
8  char c;
9  char pad1[M];
10 char *p;
11 char pad2[N];
12 int x;
```

padding.c > pad1

```c
1  char c;
2  char *p;
3  int x;
```

What can we say about M and N ?

N = 0

```c
 8  char c;
 9  char pad1[M];
10  char *p;
11  char pad2[N];
12  int x;
```

padding.c > pad1

```c
1  char c;
2  char *p;
3  int x;
```

What can we say about M and N ?

N = 0

M = 0 to 7 (anything)

```c
 8  char c;
 9  char pad1[M];
10  char *p;
11  char pad2[N];
12  int x;
```

Is there a better way to declare variables ?

```
1  char *p;      // 8 bytes
2  int x;        // 4 bytes
3  char c;       // 1 byte
```

Is there a better way to declare variables ?

```
1  char *p;     // 8 bytes
2  int x;       // 4 bytes
3  char c;      // 1 byte
```

This technique becomes more interesting when applied to nonscalar variables, especially structs.

STRUCTURE ALIGNMENT AND PADDING

In general, a struct instance will have the alignment of its widest scalar member.

Compilers do this as the easiest way to ensure that all the members are self aligned for fast access.

In general, a struct instance will have the alignment of its widest scalar member.

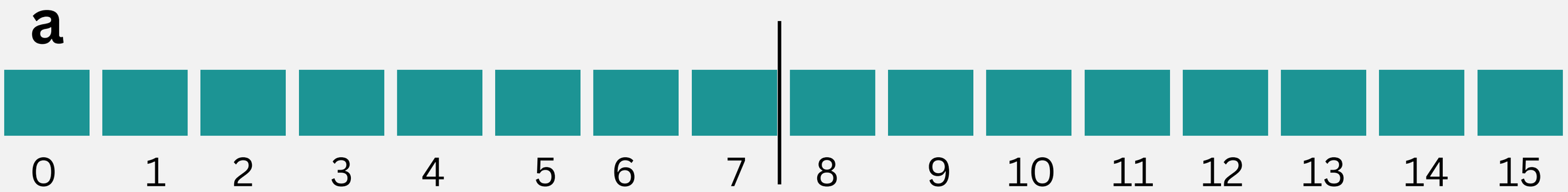
Any structure will take the size equal to the multiple of maximum bytes taken by a variable in that structure.

Compilers do this as the easiest way to ensure that all the members are self aligned for fast access.

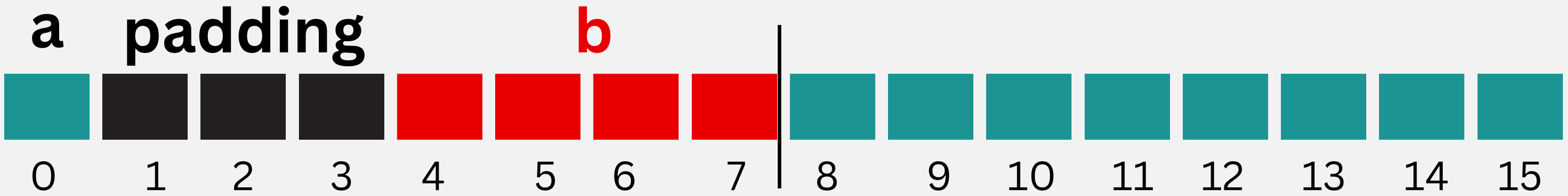
```
1  struct abc {
2      char a;
3      int b;
4  } structure_variable;
5
```

```
1  struct abc {
2      char a;
3      int b;
4  } structure_variable;
5
```

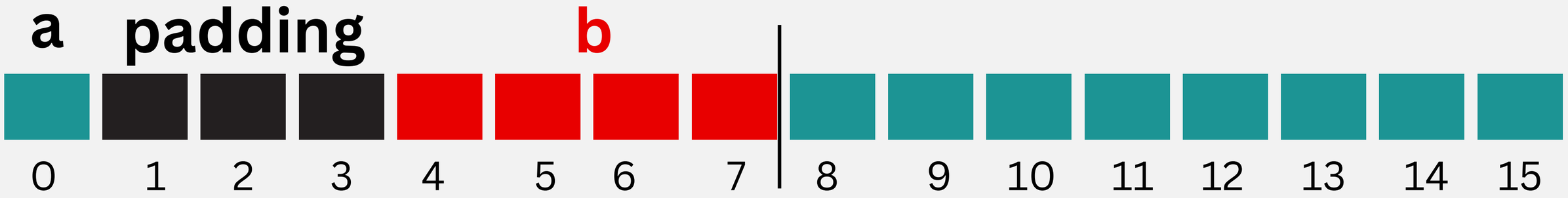
```c
struct abc {
    char a;
    int b;
} structure_variable;

```



```
1  struct abc {
2      char a;
3      int b;
4  } structure_variable;
5
```



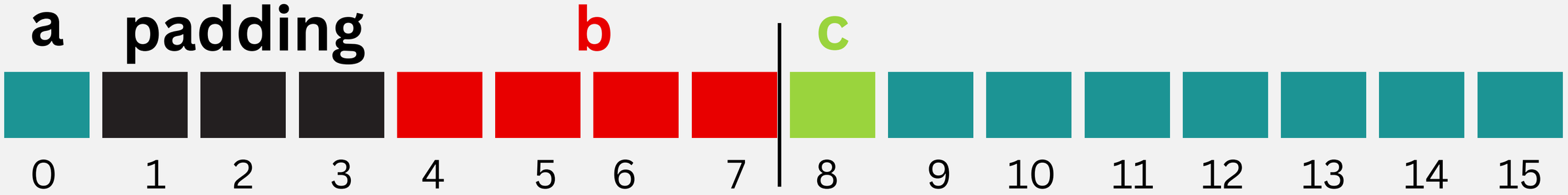
```c
struct abc {
    char a;
    int b;
} structure_variable;

```



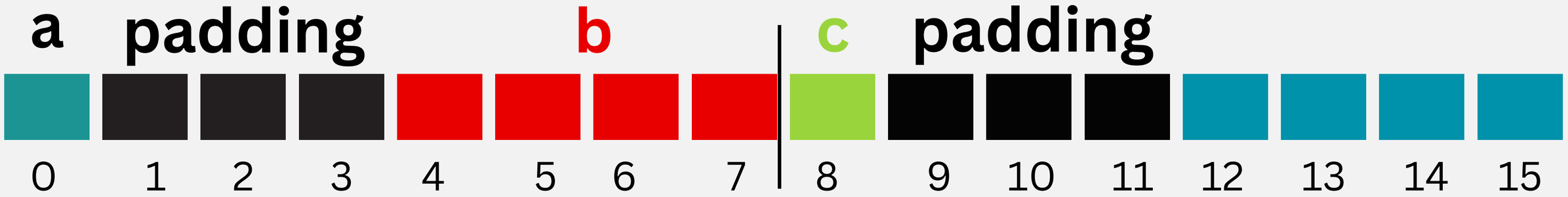
Total size : 8 bytes

