Unions

Unions are derived data types, the way structures are. But Unions have the same relationship to structures that you might have with a distant cousin who resembled you but turned out to be smuggling contraband in Mexico. That is, unions and structures look alike, but are engaged in totally different enterprises.

Both structures and unions are used to group a number of different variables together. But while a structure enables us treat a number of different variables stored at different places in memory, a union enables us to treat the same space in memory as a number of different variables. That is, a union offers a way for a section of memory to be treated as a variable of one type on one occasion, and as a different variable of a different type on another occasion.

You might wonder why it would be necessary to do such a thing, but we will be seeing several very practical applications of unions soon. First, let us take a look at a simple example:

```
/* Demo of union at work */
# include <stdio.h>
void main( )
{
    union a
    {
```

```
        int i ;
        char ch[2] ;
    } ;
    union a key ;

    key.i = 512 ;
    printf ( "\nkey.i = %d", key.i ) ;
    printf ( "\nkey.ch[0] = %d", key.ch[0] ) ;
    printf ( "\nkey.ch[1] = %d", key.ch[1] ) ;
}
```

And here is the output...

```
key.i = 512
key.ch[0] = 0
key.ch[1] = 2
```

As you can see, first we declared a data type of the type **union a**, and then a variable **key** to be of the type **union a**. This is similar to the way we first declare the structure type and then the structure variables. Also, the union elements are accessed exactly the same way in which the structure elements are accessed, using a '.' operator. However, the similarity ends here. To illustrate this let us compare the following data types:

```
struct a
{
    int i ;
    char ch[2] ;
} ;
struct a key ;
```

This data type would occupy 4 bytes in memory, 2 for **key.i** and one each for **key.ch[0]** and **key.ch[1]**, as shown in Figure 15.1.

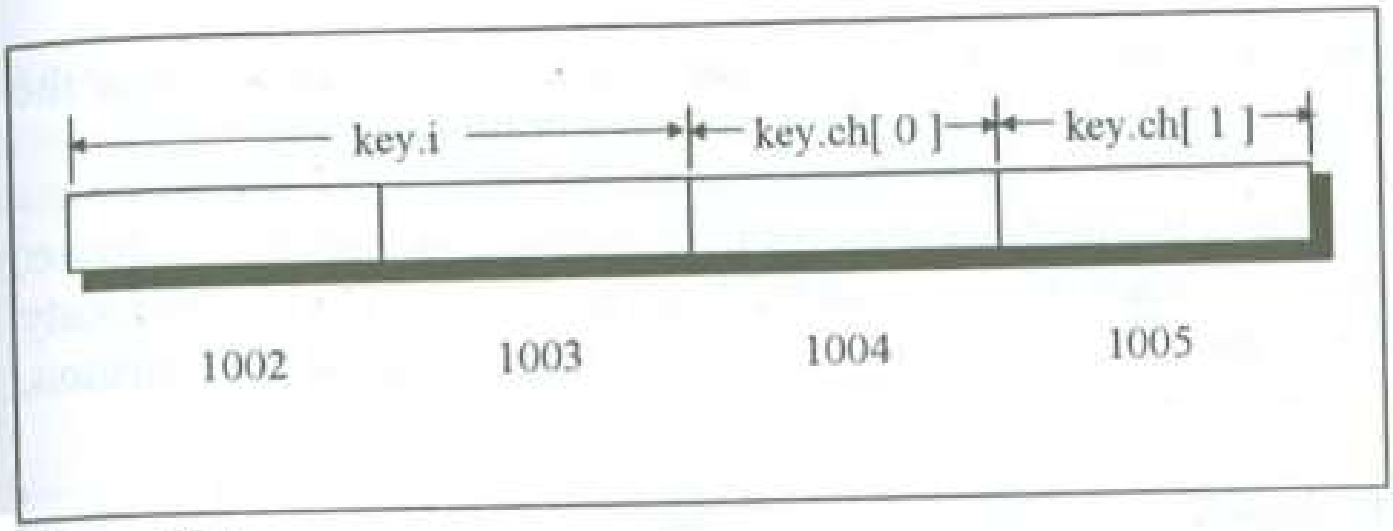

Figure 15.1

Now we declare a similar data type, but instead of using a structure we use a union.

```
union a
{
    int i ;
    char ch[2] ;
};
union a key ;
```

Representation of this data type in memory is shown in Figure 15.2.

