Unit 5
Structures, Pointers and File Management in C

Structures - Introduction

- Arrays can be used to represent a group of data elements of same type.
- However, we cannot use an array to represent a collection of data elements of different types using a single name.
- C supports a constructed (or programmer-defined) data type called structures, a mechanism for grouping data of different types.
  - A structure can be used to group logically related data items.

Structures help us to organize complex data in a meaningful way…

- book
  - author, title, price, year
- student
  - name, rollno, marks
- address
  - name, door_number, street, city, pin
- customer
  - name, phone_no, city
- inventory
  - item, stock, price

Arrays Vs. Structures

- An array is a collection of related data elements of same type.
  - Structure can have elements of different types.
- An array is a derived data type whereas a structure is a programmer-defined type.
- An array behaves like a built-in type. We just declare an array and use it.
  - In case of structure, we need to define the format of the structure before the variables of that type are declared and used.

Defining a Structure

- Syntax

```
struct structure_name
{
    data_type member1;
    data_type member2;
    …
    data_type member;
};
```

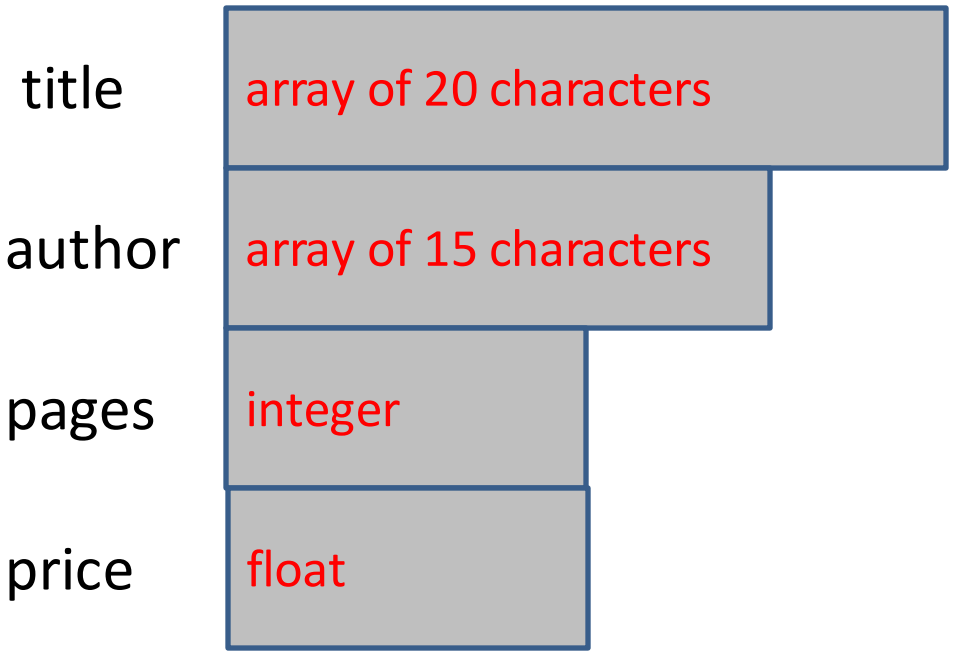
- Example

```
struct Book
{
    char title[20];
    char author[15];
    int pages;
    float price;
};
```

Defining a Structure

- The keyword struct defines a structure to hold four data fields viz., title, author, pages and price.
  - These fields are called structure elements or members.
  - Each member may belong to a different data type.
  - Book is the name of the structure and is called the structure tag.
  - The tag name can be used subsequently to declare variables.

Defining a Structure



- Note that the definition does not declare any variables. It simply describes a format called *template* to represent the information as shown above.

Declaring Structure Variables

- Syntax:
  - struct <tag_name> variable_list;
- Example:
  - struct Book book1, book2, books[10];
- Remember that the members of a structure themselves are not variables.
- They do not occupy any memory until they are associated with the structure variables such as book1.

- The following declaration is also valid.

```
struct
{
    char title[20];
    char author[15];
    int pages;
    float price;
} book1, book2, books[10];
```

- The use of tag_name is optional here.
  - However, this is not recommended, as we cannot use it for future declarations.

