**Bit Fields**

# Bit Fields

* Bit Fields allow the packing of data in a structure.
* This is especially useful when memory or data storage is at a premium.

Struct {

type [member\_name] : width ;

};

The variables defined with a predefined width are called bit fields

Examples include

* Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
* Reading external file formats -- non-standard file formats could be read in,

e.g., 9-bit integers.

* C allows us to do this in a structure definition by putting :bit length after the variable.

Example

struct packed\_struct {

unsigned int f1:1;

unsigned int f2:1;

unsigned int f3:1;

unsigned int f4:1;

unsigned int type:4;

unsigned int my\_int:9;

} pack;

Here, the packed\_struct contains 6 members: Four 1 bit flags f1..f3, a 4-bit type and a 9-bit my\_int.

* C automatically packs the above bit fields as compactly as possible, provided that
* the maximum length of the field is less than or equal to the integer word length of the computer.
* If this is not the case, then some compilers may allow memory overlap for the fields while others would store the next field in the next word.

#include <stdio.h>

#include <string.h>

struct {

unsigned int age : 3;

} Age;

int main( ) {

Age.age = 4;

printf( "Sizeof( Age ) : %d\n", sizeof(Age) );

printf( "Age.age : %d\n", Age.age );

Age.age = 7;

printf( "Age.age : %d\n", Age.age );

Age.age = 8;

printf( "Age.age : %d\n", Age.age );

return 0;

}

O/P:

Sizeof( Age ) : 4

Age.age : 4

Age.age : 7

Age.age : 0

# Bit Fields in structure

* we can specify size (in bits) of structure and union members
* used for memory efficiently
* used when we know that the value of a field or group of fields

will never exceed a limit or is within a small range

// A simple representation of date

struct date {

unsigned int d;

unsigned int m;

unsigned int y;

};

Since we know that the value of d is always from 1 to 31,

value of m is from 1 to 12, we can optimize the space using bit fields.

#include <stdio.h>

// A space optimized representation of date

struct date {

// d has value between 1 and 31, so 5 bits are sufficient

unsigned int d: 5;

// m has value between 1 and 12, so 4 bits are sufficient

unsigned int m: 4;

unsigned int y;

};

int main() {

printf("Size of date is %d bytes\n", sizeof(struct date));

struct date dt = {31, 12, 2014};

printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);

return 0;

}

Output:

Size of date is 8 bytes

Date is 31/12/2014

# Interesting facts about bit fields

1. A special unnamed bit field of size 0 is used to force alignment on next boundary
2. We cannot have pointers to bit field members as they may not start at a byte boundary.
3. It is implementation defined to assign an out-of-range value to a bit field member.
4. In C++, we can have static members in a structure/class, but bit fields cannot be static.
5. Array of bit fields is not allowed.
6. There are no default member initializers for bit-fields: int b : 1 = 0; and int b : 1 {0} are ill-formed. (until C++20)
7. Example, the below program fails in compilation.

ex01:

#include <stdio.h>

// A structure without forced alignment

struct test1 {

unsigned int x: 5;

unsigned int y: 8;

};

// A structure with forced alignment

struct test2 {

unsigned int x: 5;

unsigned int: 0;

unsigned int y: 8;

};

int main() {

printf("Size of test1 is %d bytes\n", sizeof(struct test1));

printf("Size of test2 is %d bytes\n", sizeof(struct test2));

return 0;

}

Output:

Size of test1 is 4 bytes

Size of test2 is 8 bytes

ex02:

#include <stdio.h>

struct test {

unsigned int x: 5;

unsigned int y: 5;

unsigned int z;

};

int main() {

test t;

// Uncommenting the following line will make the program compile and run

// printf("Address of t.x is %p", &t.x); //CE: cannot take address of bit-field 'x'

// The below line works fine as z is not a bit field member

printf("Address of t.z is %p", &t.z);

return 0;

}

Output:

Address of t.z is 0x7ffcc3513c54

ex03:

#include <stdio.h>

struct test {

unsigned int x: 2;

unsigned int y: 2;

unsigned int z: 2;

};

int main() {

test t;

t.x = 5;

printf("%d", t.x);

return 0;

}

Output:

1

ex04:

#include <stdio.h>

struct test1 {

static unsigned int x;

};

// But below C++ program fails in compilation as bit fields cannot be static

struct test2 {

static unsigned int x: 5; //CE: static member 'x' cannot be a bit-field

};

int main() {

return 0;

}

ex05:

#include <stdio.h>

struct test2 {

unsigned int x[10]: 5; //In C, CE: bit-field 'x' has invalid type

//In C++, CE: function definition does not declare parameters

};

int main() {

return 0;

}

# Structure Member Alignment, Padding and Data Packing

## Data Alignment

* 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
* 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.

* if the integer is allocated at an address other than multiple of 4, it spans across two rows of the banks
* Such an integer requires two memory read cycle to fetch the data
* data alignment deals with the way the data stored in these banks

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.

## Structure Padding

* Structures are used as data pack
* It doesn’t provide any data encapsulation or data hiding features (C++ is an exception)
* 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.
* To Reduce Padding : programmer should declare the structure members in their increasing/decreasing order of size

A-4

B-8

C-16 X

C-24

D-16

## What is structure packing?

* Sometimes 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

* Need to define a structure similar to that of the header layout and map it.
* care should be exercised in accessing such members
* 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

## Pointer Mishaps

* There is possibility of potential error while dealing with pointer arithmetic

example, dereferencing a generic pointer (void \*) can cause misaligned exception,

* + // Deferencing a generic pointer (not safe)
  + // There is no guarantee that pGeneric is integer aligned
  + \*(int \*)pGeneric;
* If the pointer pGeneric is not aligned as per the requirements of casted data type,

there is possibility to get misaligned exception.

* 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.

## malloc() returned pointer

pointer returned by malloc() is void \*

can be converted to any data type as per the need of programmer

malloc() should return a pointer that is aligned to maximum size of primitive data types

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

# General Questions

## Is alignment applied for stack?

Yes. The stack is also memory.

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.

Example,

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

## 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

Depending on the bank it is stored, the processor shifts the byte onto least significant data lines.

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

Yes. The compiler helps programmer in making proper alignment.

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

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.

## 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

few processors will not have last two address lines, which means there is no-way to access odd byte boundary

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

Yes. Compilers provide non standard extensions for such needs

example, \_\_alignof() in Visual Studio helps in getting the alignment requirements of data type

## 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?

* 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 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

# References

<https://www.geeksforgeeks.org/bit-fields-c/>

<https://en.cppreference.com/w/cpp/language/bit_field>

# END