```c
struct abc {
    char a;
    int b;
    char c;
} structure_variable;

```

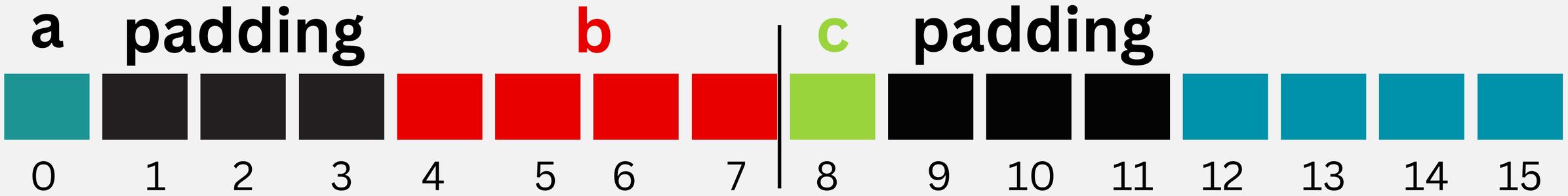
```c
struct abc {
    char a;
    int b;
    char c;
} structure_variable;

```



```c
struct abc {
    char a;
    int b;
    char c;
} structure_variable;

```



```c
struct abc {
    char a;
    int b;
    char c;
} structure_variable;

```



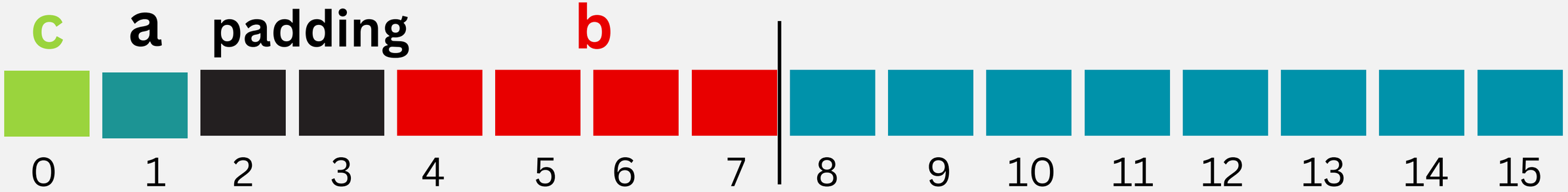
Total Size: 12 bytes

```
1  struct abc {
2      char a;
3      int b;
4      char c;
5  } structure_variable;
6
```

What if ?

```
1  struct abc {
2      char a;
3      int b;
4      char c;
5  } structure_variable;
6
```

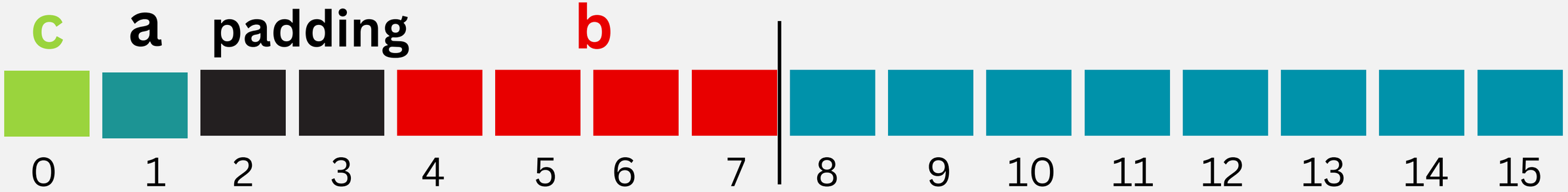
What if ?



Total size : 8 bytes

```
1  struct abc {
2      char a;
3      int b;
4      char c;
5  } structure_variable;
6
```

What if ?



Total size : 8 bytes

```c
struct abc {
    char a;
    int b;
    char c;
} structure_variable;

```

12 bytes

```c
struct abc {
    char a;
    int b;
    char c;
} structure_variable;

```

What if ?

8 bytes

```c
struct abc {
    char a;
    int b;
    char c;
} structure_variable;

```

12 bytes


```c
struct abc {
    char a;
    int b;
    char c;
} structure_variable;

```

What if ?

8 bytes

Consider an example:

```
16  struct Foo {
17      char* p;
18      long x;
19      char c;
20  };
```

```
16  struct Foo {
17      char* p;
18      long x;
19      char c;
20  };
```

```
16  struct Foo {
17      char* p;     //8 bytes
18      long x;      //4 bytes
19      char c;      //1 byte
20      char pad[3];//3 bytes-trailing padding
21  };
```

Consider another example:

```
16  struct Parent {
17      char c;
18      struct Child {
19          char *p;
20          short x;
21      } child;
22  };
```

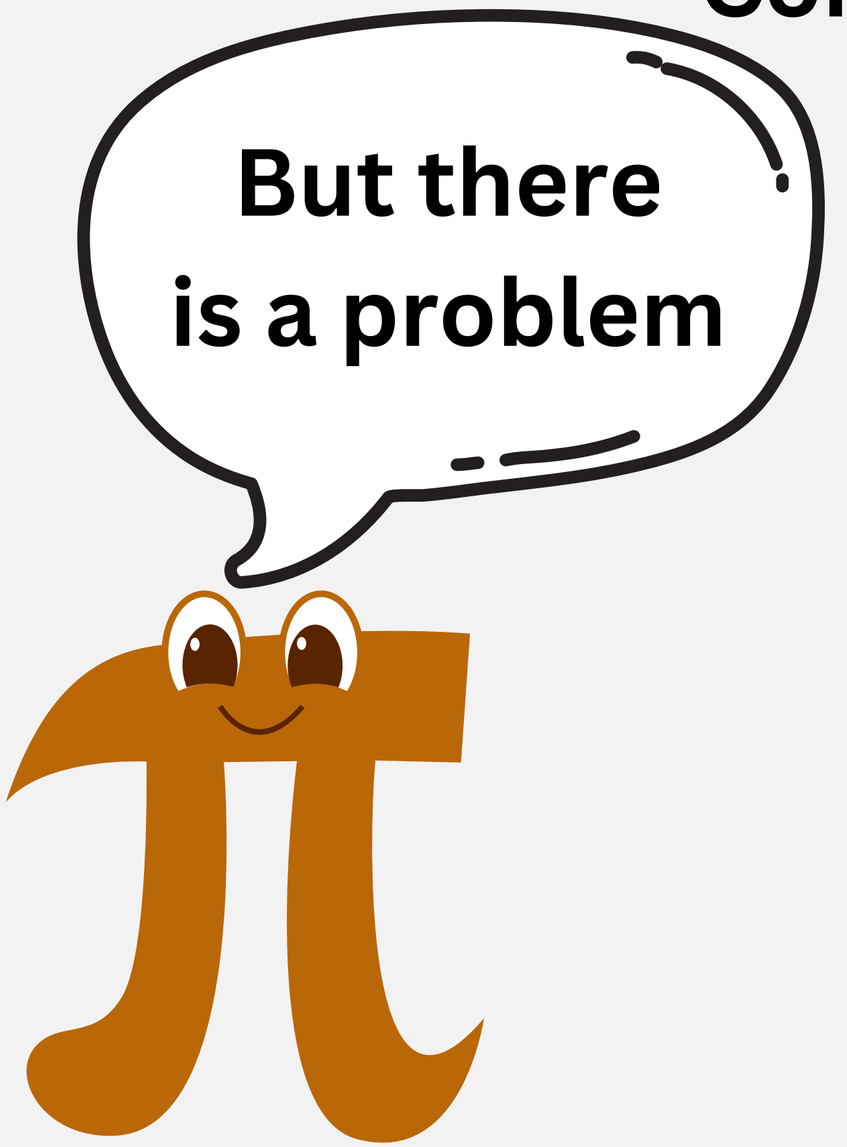
```
16  struct Parent {
17      char c;
18      struct Child {
19          char *p;
20          short x;
21      } child;
22  };
```