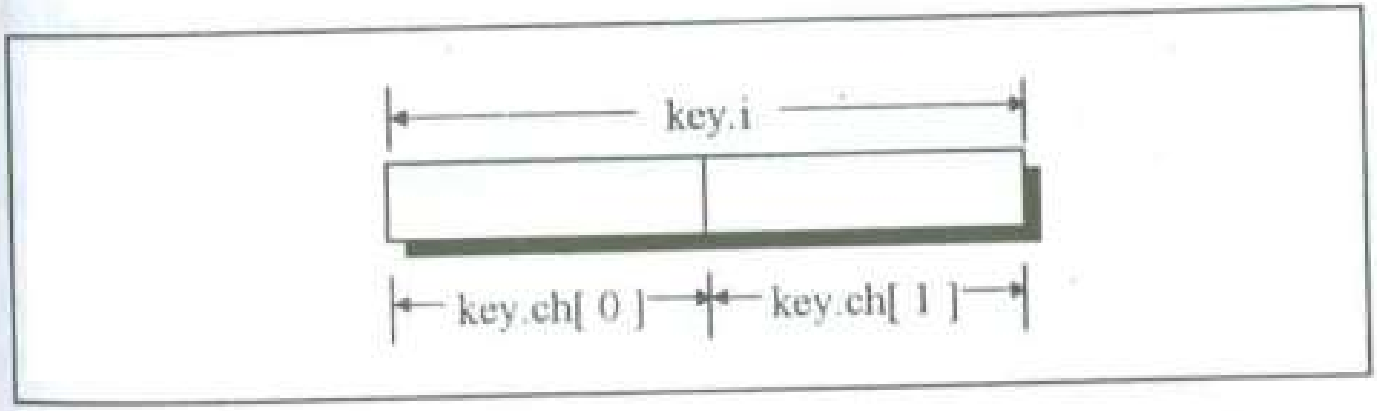

Figure 15.2

As shown in Figure 15.2, the union occupies only 2 bytes in memory. Note that the same memory locations which are used for **key.i** are also being used by **key.ch[0]** and **key.ch[1]**. It means that the memory locations used by **key.i** can also be accessed using **key.ch[0]** and **key.ch[1]**. What purpose does this serve? Well, now

we can access the two bytes simultaneously (by using **key.i**) or the same two bytes individually (using **key.ch[0]** and **key.ch[1]**).

This is a frequent requirement while interacting with the hardware. i.e. sometimes we are required to access two bytes simultaneously and sometimes each byte individually. Faced with such a situation, using union is the answer, usually.

Perhaps you would be able to understand the union data type more thoroughly if we take a fresh look at the output of the above program. Here it is...

```
key.i = 512
key.ch[0] = 0
key.ch[1] = 2
```

Let us understand this output in detail. 512 is an integer, a 2 byte number. Its binary equivalent will be 0000 0010 0000 0000. We would expect that this binary number when stored in memory would look as shown below.