Accessing Structure Members

- Structure members have no meaning individually without the structure.
- In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot . operator also called **period** or **member access** operator.
  - strcpy(book1.title, “Programming in C”);
  - strcpy(book1.author, “Balaguruswamy”);
  - book1.pages = 250
  - book1.price = 175.50;

We can also use scanf to assign values to the members.

```
struct Student
{
   char name[25];
   int rollno;
   char branch[10];
};

main()
{
   struct Student s1;
   printf("Enter name, roll number and branch:");
   scanf("%s%d%s", s1.name, &s1.rollno, s1.branch);
   printf("Student details are...\n");
   printf("Name:%s\nAge:%d\nBranch:%s",s1.name,  s1.rollno, s1.branch);
}
```

Structure Initialization

- C does not permit the initialization of individual structure members within the template.
- The initialization must be done only in the declaration of the actual variables.
- Like any other data type, a structure variable can be initialized at compile time.
- Syntax:
  - struct <tag_name> var_name = {initialization_list};
- Example:
  - struct Student s1 = {"Ramesh", 25, "Civil"};

Rules for Initializing Structures

- We cannot initialize individual members inside the structure template.
- The order of values enclosed in initialization_list must match the order of members in the structure definition.
- It is possible to have partial initialization.
  - The uninitialized members should be at the end of the list.
- The uninitialized members will be assigned with default values.
  - Zero for integer and floating point numbers,
  - '\0' for characters and strings.

Copying and Comparing Structure Variables

- Two variables of the same structure type can be copied the same way as ordinary variables.
  - book2 = book1; //Permitted
- However, C does not permit the use of logical operators (== and !=) on structure variables.
  - In such case, we need to compare them by comparing members individually.

Comparing Two Structure Variables

```
struct Student s1 = {"Ramesh", 25, "Civil"};
struct Student s2 = s1; //We can assign one to another
struct Student s3 = {"RAMESH", 25, "Civil"};
struct Student s4 = {"Ramesh", 25, "Civil"};
int res;
res = ((strcmp(s1.name,s2.name)==0) && (s1.rollno == s2.rollno)
        && (strcmp(s1.branch,s2.branch)==0)) ? 1 : 0;
printf("s1 and s2 are same:%d\n", res); //Prints 1
res = ((strcmp(s1.name,s3.name)==0) && (s1.rollno == s3.rollno)
        && (strcmp(s1.branch,s3.branch)==0)) ? 1 : 0;
printf("s1 and s3 are same:%d\n", res); //Prints 0
res = ((strcmp(s1.name,s4.name)==0) && (s1.rollno == s4.rollno)
        && (strcmp(s1.branch,s4.branch)==0)) ? 1 : 0;
printf("s1 and s4 are same:%d\n", res); //Prints 1
```

Operations on Individual Members

- Individual members of a structure are accessed using the dot operator.
- A member of a structure can be treated like any other variable name and therefore can be manipulated using expressions and operators.
  - if(s1.rollno == 111)
    - s1.marks = s1.marks + 5;
  - float sum = s1.marks + s2.marks;
  - s2.marks = s2.marks + 4;
  - s1.marks++;

Union in C

- A union is a special data type available in C that allows to store different data types in the same memory location.
- Union can be defined with many members, but only one member can contain a value at any given time.
- Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

- The format of the union statement is as follows-

```
union [union tag] {
member definition;
member definition;
...
member definition;
} [one or more union variables];
```

- The **union tag** is optional

Example

```
union Data {
    int i;
    float f;
    char str[20];
} var;
```

Example

```
#include <stdio.h>
#include <string.h>
union Data {
        int i;
        float f;
        char str[20];
};
int main( ) {
    union Data data;
    printf( "Memory size occupied by data : %d\n",
sizeof(data));
    return 0;
}
```

Output

- When the above code is compiled and executed, it produces the following result –

```
Memory size occupied by data : 20
```