```
16  struct Parent {
17      char c;             //1 byte
18      char pad1[7];       //7 bytes
19      struct Child {
20          char *p;        //8 bytes
21          short x;        //2 bytes
22          char pad2[6];//6 bytes
23      } child;
24  };
```

Compiler does it this way:

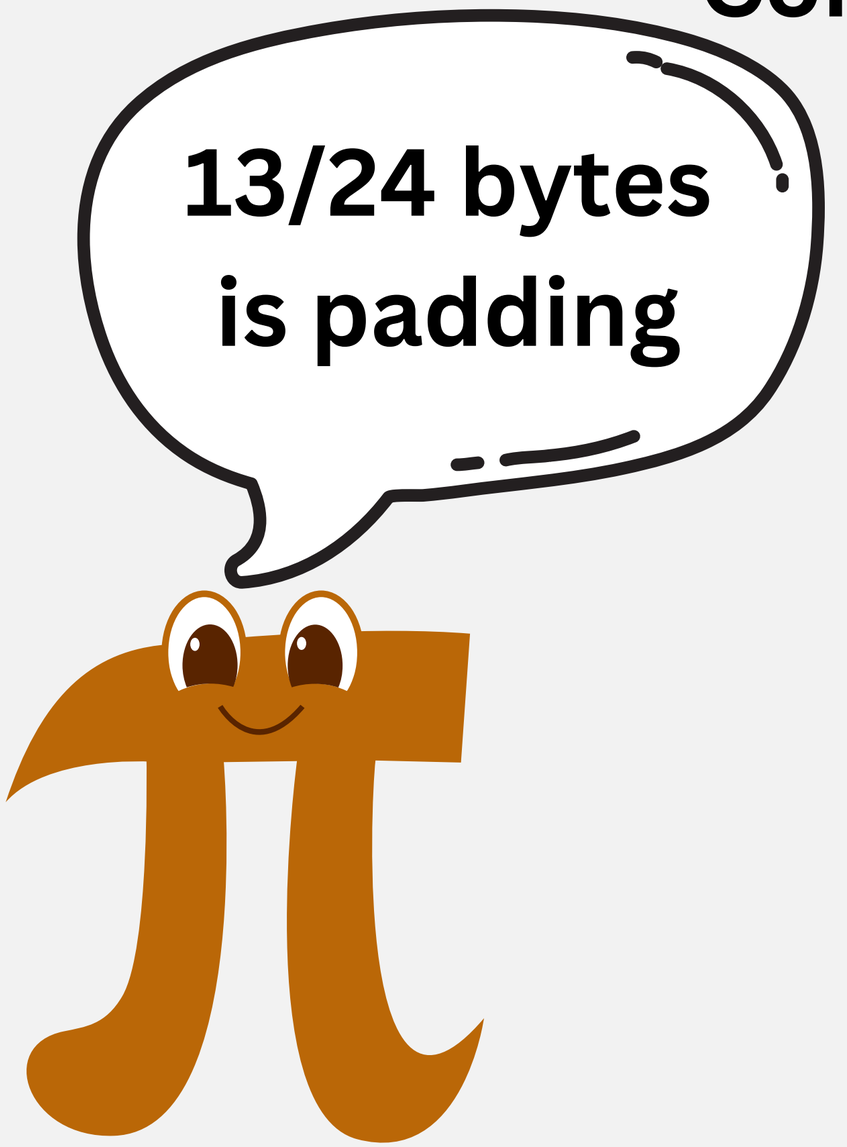
```
16  struct Parent {
17      char c;             //1 byte
18      char pad1[7];       //7 bytes
19      struct Child {
20          char *p;        //8 bytes
21          short x;        //2 bytes
22          char pad2[6];//6 bytes
23      } child;
24  };
```

Compiler does it this way:




```
16  struct Parent {
17      char c;             //1 byte
18      char pad1[7];       //7 bytes
19      struct Child {
20          char *p;        //8 bytes
21          short x;        //2 bytes
22          char pad2[6];//6 bytes
23      } child;
24  };
```

Compiler does it this way:




```
16  struct Parent {
17      char c;             //1 byte
18      char pad1[7];       //7 bytes
19      struct Child {
20          char *p;        //8 bytes
21          short x;        //2 bytes
22          char pad2[6];//6 bytes
23      } child;
24  };
```


Abhi theek karke deta hu

Structure Reordering :

```
16  struct Bhupendar_Jogi {
17      char c;
18      struct Bhupendar_Jogi* p;
19      short x;
20  };
```

Structure Reordering :

```
16  struct Bhupendar_Jogi {
17      char c;
18      struct Bhupendar_Jogi* p;
19      short x;
20  };
```

```
16  struct Bhupendar_Jogi {
17      char c; // 1 byte
18      char pad1[7]; // 7 bytes
19      struct Bhupendar_Jogi* p; // 8 bytes
20      short x; // 2 bytes
21      char pad2[6]; //6 bytes
22  };
```

24 bytes

Structure Reordering :