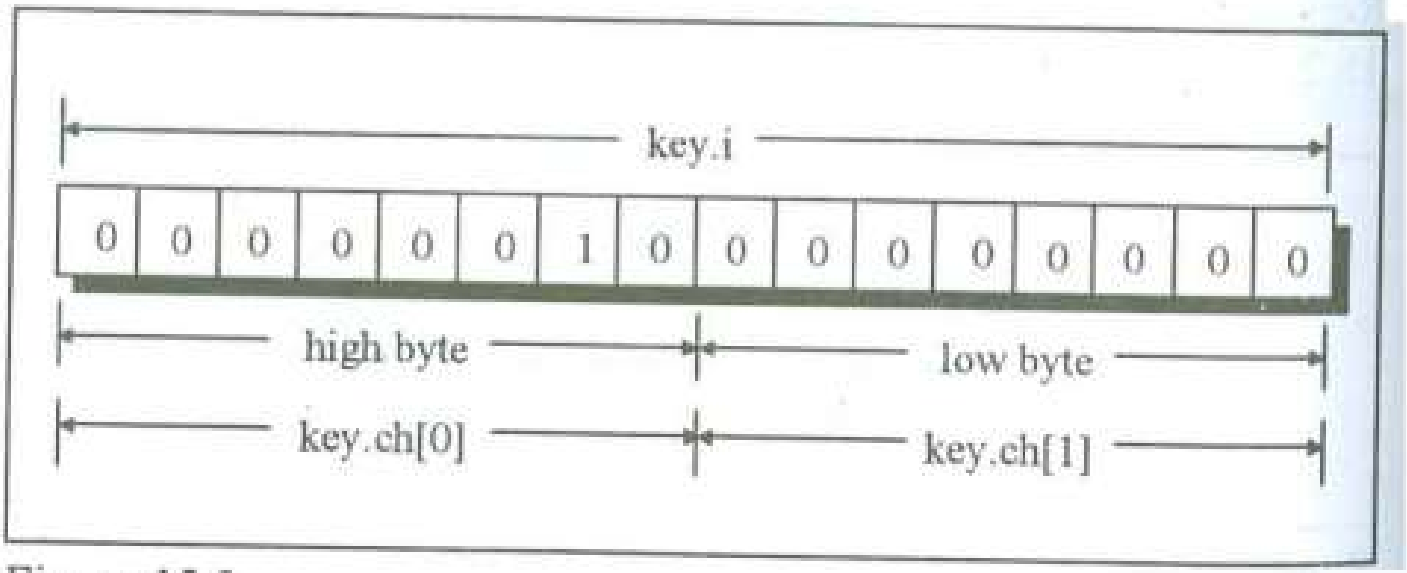

Figure 15.3

If the number is stored in this manner, then the output of **key.ch[0]** and **key.ch[1]** should have been 2 and 0. But, if you look at the output of the program written above, it is exactly the opposite. Why is it so? Because, in CPUs that follow little-endian

architecture (Intel CPUs, for example), when a two-byte number is stored in memory, the low byte is stored before the high byte. It means, actually 512 would be stored in memory as shown in Figure 15.4. In CPUs with big-endian architecture this reversal of bytes does not happen.

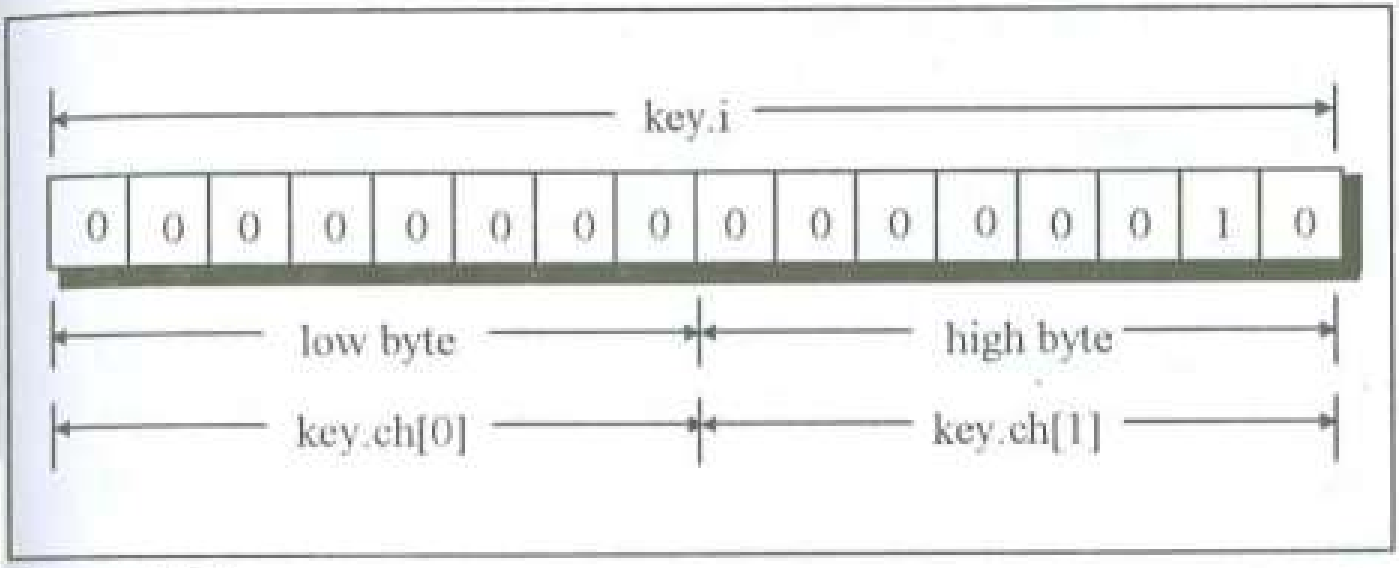

Figure 15.4

Now, we can see why value of **key.ch[0]** is printed as 0 and value of **key.ch[1]** is printed as 2.