Accessing Union Members

- To access any member of a union, we use the member access operator (.)

```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

Output:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

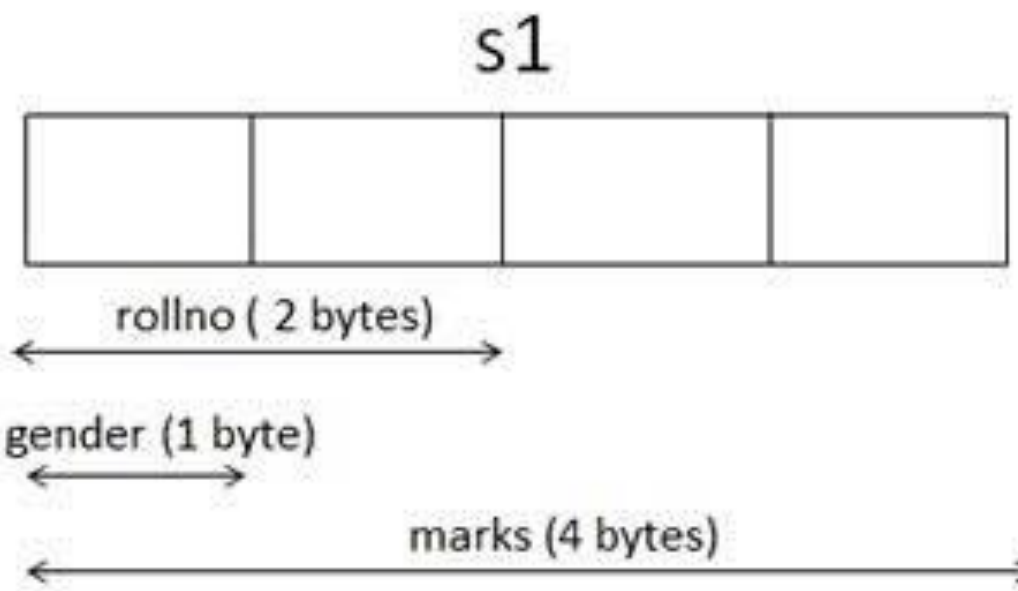
- In the previous output we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

```c
#include <stdio.h>
#include <string.h>

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

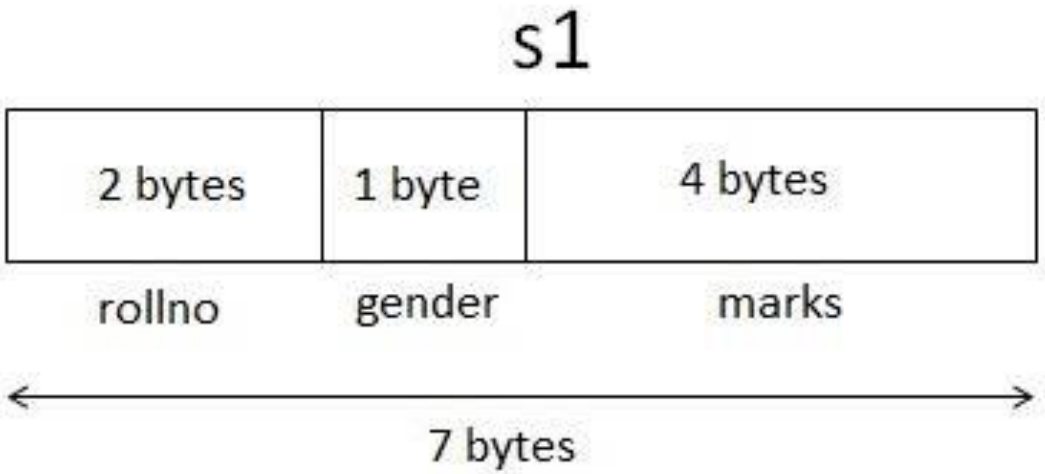
When the above code is compiled and executed, it produces the following result –

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.



Memory Allocation in Union



Memory Allocation in Structure

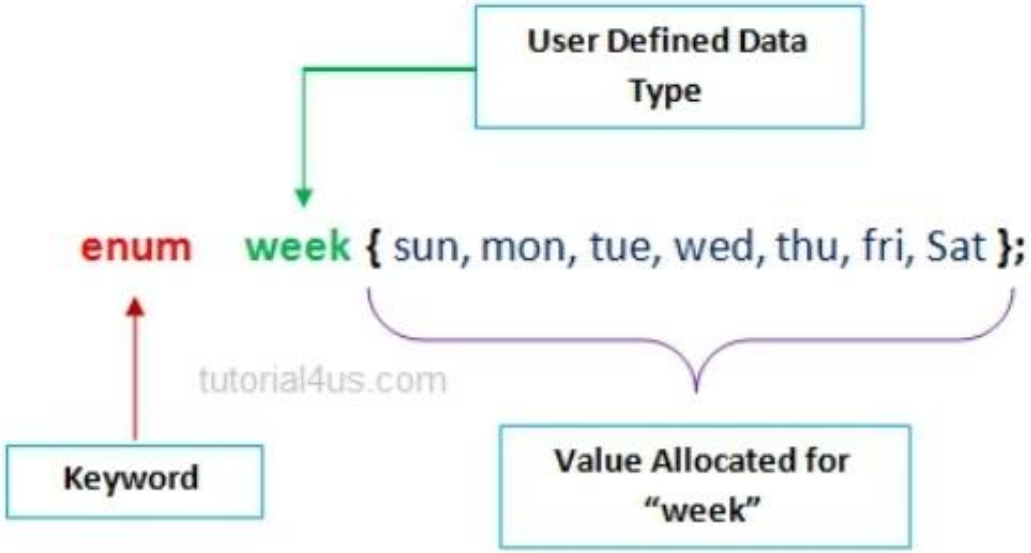
Enumerated Data Type in C

- Enumeration (or enum) is a user defined data type in C.
- It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.
- It is used for creating an user defined data type of integer. Using enum we can create sequence of integer constant value.