```
16  struct Bhupendar_Jogi {
17      char c;
18      struct Bhupendar_Jogi* p;
19      short x;
20  };
```

```
16  struct Bhupendar_Jogi {
17      struct Bhupendar_Jogi* p;
18      short x;
19      char c;
20  };
```

Structure Reordering :

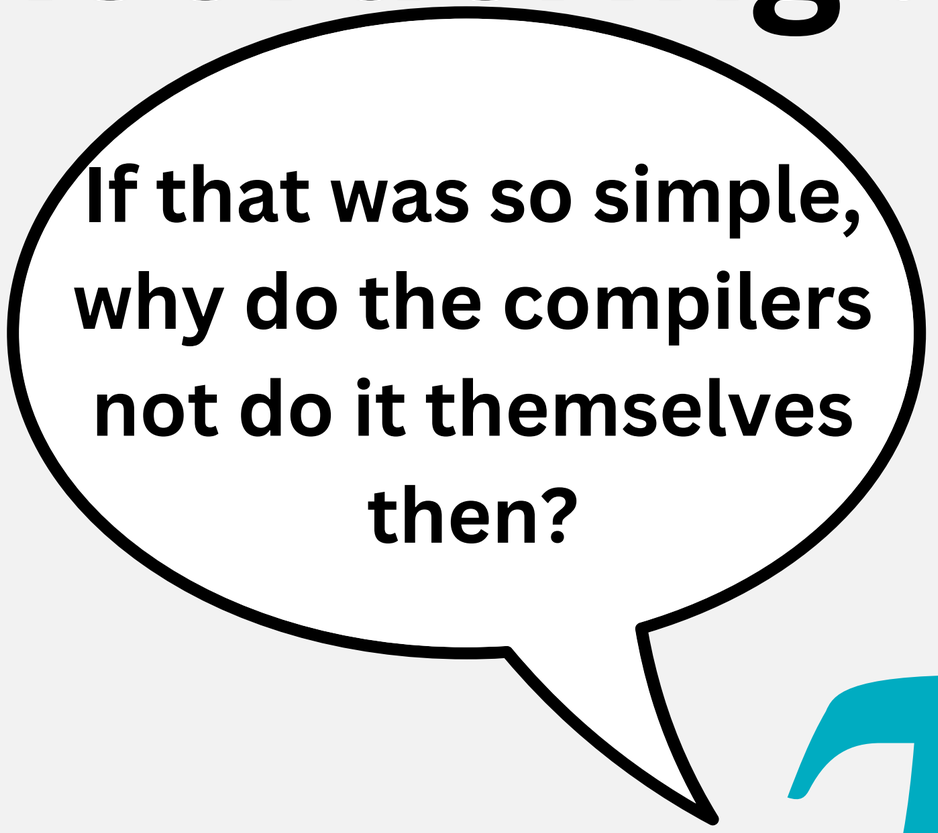
```
16  struct Bhupendar_Jogi {
17      struct Bhupendar_Jogi* p;
18      short x;
19      char c;
20  };
```

```
16  struct Bhupendar_Jogi {
17      struct Bhupendar_Jogi* p; //8 bytes
18      short x;                  //2 bytes
19      char c;                   //1 byte
20      char pad[5];              //5 bytes
21  };
```

16 bytes


Fair & Lovely
winter FAIRNESS
Before:
24 bytes
After:
16 bytes
Fair & Lovely
winter fairness
FAIRNESS EXPERT FOR SOFT, FAIR SKIN

Structure Reordering :



Structure Reordering :

C was originally designed for writing OS and other codes close to hardware.

Structure Reordering :

C was originally designed for writing OS and other codes close to hardware.

Automatic reordering often interferes with a system programmer’s ability to lay out structures that exactly match the byte and bit-level layout of memory-mapped device control blocks.

Structure Reordering :

example: PNG image

STRUCTURE BITFIELDS

example :

Switch: 1 or 0

example :

Switch: 1 or 0

If we store it as int,

example :

Switch: 1 or 0

If we store it as int,

We use just 1 bit out of 32 bits.
Memory wastage.

Consider this example :

Assuming we are on a 32-bit system:

```
1  struct
2  {
3      unsigned int first : 9;
4      unsigned int second : 7;
5      unsigned int may_straddle : 30;
6      unsigned int last : 18;
7  } tricky_bits;
8
```

Consider this example :

Assuming we are on a 32-bit system:

```
1  struct
2  {
3      unsigned int first : 9;
4      unsigned int second : 7;
5      unsigned int may_straddle : 30;
6      unsigned int last : 18;
7  } tricky_bits;
8
```

The Microsoft compiler stores each bit field in the above example so it fits completely in a single 32-bit integer.

Consider this example :

```
1  struct
2  {
3      unsigned int first : 9;
4      unsigned int second : 7;
5      unsigned int may_straddle : 30;
6      unsigned int last : 18;
7  } tricky_bits;
8
```

Assuming we are on a 32-bit system:

```
1  struct
2  {
3      unsigned int first : 9;          //9 bits
4      unsigned int second : 7;         //7 bits
5      unsigned int padding1 : 16;      //16 bits padding
6      unsigned int may_straddle : 30; //30 bits
7      unsigned int padding2 : 2;       //2 bits padding
8      unsigned int last : 18;          //18 bits
9      unsigned int padding3 : 14;      //14 bits padding
10 } tricky_bits;
```

In this case, first and second are stored in one integer, may_straddle is stored in a second integer, and last is stored in a third integer.

Total: 12 bytes

let’s meet

Readability and cache locality

Simplest method but may harm readability.
Considerations: Readability and cache locality

Readability

Code is not just for machines; it's also for human developers who need to understand, maintain, debug and extend it.

Maintaining coherent groups of related data can improve code readability.

Cache locality

Cache locality refers to the property of computer programs where they tend to access a small, specific set of data repeatedly in a short period of time.

Did You Know?

Cache is a small portion of main memory (RAM) set aside as a temporary storage area for frequently accessed data.

Cache locality

Accesses fitting within a cache line enhance performance.

Some of the structure's members to be split across different cache lines due to reordering.

CPU fetches main memory every now and then .

Overriding alignment rules

Padding may not be very useful to some file formats.
Padding for fast access can be overridden.

#pragma pack(n)

Overriding alignment rules

controls the alignment of structure members in memory.

The n in #pragma pack(n) represents the alignment value, which is usually in bytes.

for n=1, it means 1 word = 1 byte

which also means minimal or no padding between members

C operators alignof and alignas

alignof and alignas: Set and check alignment for variables.
_Alignof (alignof): Check existing alignment.
_Alignas (alignas): Set custom alignment.