One last thing. We can't assign different values to the different union elements at the same time. That is, if we assign a value to **key.i**, it gets automatically assigned to **key.ch[0]** and **key.ch[1]**. Vice versa, if we assign a value to **key.ch[0]** or **key.ch[1]**, it is bound to get assigned to **key.i**. Here is a program that illustrates this fact.

```
#include <stdio.h>
void main( )
{
    union a
    {
        int i ;
        char ch[2] ;
    } ;
    union a key ;
```

```
    key.i = 512 ;
    printf ( "\nkey.i = %d", key.i ) ;
    printf ( "\nkey.ch[0] = %d", key.ch[0] ) ;
    printf ( "\nkey.ch[1] = %d", key.ch[1] ) ;

    key.ch[0] = 50 ; /* assign a new value to key.ch[0] */
    printf ( "\nkey.i = %d", key.i ) ;
    printf ( "\nkey.ch[0] = %d", key.ch[0] ) ;
    printf ( "\nkey.ch[1] = %d", key.ch[1] ) ;
}
```

And here is the output...

```
key.i = 512
key.ch[0] = 0
key.ch[1] = 2
key.i= 562
key.ch[0] = 50
key.ch[1] = 2
```

Before we move on to the next section, let us reiterate that a union provides a way to look at the same data in several different ways. For example, there can exist a union as shown below.

```
union b
{
    double d ;
    float f[2] ;
    int i[4] ;
    char ch[8] ;
} ;
union b data ;
```

In what different ways can the data be accessed from it? Sometimes, as a complete set of eight bytes (**data.d**), sometimes as two sets of 4 bytes each (**data.f[0]** and **data.f[1]**), sometimes as

four sets of 2 bytes each (**data.i[0]**, **data.i[1]**, **data.i[2]** and **data.[3]**) and sometimes as eight individual bytes (**data.ch[0]**, **data.ch[1]... data.ch[7]**).

Also note that, there can exist a union, where each of its elements is of different size. In such a case, the size of the union variable will be equal to the size of the longest element in the union.

Union of Structures

Just as one structure can be nested within another, a union too can be nested in another union. Not only that, there can be a union in a structure, or a structure in a union. Here is an example of stuctures nested in a union.

```
# include <stdio.h>
void main( )
{
    struct a
    {
        int i ;
        char c[2] ;
    } ;
    struct b
    {
        int j ;
        char d[2] ;
    } ;
    union z
    {
        struct a key ;
        struct b data ;
    } ;
    union z strange ;

    strange.key.i = 512 ;
    strange.data.d[0] = 0 ;
```

```
    strange.data.d[1] = 32 ;

    printf ( "\n%d", strange.key.i ) ;
    printf ( "\n%d", strange.data.j ) ;
    printf ( "\n%d", strange.key.c[0] ) ;
    printf ( "\n%d", strange.data.d[0] ) ;
    printf ( "\n%d", strange.key.c[1] ) ;
    printf ( "\n%d", strange.data.d[1] ) ;
}
```

And here is the output...

```
512
512
0
0
32
32
```

Just as we do with nested structures, we access the elements of the union in this program using the '.' operator twice. Thus,

```
strange.key.i
```

refers to the variable **i** in the structure **key** in the union **strange**. Analysis of the output of the above program is left to the reader.