Syntax

```
enum tagname {value1, value2, value3,....};
```

- In the above syntax enum is a keyword. It is a user defined data type.
- tagname is our own variable. Tagname is any variable name.
- value1, value2, value3,.... create set of enum values.



Example:

enum cars{BMW, Ferrari, Jeep, Benz};

Here, the default values for the constants are:

BMW=0, Ferrari=1, Jeep=2, and Benz=3.

- However, to change the default values, you can define the enum as follows:

enum cars{ BMW=3, Ferrari=5, Jeep=0, Benz=1 };

Example 1: Printing the Values of Weekdays

```
#include <stdio.h>

    enum days{Sunday=1, Monday, Tuesday,
Wednesday, Thursday, Friday, Saturday};

int main(){

    // printing the values of weekdays

    for(int i=Sunday;i<=Saturday;i++){

        printf("%d, ",i);

    }

    return 0;

}
```

Output:

```
1, 2, 3, 4, 5, 6, 7,
```

Multiple enum names or elements can have the same value. Here's an example of two enum elements having a similar value.

Example:

```
#include <stdio.h>
enum Cars{Jeep = 1, BMW = 0, Mercedes_Benz = 0};

int main(){
    printf("%d, %d, %d", Jeep, BMW, Mercedes_Benz);
    return 0;
}
```

Output:

```
1, 0, 0
```

We can provide values to any elements of enum in any order. All the unassigned elements will get the value as previous + 1. The following program demonstrates the same.

```
#include <stdio.h>
enum weekdays {Sunday, Monday = 2, Tuesday, Wednesday = 6, Thursday,
Friday = 9, Saturday = 12};

int main()
{
printf("%d %d %d %d %d %d %d", Sunday, Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday);
return 0;
}
```

Output:

```
0 2 3 6 7 9 12
```

If we do not assign custom values to enum elements, the compiler will assign them default values starting from 0. For instance, the compiler will assign values to the months in the example below, with January being 0

```
#include <stdio.h>

enum Months{January, February, March, April, May, June, July, August, September, October, November, December};

int main(){

    enum Months m = May;

    printf("The Value of May in Months is %d", m);

return 0;

}
```

Output:

```
The Value of May in Months is 4
```

All enum constants must be unique in their scope. For example, the following program fails in compilation.

```
enum state  {working, failed};
enum result {failed, passed};

int main()  { return 0; }
```

Output:

```
Compile Error: 'failed' has
a previous declaration as
'state failed'
```