```c
#include<stdio.h>
#include<stdalign.h>

// struct is aligned to 16 bytes in memory.
struct alignas(16) Demo
{

    int var1l; // 4 bytes
    int var2; // 4 bytes
    short s; // 2 bytes
    // char aligned to 4 bytes in memory.
    alignas(4) char arr[5];
};

// driver code
int main()
{
    printf(alignof(Demo)) // output: 16
    return 0;
}
```

```c
int main()
{
    int data;

    printf("Size in Bytes = %u\n",sizeof data);

    printf("Alignment Require = %u\n",alignof data);

    return 0;
}
```

Output:

```
Size in Bytes = 4
Alignment require = 4
```

```c
// Compile with /std:c11
#include <stdalign.h>
#include <stdio.h>
#include <stddef.h>


int main()
{
    int data[20];

    printf("Size in Bytes = %u\n",sizeof data);

    printf("Alignment Require = %u\n",alignof data);

    return 0;
}
```

Output:

```
Size in Bytes = 80
Alignment require = 4
```

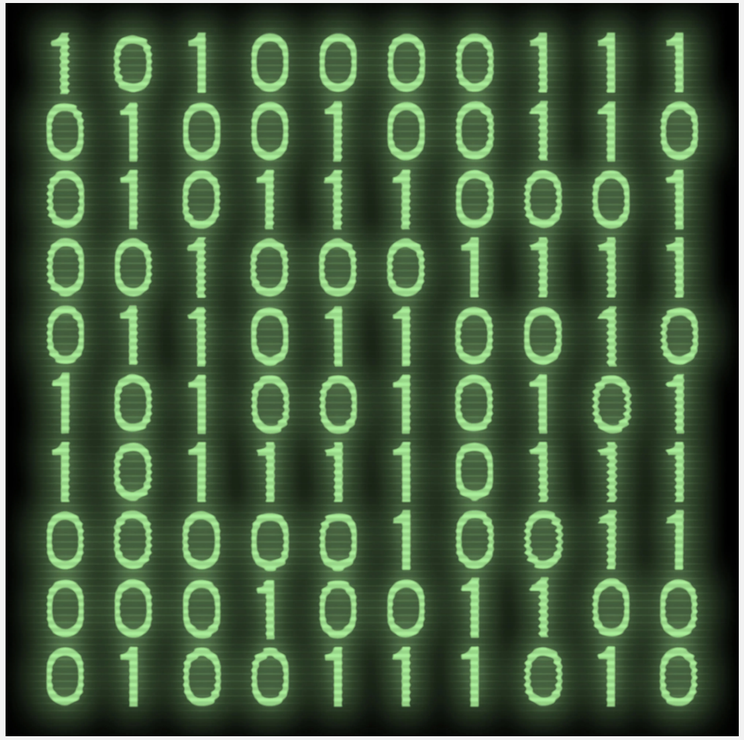
```c
// Compile with /std:c11
#include <stdalign.h>
#include <stdio.h>
#include <stddef.h>

typedef struct
{
    char a;
    int b;
    float c;
    double d;
} data;

int main()
{

    /*Alignment would be according to
    the largest element in the structure*/
    printf("alignof(data) = %d\n", alignof(data));

    //total number of bytes including padding bytes
    printf("sizeof(data) = %d\n", sizeof(data));

    return 0;
}
```

Output:

```
alignof(data) = 8
sizeof(data) = 24
```

Compilation

Source Code to Machine Code(.o or .obj file)
Transformation



Assembly code

Low-level representation of the program.
Human-readable but closely tied to machine instructions.

