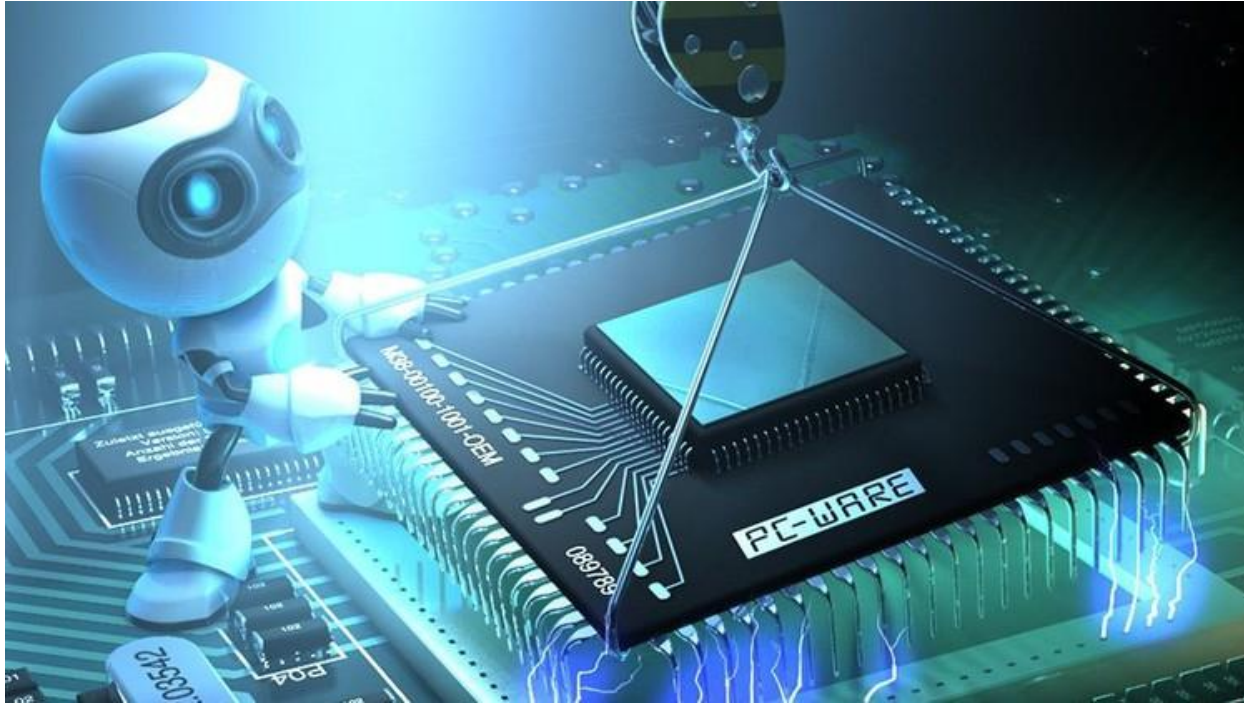


Data Alignment



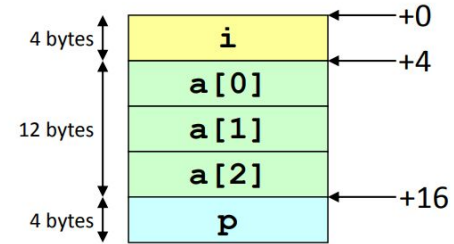
Structures (Structs)



- **Heterogeneous data set:**
 - contiguously stored in memory
 - referenced by name

Example:

```
typedef struct {  
    int i;  
    int a[3];  
    int *p;  
} X;  
X S;  
  
Init(&S);
```



Example:

```
void Init (X *S) {  
    (*S).i = 1;  
    S->a[2] = 0;  
    S->p = &(*S).a[0];  
}
```

Structures (Struct)



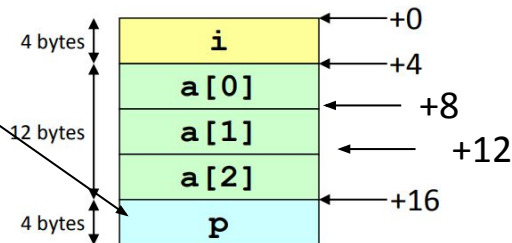
Example:

```
typedef struct {  
    int i;  
    int a[3];  
    int *p;  
} X;  
X S;  
Init(&S);
```

