Size of Structures

Padding and Alignment of structure [1]

```
struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
    char name[6] ; // name of the point
} ;

int main( int argc, char* argv[] )
{
    struct point  pt ;
    struct point  maxpt = { 20, 30, "Earth" } ;

    pt.x = 4 ; // set x component of point pt as 4
    pt.y = 3 ; // set y component of point pt as 3
    strcpy( pt.name, "Venus") ; // set name of pt

    printf("pt = (%d. %d. %s )\n". pt.x . pt.u. pt.name ):
```

Context: main(int, cha

Name	Value
argc	1
⊞ argv	0x003720a(
⊞ maxpt	{...}
⊞ pt	{...}

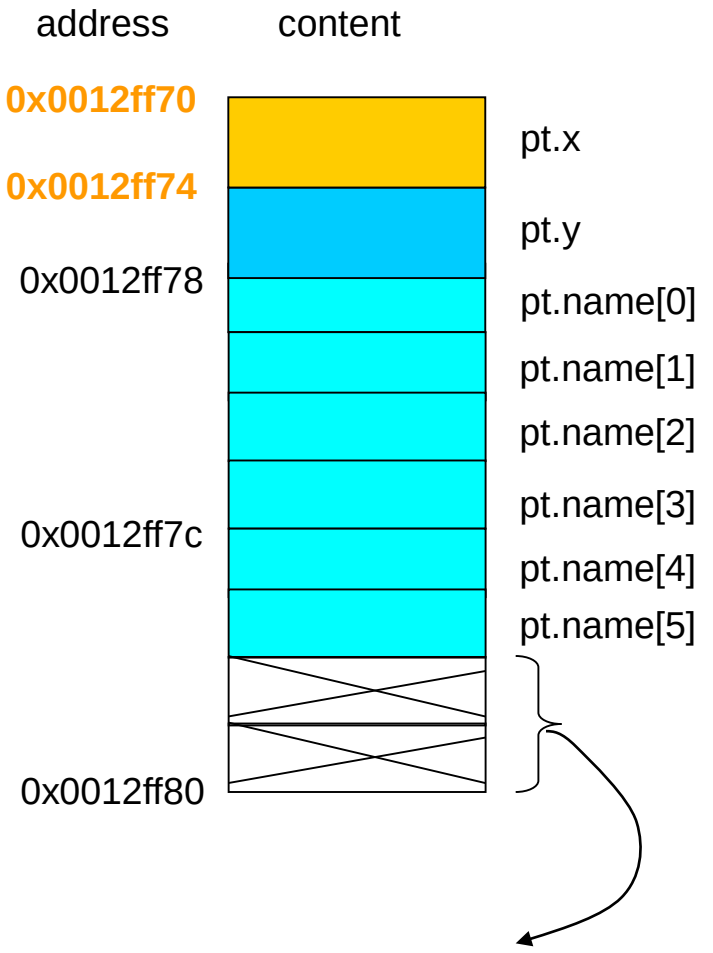
main(
mainC
KERNE

Name	Value
⊞ &pt	0x0012ff70
⊞ &pt.x	0x0012ff70
⊞ &pt.y	0x0012ff74
⊞ &pt.name	0x0012ff78
⊞ &maxpt	0x0012ff60
⊞ &maxpt.x	0x0012ff60
⊞ &maxpt.y	0x0012ff64
⊞ &maxpt.name	0x0012ff68

```
"F:\COURSE\2008SUMMER\C_LANG\E
pt = (4, 3, Venus )
maxpt = (20, 30, Earth )
size of structure point = 16
Press any key to continue
```

size of structure point

!= 14 (4+4+6)



Two bytes Padding by compiler

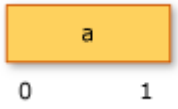
Padding and Alignment of structure [2]

- The padding and alignment of members of structures and whether a bit field can straddle a storage-unit boundary.
- Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest
- Every data object has an alignment-requirement. For structures, the alignment-requirement is the largest alignment-requirement of its members. Every object is allocated an offset so that
  offset **%** *alignment-requirement* **==** 0
- When you use the /Zp[*n*] option, where *n* is 1, 2, 4, 8, or 16, each structure member after the first is stored on byte boundaries that are either the alignment requirement of the field or the packing size (*n*), default is 4.

Padding and Alignment of structure [3]

```c
#include <stdio.h>

// word = 2 bytes, doubleword = 4 bytes, quadword = 8 bytes
struct S1 {
    short a ;  // size = 2 bytes, alignment = 2 bytes;
} ;

struct S2 {  // size = 24 bytes, alignment = quadword
    int   a ;
    double b ;
    short c ;
} ;

struct S3 {  // size = 12 bytes, alignment = doublewor
    char  a ;
    short b ;
    char  c ;
    int   d ;
} ;

int main( int argc, char* argv[] )
{
    struct S1  x ;
    struct S2  y ;
    struct S3  z ;
    printf("size of struct S1 = %d\n", sizeof(struct S1) ) ;
    printf("size of struct S1 = %d\n", sizeof(struct S2) ) ;
    printf("size of struct S1 = %d\n", sizeof(struct S3) ) ;
    return 0 ;
}
```

Structure S1



Structure S2

Structure S3

Example from MSDN Library

Padding and Alignment of structure [4]

suggested alignment for the scalar members of unions and structures

from MSDN Library

Scalar Type	C Data Type	Required Alignment
INT8	**char**	Byte
UINT8	**unsigned char**	Byte
INT16	**short**	Word
UINT16	**unsigned short**	Word
INT32	**int, long**	Doubleword
UINT32	**unsigned int, unsigned long**	Doubleword
INT64	**__int64**	Quadword
UINT64	**unsigned __int64**	Quadword
FP32 (single precision)	**float**	Doubleword
FP64 (double precision)	**double**	Quadword
POINTER	*	Quadword

Padding and Alignment of structure [5]

alignment rules

- The alignment of an array is the same as the alignment of one of the elements of the array.
- The alignment of the beginning of a structure is the maximum alignment of any individual member. Each member within the structure must be placed at its proper alignment as defined in the previous table, which may require implicit internal padding, depending on the previous member.
- Structure size must be an integral multiple of its alignment.
- It is possible to align data in such a way as to be greater than the alignment requirements as long as the previous rules are maintained.
- An individual compiler may adjust the packing of a structure for size reasons.