One more Example:

```
struct book
{
char book_name[100];
char author[100];
int pages;
float price;
};

struct book b[100];/* Array of structure *
```

Here, b[100] declares 100 variables of structure book.

Example:

```
#include <stdio.h>
#include<conio.h>
struct student
{
    char name[100];
    char branch[100];
    int roll_no;
};
void main()
{
int i;
struct student std[100];
clrscr();
printf("\nEnter Student Details\n");
for(i=0;i<3;i++)
{
    fflush(stdin);
    printf("Student %d \nName : ",i);
    gets(std[i].name);
    printf("Branch : ");
    gets(std[i].branch);
    printf("Roll No. : ");
    scanf("%d",&std[i].roll_no);
}
printf("\nStudent Information\n");
for(i=0;i<3;i++)
{
    printf("Student %d\n",i);
    printf("Name   : %s\n",std[i].name);
    printf("Branch   : %s\n",std[i].branch);
    printf("Roll No. : %d\n",std[i].roll_no);
}
getch();
}
```

Output:

```
Enter Student Details
Student 0
Name : a
Branch  : CE
Roll No. : 01
Student 1
Name : b
Branch  : ME
Roll No. : 12
Student 2
Name : c
Branch  : EC
Roll No. : 60
Student Information
Student 0
Name : a
Branch  : CE
Roll No. : 01
Student 1
Name : b
Branch  : ME
Roll No. : 12
Student 2
Name : c
Branch  : EC
Roll No. : 60
```

Array Vs Structure

Both the arrays and structures are classified as structured data types as they pro-vide a mechanism that enable us to access and manipulate data in a relatively easy manner. But they differ in a number of ways.

Array	Structure
– An array is a collection of related data elements of same type. – An array is derived data type. – Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it.	– Structure can have elements of different types. – Structure is a programmer-defined. – First we have to design and declare a data structure before the variables of that type are declared and used.

6.5 UNIONS

Same as structure, Union is a user defined data type. It allows grouping of data of different types. But the difference is: Structure occupy more memory because all the member variables occupy separate storage, while Union

occupy less memory because all the member variables are not allocated separate storage but share a memory area between all the memory variables.

Use of union

They are useful for applications involving multiple members.

The real purpose of unions is to prevent memory fragmentation by arranging for a standard size for data in the memory. By having a standard data size dynamically allocated memory will always be reusable by another instance of the same type of union.So, Union allows storage of one member at a time.

6.6 DECLARATION, INITIALIZATION AND ACCESSING OF UNIONS

Like structures union can be declared using the keyword union as follows

Syntax:

```
union union-name
{
datatype membername1;
datatypte membername2;
datatype membername n;
} union-variable1,union-variable2,....union-variable n;
```

Where

- Union is the keyword
- Union-name is the name of union required
- Memebername ,memebername .....membername are members of union

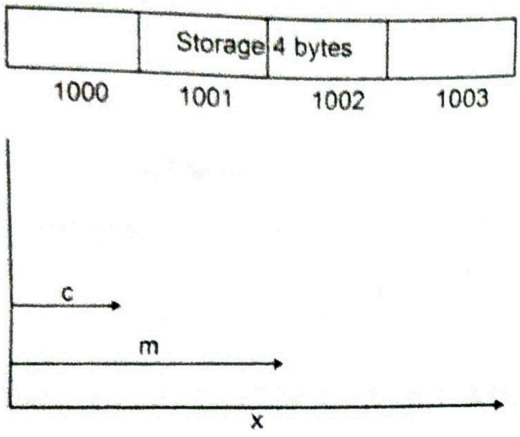
Example

```
union item
{
int m;
float x;
char c;
} code;
```

This declares a variable code of type union item. Memory calculation for union can be calculated as follows:

The size of a union is the size of its largest field. This is because a sufficient number of bytes must be reserved for the largest sized field. The union contains three members each with a different data type.

As shown in above figure a variable code is of union type. The union contains three members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size. This will be clear from the figure given above. The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.



As shown in above table union variable will require the maximum of the biggest size member of union. In above example the member "x" requires the maximum size (4 bytes) so the union variable "code" will require 4 bytes memory even if the total of all members is 7 bytes. This is the basic difference between structure and unions.

Member name	datatype	size in byte
m	int	2
x	float	4
c	char	1
size of union variable "code"		4

That is

```
code.m
code.x
code.c
```

are all valid member variables.

ACCESSING UNION MEMBERS

Syntactically, members of a union are accessed as

```
union-variable.member
```

During accessing we should make sure that we are accessing the member whose value is currently stored.

For example, a statement such as

```
code.m=456;
code.x=456.78;
printf("&%d",code.m);
```

Would produce erroneous result.

In effect a union creates a storage location that can be used by one of its members at a time. When a different member is assigned a value the new member's value supersedes the previous member's value.

Example program demonstrating union

```
#include <stdio.h>
#include<conio.h>

union book
{
int pages;
float price;
};

void main()
{
union book book1;
printf("\nBook Information \n");
book1.pages=406;
printf("Book Pages  : %d\n",book1.pages);
book1.price=450.40;
printf("Book Price  : %.2f\n",book1.price);
getch();
}
```

Output

```
Book Information
Book Pages  : 406
Book Price  : 450.40
```

This declares a union of book type. Variables of unions are then declared as:

```
union book book1,book2,book3;
```

Or array of union variables can be declared as follows:

```
union book book[5];
```

We can initialize or assign the members of union as follows:

```
book1.pages=406;
```

We can read or access the value of members of union as follows:

```
printf("Book Pages  : %d\n",book1.pages);
```