Utility of Unions

Suppose we wish to store information about employees in an organization. The items of information are as shown below:

```
Name
Grade
Age
If Grade = HSK  (Highly Skilled)
    hobbie name
```

```
    credit card no.
If Grade = SSK (Semi Skilled)
    Vehicle no.
    Distance from Co.
```

Since this is dissimilar information we can gather it together using a structure as shown below:

```
struct employee
{
    char n[ 20 ] ;
    char grade[ 4 ] ;
    int age ;
    char hobby[10] ;
    int crcardno ;
    char vehno[10] ;
    int dist ;
} ;
struct employee e ;
```

Though grammatically there is nothing wrong with this structure, it suffers from a disadvantage. For any employee, depending upon his grade either the fields hobby and credit card no. or the fields vehicle number and distance would get used. Both sets of fields would never get used. This would lead to wastage of memory with every structure variable that we create, since every structure variable would have all the four field apart from name, grade and age. This can be avoided by creating a **union** between these sets of fields. This is shown below:

```
sruct info1
{
    char hobby[10] ;
    int crcardno ;
} ;
sruct info2
{
```

```
    char vehno[10] ;
    int dist ;
} ;
union info
{
    struct info1 a ;
    struct info2 b ;
} ;
struct emp
{
    char n[ 20 ] ;
    char grade[ 4 ] ;
    int age ;
    union  info f ;
} ;
struct employee e ;
```

The *volatile* Qualifier

When we define variables in a function the compiler may optimize the code that uses the variable. That is, the compiler may compile the code in a manner that will run in the most efficient way possible. The compiler achieves this by using a CPU register to store the variable's value rather than storing it in stack.

However, if we declare the variable as volatile, then it serves as a warning to the compiler that it should not *optimize* the code containing this variable. In such a case whenever we use the variable its value would be loaded from memory into register, operations would be performed on it and the result would be written back to the memory location allocated for the variable.

We can declare a volatile variable as:

```
volatile int j ;
```

Another place where we may want to declare a variable as volatile is when the variable is not within the control of the program and is likely to get altered from outside the program. For example, the variable

```
volatile float temperature ;
```

might get modified through the digital thermometer attached to the computer.

Summary

(a) The enumerated data type and the **typedef** declaration add to the clarity of the program.
(b) Typecasting makes the data type conversions for specific operations.
(c) When the information to be stored can be represented using a few bits of a byte we can use bit fields to pack more information in a byte.
(d) Every C function has an address that can be stored in a pointer to a function. Pointers to functions provide one more way to call functions.
(e) We can write a function that receives a variable number of arguments.
(f) Unions permit access to same memory locations in multiple ways.

Exercise

[A] What will be the output of the following programs:

(a)
```
# include <stdio.h>
void main( )
{
    enum status { pass, fail, atkt } ;
    enum status  stud1, stud2, stud3 ;
    stud1 = pass ;
```

```
    stud2 = fail ;
    stud3 = atkt ;
    printf ( "\n%d %d %d", stud1, stud2, stud3 ) ;
}
```

(b)
```
# include <stdio.h>
void main( )
{
    printf ( "%f", ( float ) ( ( int ) 3.5 / 2 ) ) ;
}
```

(c)
```
# include <stdio.h>
void main( )
{
    float i, j ;
    i = ( float ) 3 / 2 ;
    j = i * 3 ;
    printf ( "\n%d", ( int ) j ) ;
}
```

[B] Point out the error, if any, in the following programs:

(a)
```
# include <stdio.h>
void main( )
{
    typedef struct patient
    {
        char name[20] ;
        int age ;
        int systolic_bp ;
        int diastolic_bp ;
    } ptt ;
    ptt p1 = { "anil", 23, 110, 220 } ;
    printf ( "\n%s %d", p1.name, p1.age ) ;
    printf ( "\n%d %d", p1.systolic_bp, p1.diastolic_bp );
}
```

(b)
```
# include <stdio.h>
```



```
void show( ) ;
void main( )
{
    void ( *s )( ) ;
    s = show ;
    ( *s )( ) ;
}
void show( )
{
    printf ( "\ndon't show off. It won't pay in the long run" ) ;
}
```

(c)
```
# include <stdio.h>
int show( ) ;
void main( )
{
    int ( *s )( ) ;
    s = show( ) ;
    ( *s )( ) ;
}
float show( )
{
    printf ( "\nControl did reach here" ) ;
    return ( 3.33 ) ;
}
```

(d)
```
# include <stdio.h>
void show ( int, float ) ;
void main( )
{
    void ( *s )( int, float ) ;
    s = show ;
    ( *s )( 10, 3.14 ) ;
}
void show ( int i, float f )
{
    printf ( "\n %d %f", i, f ) ;
}
```