```
mov eax, 4
  mov ebx, 1
  mov ecx, message
  mov edx, 13
  int 0x80
```

MEMORY SEGMENTATION

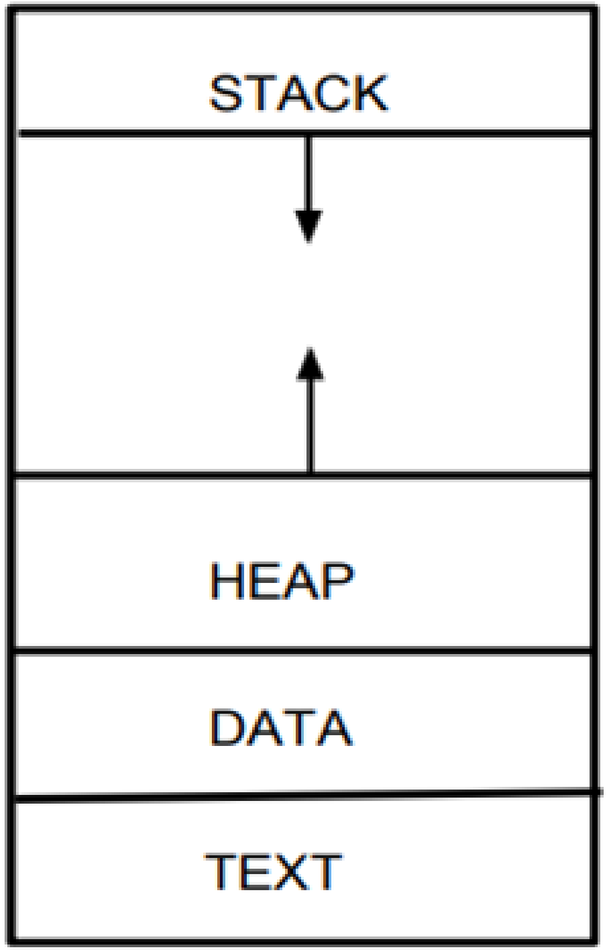
Memory

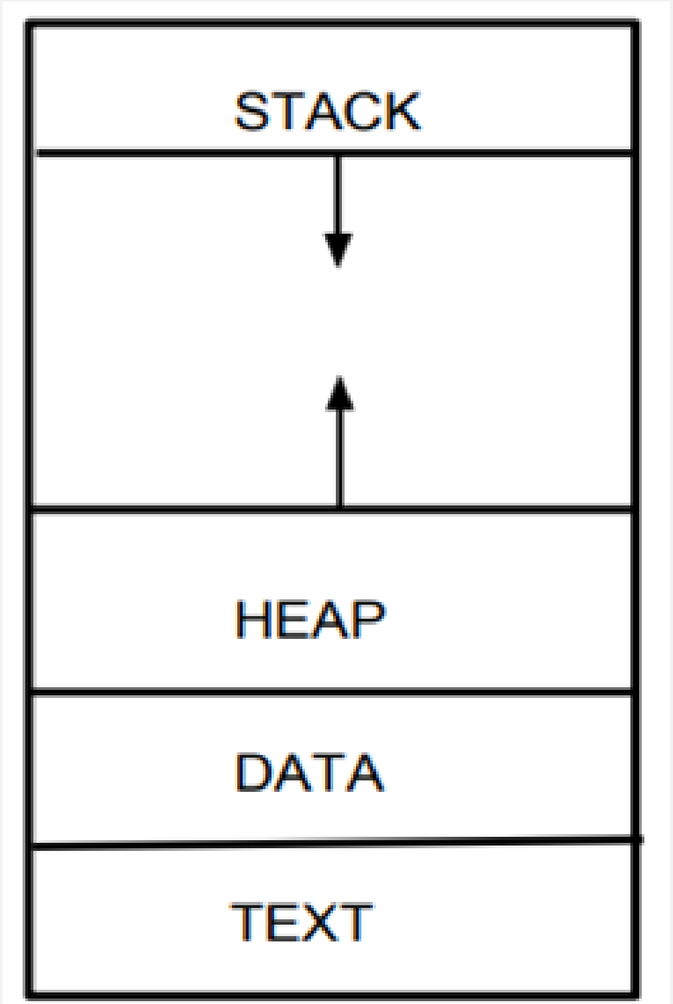
Memory segmentation of a program

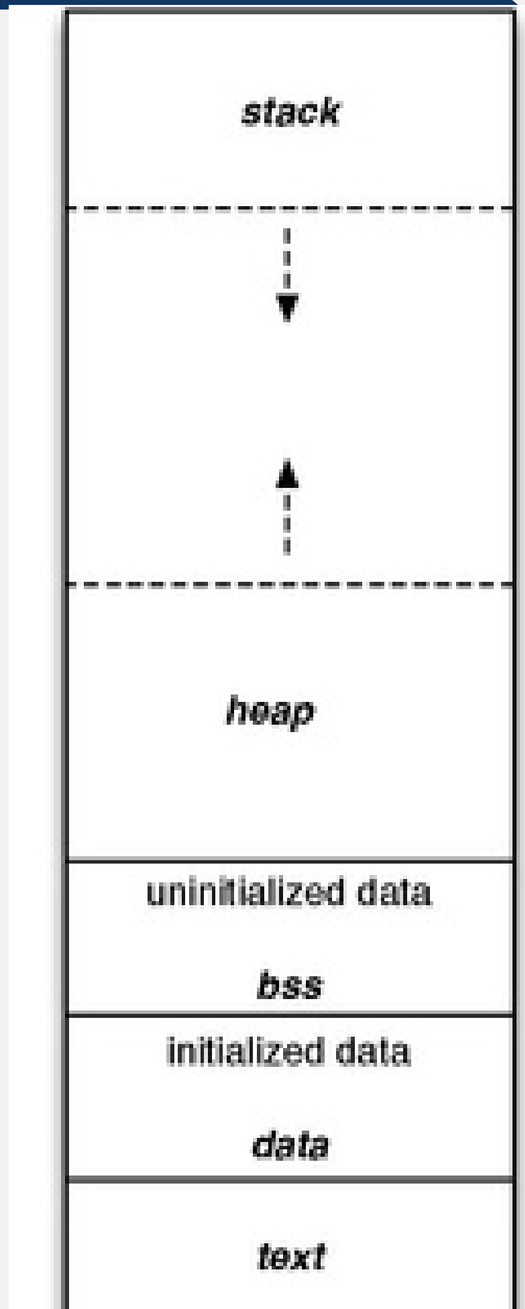
Memory segmentation of a program

When a program is written, it allocates virtual memory space in RAM, at various regions of memory to the program. Some of these regions correspond to segment, to hold different parts of the program.

Memory segmentation of a program

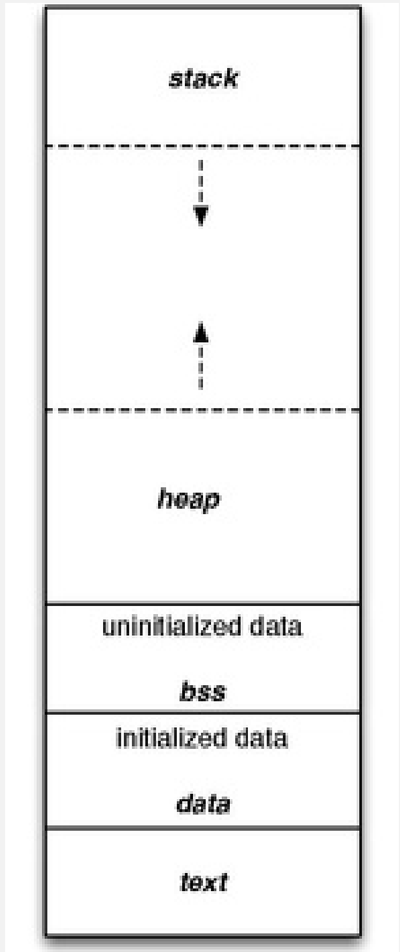



STACK
HEAP
DATA
TEXT

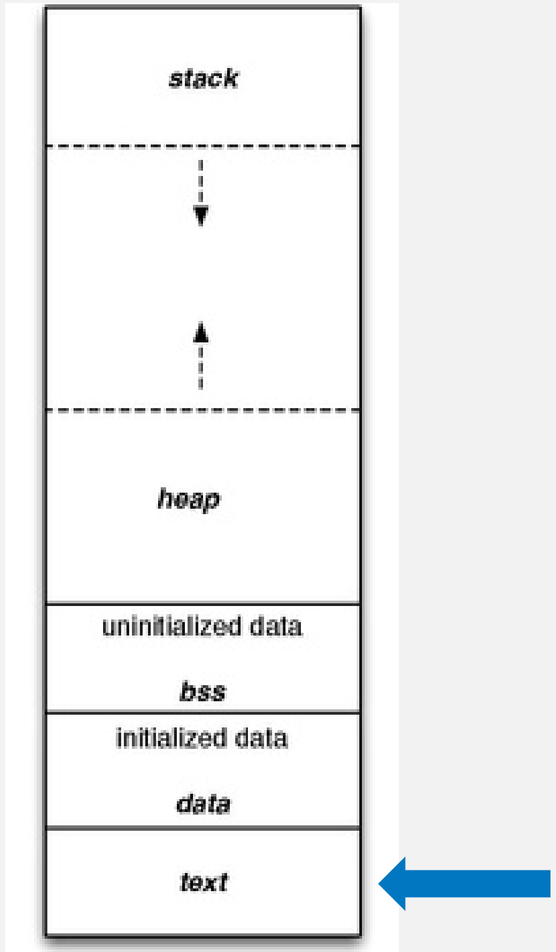

stack
heap
uninitialized data
bss
initialized data
data
text

Text or code segment

Text or code segment



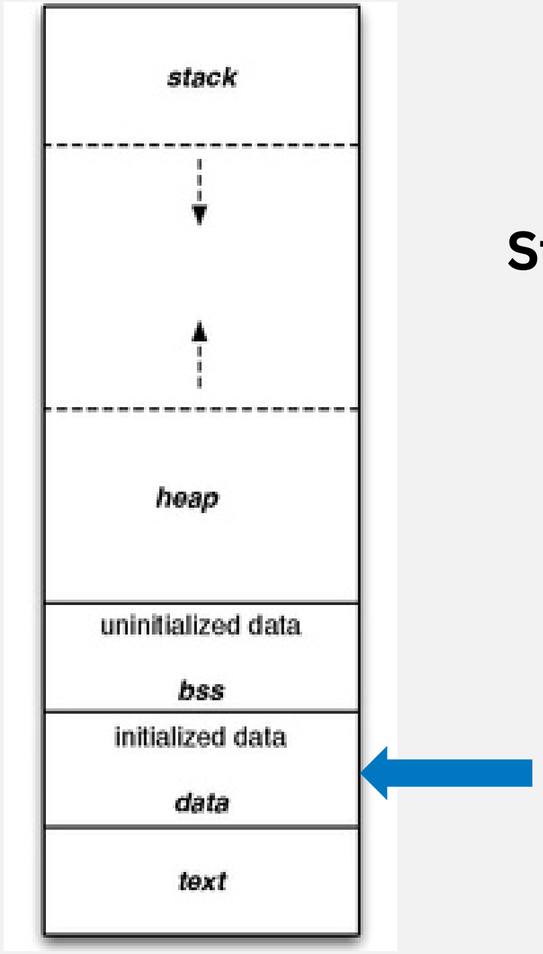
Text or code segment



Machine code of the object file
Typically read-only and fixed-size.
Often placed in ROM on embedded systems.

Initialized data segment

Initialized data segment



Stores all global, static, constant and external variables that are initialized beforehand.

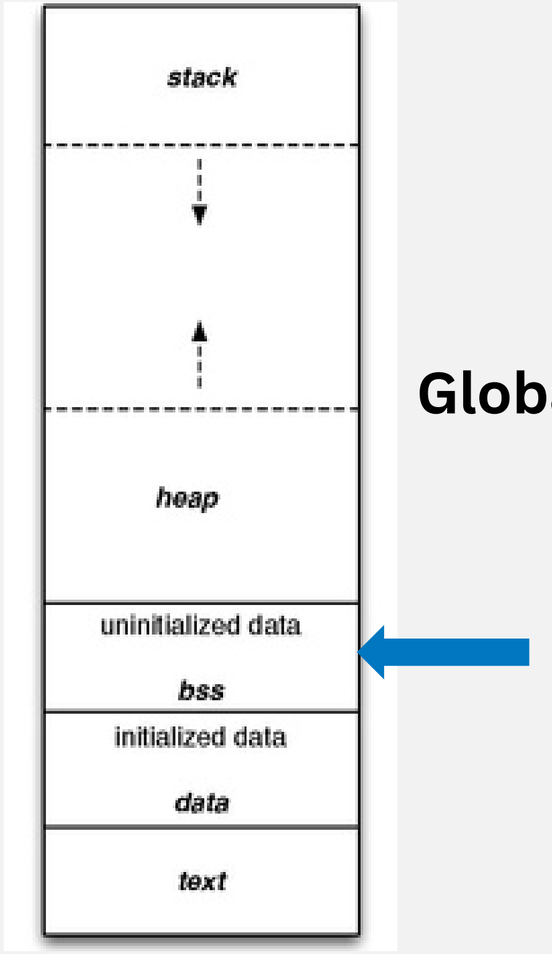
This segment can be further classified into:

- **initialized read-only area**
- **initialized read-write area**