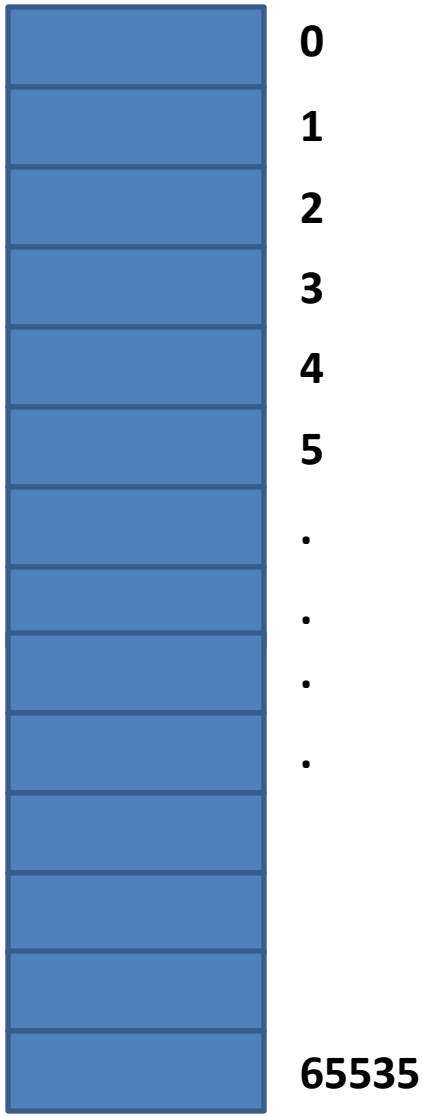
Pointers

Introduction

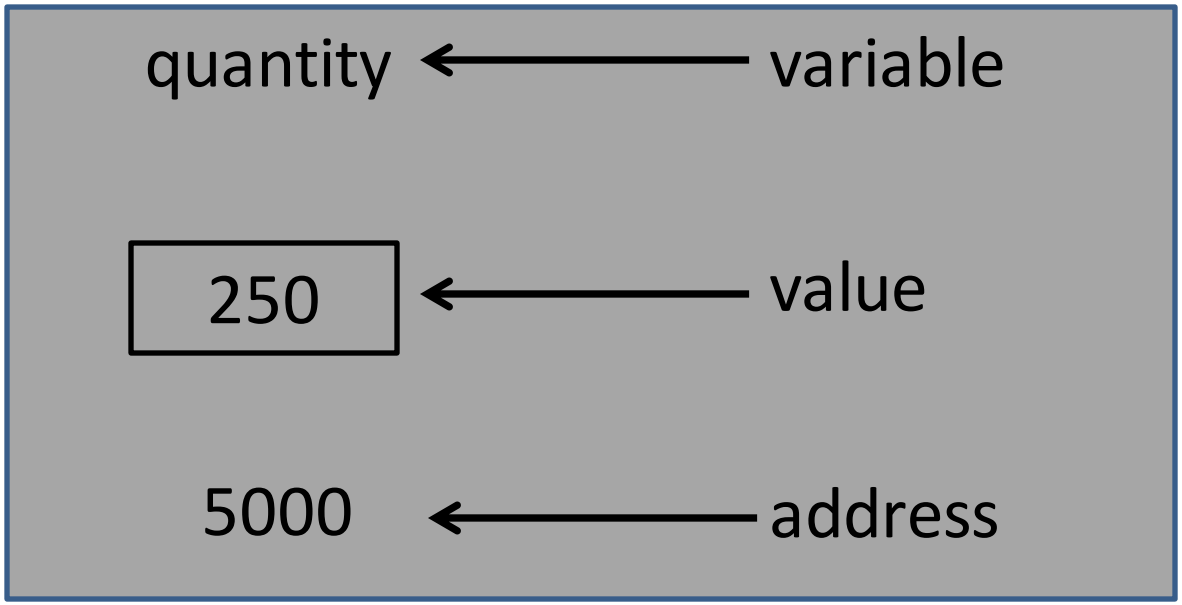
- A pointer is a derived type in C that contains memory address as its value.
- Pointers can be used to return multiple values from a function via function arguments (output parameters).
- Pointers allow C to support dynamic memory management.
- Use of pointers increases the execution speed and thus reduces the program execution time.

Understanding Pointers

- Computer's memory is a sequential collection of storage location.
- Each location, commonly known as a byte, is identified by an address associated with it.
- These addresses are numbered consecutively, starting from zero.
- The last address depends on the memory size.
  - A computer system having 64K memory has its last address as 65535.