Traduction:

```
Init: push %ebp  
      movl %esp,%ebp  
      movl 8(%ebp),%edx  
      movl $1, (%edx)  
      movl $0,12(%edx)  
      leal 4(%edx),%eax  
      movl %eax,16(%edx)  
      popl %ebp  
      ret
```

24(%ebp)



Example:

```
void Init (X *S) {  
    (*S).i = 1;  
    S->a[2] = 0;  
    S->p = &(*S).a[0];  
}
```

Data Alignment



- **Data Alignment**
 - A primitive data type requires k bytes
 - The **address** must be a multiple of k

Data Alignment



- **Data Alignment**
 - A primitive data type requires k bytes
 - The **address** must be a multiple of k
- **Motivation to align data**
 - Memory accesses by aligned longword or quadwords
 - Unaligned accesses can cause the same data to be found in 2 different cache lines
 - Virtual memory: problems if the data is between two pages

Data Alignment



- **Data Alignment**
 - A primitive data type requires k bytes
 - The **address** must be a multiple of k
- **Motivation to align data**
 - Memory accesses by aligned longword or quadwords
 - Unaligned accesses can cause the same data to be found in 2 different cache lines
 - Virtual memory: problems if the data is between two pages
- **Compiler**
 - Inserts “spaces” in the structure to ensure that the data is aligned.

Data Alignment



- **Alignment in linux-32 (gcc):**
 - **char** (1 byte): 1-byte aligned (no restrictions on the @)
 - **short** (2 bytes): 2-byte aligned (lowest bit of the @ must be 0)
 - **int** (4 bytes): 4-byte aligned (the lower 2 bits of the @ must be 00)
 - **pointer** (4 bytes): 4-byte aligned
 - **double** (8 bytes): 4-byte aligned
 - **Long double** (12 bytes): aligned to 4-bytes
- **Offsets within a structure:**
 - must satisfy the alignment requirements of their elements
- **Structure direction**
 - Each structure has an alignment requirement k
 - k = the largest of the alignments of any element
 - The initial @ and the size of the structure must be a multiple of k

Data Alignment



- **Linux-64 differences:**
 - **double** (8 bytes): 8-byte aligned.
 - **long double** (16 bytes): aligned to 16-bytes.
 - **pointer** (8 bytes): 8-byte aligned.
- **Windows-32 differences:**
 - **double** (8 bytes): 8-byte aligned.
 - **long double** (10 bytes): aligned to 2-bytes.

Data Alignment: Example



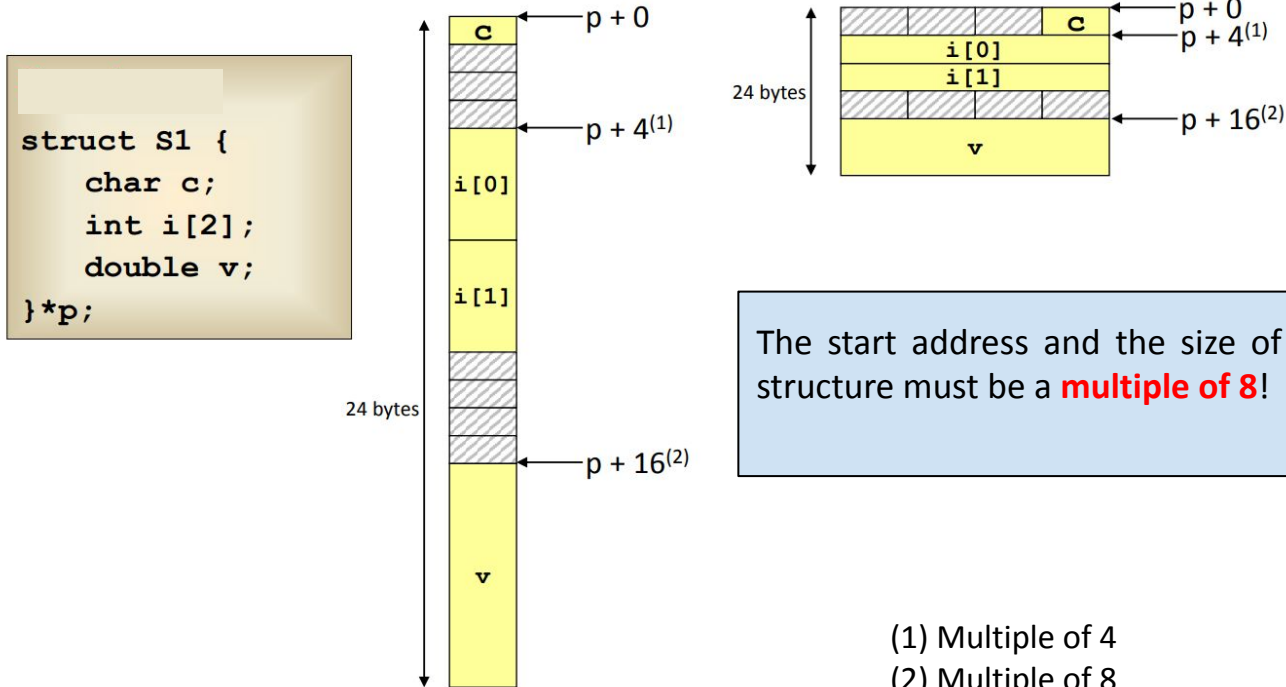
```
struct S1 {  
    char c;    // aligned to 1  
    int i[2];  // aligned to 4  
    double v;  // aligned to 4 (8 in Linux-64)  
} *p;
```