INITIALIZING UNIONS

With a union, fields share the same memory space, so fresh data replaces any existing data. Let's see an example to demonstrates what happens if you initialize all the data for a union variable, all at once:

```
#include <stdio.h>
#include<conio.h>
struct sbook
{
    int pages;
    float price;
};

union ubook
{
    int pages;
    float price;
};

void main()
{
    struct sbook c={300,100.00};
    union ubook cpp;
    /* union ubook cpp(400,150.00); not possible with union*/

    printf("\nC Book Information using Structure \n");
    printf("Book Pages  : %d\n",c.pages);
    printf("Book Price  : %.2f\n",c.price);

    printf("\nC++ Book Information using Union\n");
    cpp.pages=400;
    cpp.price=150.00;
    printf("Book Pages  : %d\n",cpp.pages);
    printf("Book Price  : %f\n",cpp.price);

    printf("\nWrong Result!!! Here is the right way\n");
    cpp.pages=400;
    printf("Book Pages  : %d\n",cpp.pages);
    cpp.price=150.00;
    printf("Book Price  : %f\n",cpp.price);
    getch();
}
```

Output

```
C Book Information using Structure
Book Pages  : 300
Book Price  : 100.00
```

```
C++ Book Information using Union
Book Pages  : 400
Book Price  : 150.000000

C Book Information using Structure
Book Pages  : 300
Book Price  : 100.00

C++ Book Information using Union
Book Pages  : 0
Book Price  : 150.000000

Wrong Result!!! Here is the right way
Book Pages  : 400
Book Price  : 150.000000
```

Inside main, you can't initialize fields of a union variable all at once - try the commented initialization above and see what your compiler says. The first printf statement displays the expected output for the struct version of our C Book. The second printf shows you that the program has overwritten data in the ammo field with data of the energy field - this is how unions work. All fields share the same address, whereas with structs, each field has its own address.

Structure Vs Union

Structure	Union
– Each member in structures is stored in their own separate memory locations.	– All the members in Union share the same common storage area in memory.
– Structure allocates memory equal to the total memory required by all the members.	– Union allocates the memory equal to the maximum memory required by any of the member.
– Any member can be accessed at a time because each and every member is alive and in memory.	– In union only one member can be accessed at a time because only one member is alive and in memory.
– All members of structure variables can be initialized.	– When declaring union variable we can initialize only first member of union.
– Used when memory is not constraints.	– Used to conserve memory.
– Example: Struct test {int a; float f; char c;} Size of a=2 byte Size of f=4 byte Size of c=1 byte Size of struct= total size of (a,b,c) = 2+4+1=7 bytes.	– Example: Union test {int a; float f; char c;} Size of a=2 byte Size of f=4 byte Size of c=1 byte Size of union = max size of (a,b,c) = 4 bytes.

6.7 INTRODUCTIONS TO FILES

In C, a file can refer to a disk file, a terminal, a printer, or a tape drive. In other words, a file represents a concretedevice with which you want to exchange information. Before you perform any communication to a file, you have toopen the file. Then you need to close the opened file after you finish exchanging information with it.

6.7.1 Streams

The data flow you transfer from your program to a file, or vice versa, is called a stream, which is a series of bytes.Not like a file, a stream is device-independent. All streams have the same behavior. To perform I/O operations, youcan read from or write to any type of files by simply associating a stream to the file.

There are two formats of streams. The first one is called the text stream, which consists of a sequence of characters(that is, ASCII data)..The second format of streams is called the binary stream, which is a series of bytes. The content of an .exe file wouldbe one example. Binary streams are primarily used for non-textual data, which is required to keep the exact contents ofthe file.

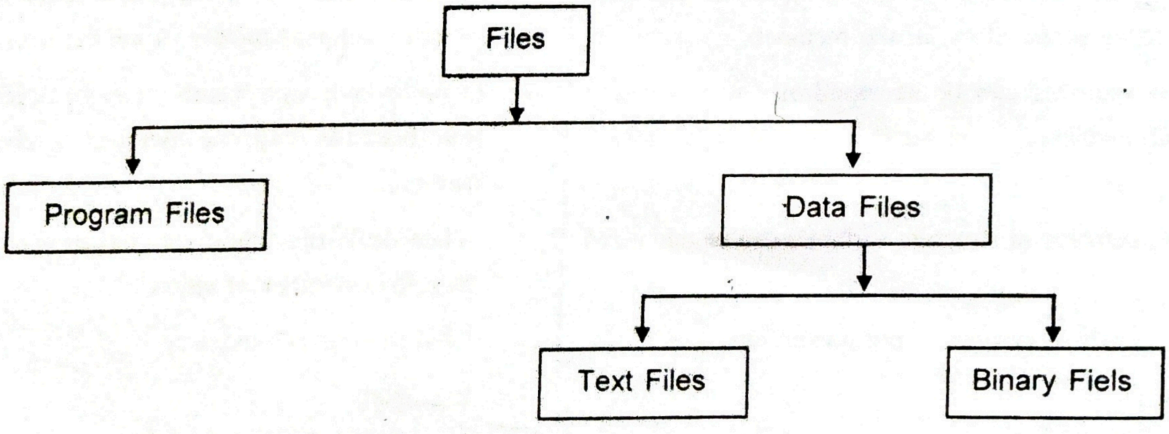
6.7.2 Buffered I/O

In C, a memory area, which is temporarily used to store data before it is sent to its destination, is called a buffer. Withthe help of buffers, the operating system can improve efficiency by reducing the number of accesses to I/O devices(that is, files).

By default, all I/O streams are buffered.

6.7.3 Types of Files

Data files can be subdivided into two categories.



Text Files

Text files are consisting of consecutive characters. These characters can be interpreted as individual data items, or as components of strings or numbers. The manner in which these characters are interpreted is determined either by the particular library functions used to transfer the information, or by format specifications within the library functions, as in the scanf and printf functions.