```c
#include<stdio.h>
//global variable
//initialized read-write area
char c[]="majnu";

//initialized read-only area
const char s[]="laila";


int main()
{
    static int i=11;
    return 0;

}
```

Uninitialized data segment(bss)

Uninitialized data segment(bss)



Contains variables with no explicit initialization.
Global and static variables declared but not assigned a value.
Often initialized to arithmetic zero.

```c
#include<stdio.h>
//Uninitialized variable stored in bss
char c;

int main()
{
    static int i;
    return 0;
}
```

Stack

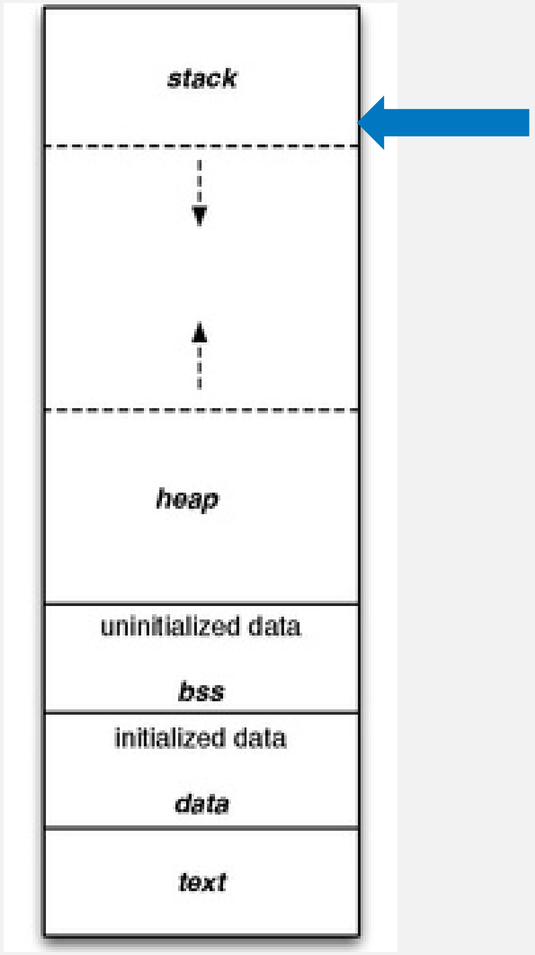
Works on a principal- a Last In, First Out (LIFO) structure.

"push" and "pop" operations.

Memory allocated by functions, function call

Local variables are allocated on the stack.

Stack



Adjacent to the heap; grows towards each other.

With large address spaces, they may grow more freely.

