Structs

- A collection of variables of possibly different types, laid out contiguously in memory
- The struct has to be aligned internally and externally depending on what data types it contains. This means that the variables may not be laid out in memory immediately next to each other - there may be padding bytes between them
- Internal alignment - each variable has to start at an address that's a multiple of its data type size
- External alignment - the struct has to begin and end at an address that's a multiple of the largest integral type in the struct. This may involve adding padding bytes after the last variable in the struct has been laid out

Unions

- A data structure that stores variables, possibly of different types, at the same memory location
- This means that modifying the value of one variable may affect the value of another variable, because their memories overlap
- Because all variables are stored at the same memory location, there is no notion of internal alignment
- However, unions, like structs, have external alignment. The rule is the same : the union has to begin and end at an address that's a multiple of the largest integral type in the union

When we say that a union/struct has to be aligned to the largest integral type, this exclusively refers to a C data type. So if the union/struct has as one of its variables an array/union/struct, it will NOT be aligned to the size of the array/union/struct - it will be aligned to the largest C data type present within the union/struct

Dealing with internal alignment when a variable is followed by a union/struct : Align to largest integral type present within the union/struct (even if that integral type is present multiple levels deep within the union/struct, where each level is defined by the definition of a new nested union/struct)

Useful tips :

1. Deal with internal alignment first, and then at the end, external alignment
2. Work inside-out when dealing with nested structs/unions

Practice Problems

Find the size of each of the following unions or structs

```
1. struct {
      char ch_arr[5];
      int i;
      float f;
      char ch;
      long l;
   } s;

2. struct {
      struct {
        char ch_array[51];
       } s2;
      int i;
  } s1;

3. struct {
      int i;
      struct {
         char ch;
         long l;
      } s2;
   } s1;

4. struct {
      char ch;
      struct {
         int i;
         struct {
           long l;
         } s3;
      } s2;
   } s1;

5. union {
      char ch_array[9];
      int i;
   } u;
```

6.
```
union {
  char ch_array[10];
  struct{
    int i;
   } s;
} u
```

7.
```
union {
  long l;
  struct {
    char ch;
    short sh_array[5];
  } s;
} u;
```

8.
```
union {
  char ch_array[5];
  struct {
    int i;
    struct {
      char ch;
      long l;
    } s2;
  } s1;
} u;
```

9.
```
 struct {
   char ch_array[5];
   union {
     char ch_array[15];
     long l;
   } u[3];
  float f;
 } s[7];
```

```
10. union {
      char ch_array[70];
      struct {
        char ch;
        int i;
        short s;
        struct {
          char ch_array[5];
          union {
            char ch_array[15];
            long l;
          } u2[3];
          float f;
        } s2[7];
      } s1;
    } u1;
```

Solutions

```
1. struct {
      char ch_arr[5];
      int i;
      float f;
      char ch;
      long l;
    } s;
Ans : 5 + (3) + 4 + 4 + 1 + (7) + 8 = 32
```

- ch_array takes 5 bytes
- 3 bytes of padding before i so i can start at 8, a multiple of 4
- i takes 4 bytes
- f takes 4 bytes
- ch takes 1 byte
- 7 bytes of padding before l so l can start at 24, a multiple of 8
- l takes 8 bytes

```
2. struct {
     struct {
       char ch[51];
     } s2;
```

```
    int i;
  } s1;
```

Ans : 51 + (1) + 4 = 56

- s1 takes 51 bytes
- 1 byte of padding before i
- i takes 4 bytes

```
3. struct {
     int i;
     struct {
        char ch;
        long l;
     } s2;
   } s1;
```

Ans : 4 + (4) + 1 + (7) + 8 = 24

- s2 :
- ch takes 1 byte
- 7 bytes of padding
- l takes 8 bytes
- size of s2 : 16 bytes
- s1 :
- i takes 4 bytes
- 4 bytes of padding before s2 because s2 contains a long
- s2 takes 16 bytes

```
4. struct {
     char ch;
     struct {
        int i;
        struct {
          long l;
        } s3;
     } s2;
  } s1;
```

Ans : 1 + (7) + 4 + (4) + 8 = 24

- s3 : 8 bytes
- s2 :
- i takes 4 bytes
- 4 bytes of padding before s3 because s3 contains a long
- s1 :
- ch takes 1 byte
- 7 bytes of padding before s2 because s2 contains a long (in s3)

5. union {
     char ch_array[9];
     int i;
   } u;

Ans : 9 + (3) = 12

- The largest variable is ch_array which takes 9 bytes, so the union needs at least 9 bytes
- But the union also contains an int, so due to external alignment, 3 bytes of padding are added at the end

6. union {
     char ch_array[10];
     struct{
       int i;
      } s;
   } u

Ans : 10 + (2) = 12

- The largest variable is ch_array with takes 10 bytes
- 2 bytes of padding at the end because the largest integral type within the union is the int, even though it's a level deeper

7. union {
     long l;
     struct {
       char ch;
       short sh_array[5];
     } s;
   } u;

Ans : 1 + (1) + 10 + (4) = 16

- s :
- ch takes 1 byte
- 1 byte of padding before sh_array
- sh_array takes 10 bytes
- size of s : 12 bytes
- The largest variable is s which takes 12 bytes
- 4 bytes of padding at the end because of the long

8.
```
union {
    char ch_array[5];
    struct {
      int i;
      struct {
        char ch;
        long l;
      } s2;
    } s1;
  } u;
```

Ans : 4 + (4) + 1 + (7) + 8 = 24

- s2 : 1 + (7) + 8 = 16 bytes
- s3 :
- i takes 4 bytes
- 4 bytes of padding before s2 because s2 contains a long
- s2 takes 16 bytes
- size of s2 : 24 bytes
- The largest variable is s1 which takes 24 bytes
- The union has the same size

9.
```
struct {
     char ch_array[5];
     union {
       char ch_array[15];
       long l;
     } u[3];
     float f;
   } s[7];
```

Ans : (5 + (3) + (15 + (1))*3 + 4 + (4))*7 = 448

- u :
- The largest variable is ch_array which takes 15 bytes
- 1 byte of padding at the end of the u because of the long
- size of u : 16 bytes
- u[3] takes 3 * 16 = 48 bytes
- ch_array takes 5 bytes
- 3 bytes of padding before u[3] because the union contains a long
- u[3] takes 48 bytes
- f takes 4 bytes
- 4 bytes of padding at the end because s contains a long
- size of s : 64 bytes

- size of s[7] = 64 * 7 = 448

10.
```
union {
     char ch_array[70];
     struct {
       char ch;
       int i;
       short s;
       struct {
         char ch_array[5];
         union {
           char ch_array[15];
           long l;
         } u2[3];
         float f;
       } s2[7];
     } s1;
   } u1;
```

Ans : 1 + (3) + 4 + 2 + (6) + (5 + (3) + (15 + (1))*3 + 4 + (4))*7 = 464

- From Q9 we know that s2[7] takes 448 bytes
- s1 :
- ch takes 1 byte
- 3 bytes of padding before i
- i takes 4 bytes
- s takes 2 bytes
- 6 bytes of padding before s2[7] because s2 contains a long
- s2[7] takes 448 bytes
- size of s1 : 464 bytes
- The largest variable in u is s1 which takes 464 bytes
- The union has the same size