- (Linux-32) $k = \max(1, 4, 4) = 4$
- (Linux-64) $k = \max(1, 4, 8) = 8$

Data Alignment: Example in Linux-64



- **k = 8** due to the *double* element

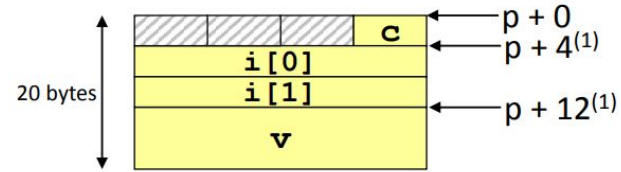
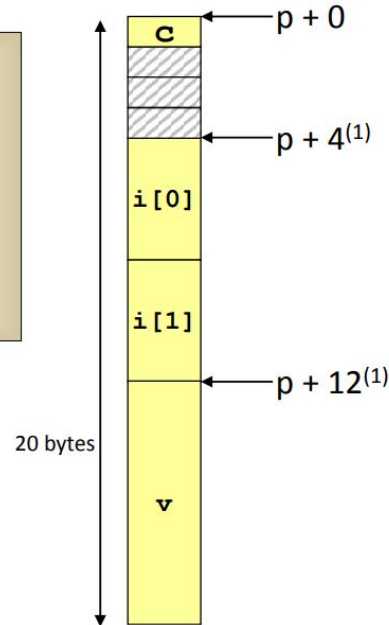


Data Alignment: Example in Linux-32



- **k = 4** due to the *double* element is aligned to 4 bytes

```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
}*p;
```



The start address and the size of the structure must be a **multiple of 4**!

(1) Multiple of 4

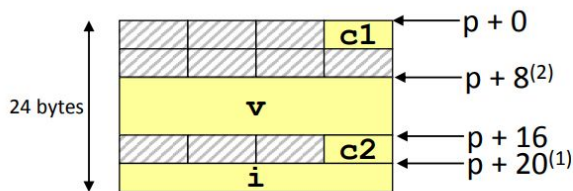
Alignment and Order of the Elements



- The order of the elements of a structure **influences** its **size**.

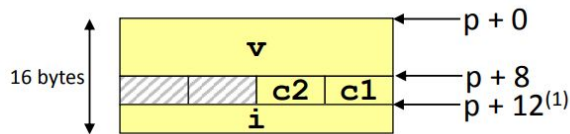
Example in Linux-64

```
struct S4 {  
    char c1;  
    double v;  
    char c2;  
    int i;  
} *p;
```



- (1) Multiple of 4
(2) Multiple of 8

```
struct S5 {  
    double v;  
    char c1;  
    char c2;  
    int i;  
} *p;
```



The start address and the size of the structure must be a **multiple of 8**!