```c
#include<stdio.h>
//Uninitialized variable stored in bss
void add(void){
    printf("Enter the numbers you want to add: ");
    int a,b;
    scanf("%d%d",&a,&b);
    printf("The sum is %d",a+b);
    return;
}
int main()
{
    printf("Hello world");
    add();
    return 0;
}
```

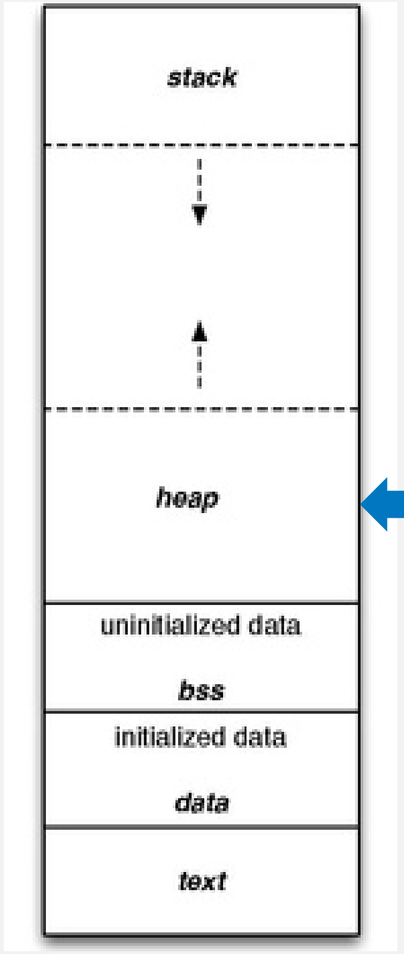
Heap

Free store of memory for “dynamic allocation”.

Dynamically allocated based on programmer's requests.

Not tied to scope; managed explicitly by the programmer.

Heap



Below the stack, beginning at the end of the BSS segment.

Grows upwards towards the stack.

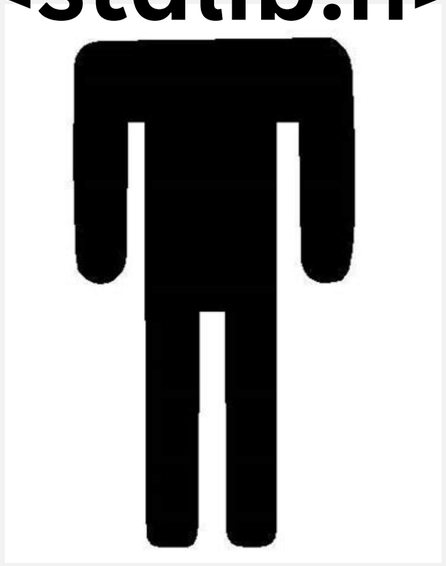
Dynamic memory allocation

Dynamic memory allocation

malloc(), realloc(), free()

Dynamic memory allocation

<stdlib.h>



malloc()

Syntax:

```
ptr = (castType*) malloc(size);
```

The malloc() function reserves a block of memory of the specified number of bytes.

Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    float *ptr;

    ptr= (float*)malloc(sizeof(float));

    ptr= (float*)malloc(10*sizeof(float));

}
```

realloc()

```
void *realloc(void *ptr, size_t size)
```

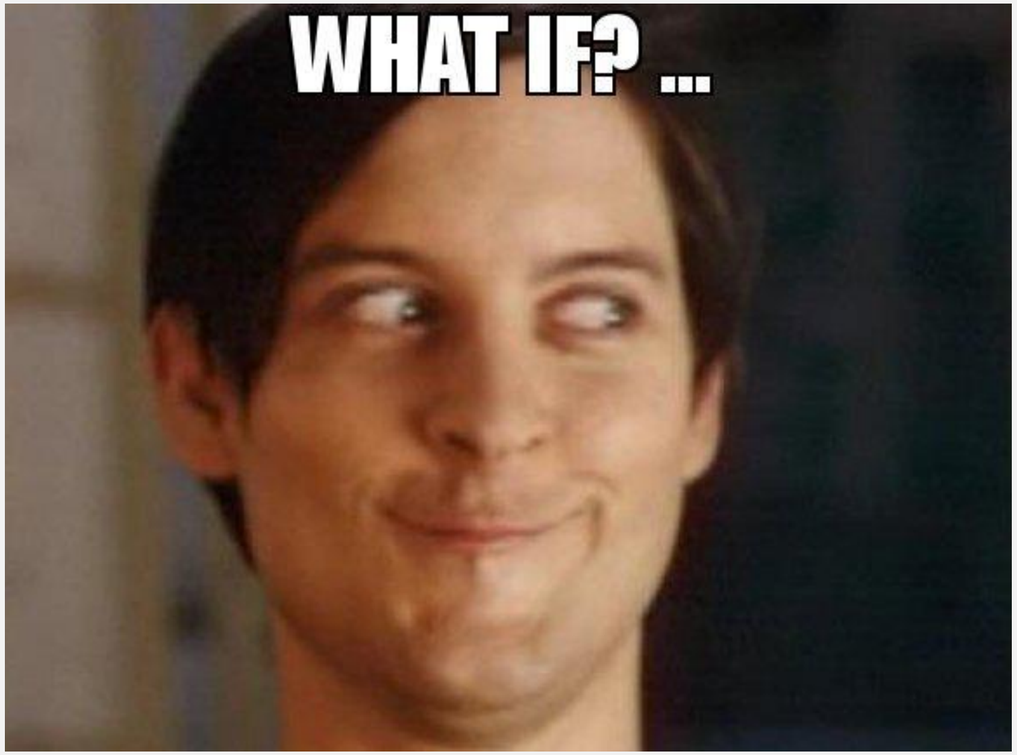
If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the realloc() function.

Example

```
ptr = realloc(ptr, n * sizeof(float));
```

realloc()

void *realloc(void *ptr, size_t size)



ptr is null

realloc()

void *realloc(void *ptr, size_t size)

In case a null pointer is passed, it acts just like malloc

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    float *ptr;
    ptr= (float*)malloc(10*sizeof(float));

    ptr= realloc(ptr, 5* sizeof(float));

}
```

free()

```
free(pointer_name)
```

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own.

You must explicitly use free() to release the space.

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    float *ptr;
    ptr= (float*)malloc(10*sizeof(float));

    ptr= realloc(ptr, 5* sizeof(float));
    // code.....

    free(ptr);

}
```

Pointer Alignment

Pointer Alignment

Self-alignment of pointers is crucial for efficient memory access.

Pointers returned by memory allocation functions (e.g., malloc, calloc) are self-aligned.

Pointer Alignment

```
char c;
int* ptr = &c;
```

Here, the cast involve converting from a less strict type to a more strict type.

If char c is not aligned to a 4-byte boundary (which is common for int), casting it to int* may lead to non-aligned access.

How does it occur?

Occurs when the memory address doesn't align with the size of the data type.

Non-aligned access can result in various issues:
Slower Access Speed
Program Crashes

THANK YOU