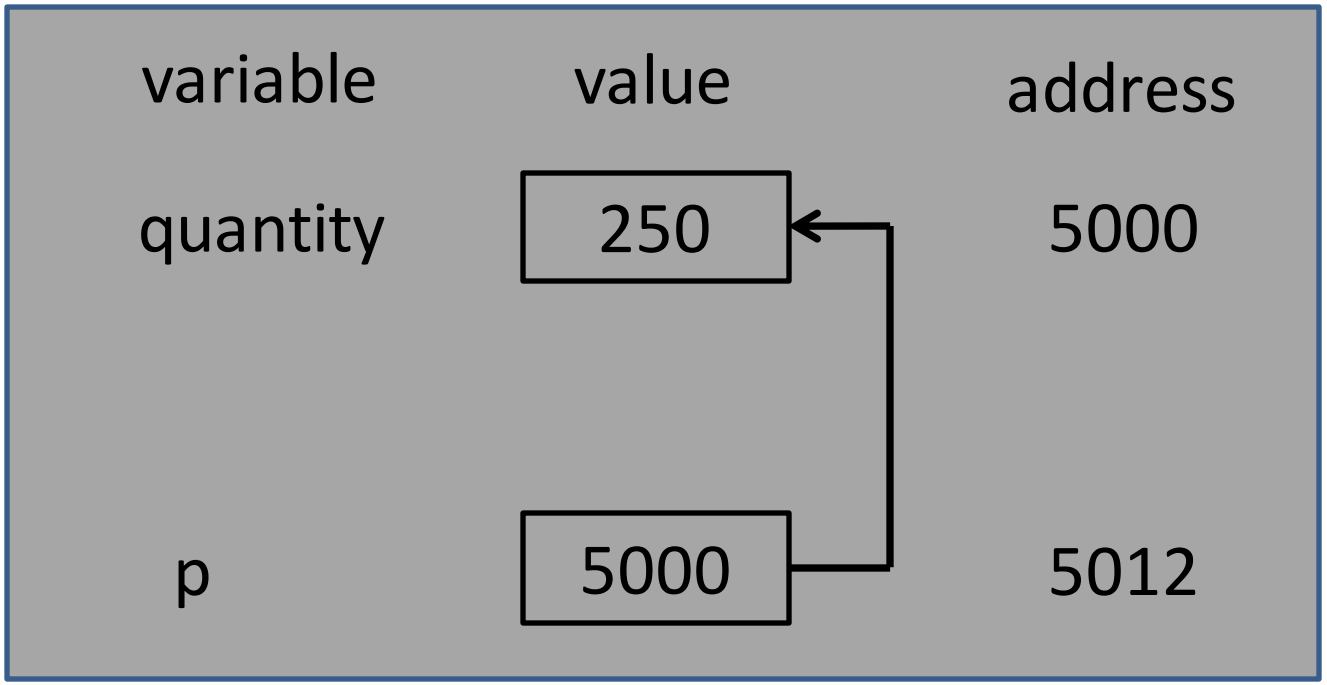


- int quantity = 250;



- A variable that can hold address of another variable is called a pointer variable.

- int quantity = 250;
- int *p = &quantity;



- Since the value of the variable p is the address of the variable quantity, we may access the value of quantity by using the variable p, and therefore we say that the variable p points to the variable quantity. Thus p is the name of the pointer.

Accessing the Address of a Variable

- We can access the address of a variable with the & operator in C.
  - We have already used & operator in scanf function.
- The operator & immediately preceding the variable returns the address of the variable associated with it.
  - p = &quantity; //assigns the address 5000 to p.
- The & operator can be used only with a simple variable or an array element.

```
main()
{
        int a;
        float p, q;
        char choice;
        a = 15;
        p = 10.25f;
        q = 5.46f;
        choice = 'Y';
        printf("%d is stored at address %u\n", a, &a);
        printf("%f is stored at address %u\n", p, &p);
        printf("%f is stored at address %u\n", q, &q);
        printf("%c is stored at address %u\n", choice, &choice);
}
```

Output:

15 is stored at address 6356748

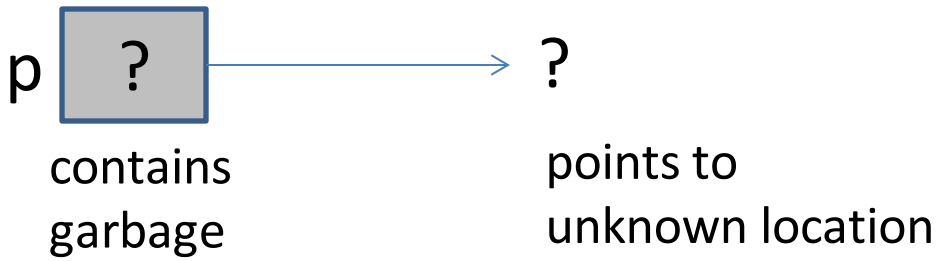
10.250000 is stored at address 6356744

5.460000 is stored at address 6356740

Y is stored at address 6356739

Declaring Pointer Variables

- datatype *ptr_name;
  - The asterix (*) tells that the variable ptr_name is a pointer variable.
  - ptr_name needs a memory location.
  - ptr_name points to a variable of type *data_type.*
- int *p; // declares the variable p as a pointer variable that can point to an // integer data type.
- float *q; // declares the variable q as a pointer variable that can point to // an float data type.
- Since the memory locations corresponding to p and q have not been assigned any values, these contain garbage value aand therefore point to unknown locations.
  - Example: int *p;



Initialization of Pointer Variable

- The process of assigning the address of a variable to a pointer variable is known as initialization.
  - Uninitialized pointers will have unknown values.
  - It is important to initialize pointer variables before they are used in the program.
    - int quantity;
    - int *p;
    - p = &quantity;

} int *p = &quantity;

Initialization of Pointer Variable

- We must ensure that the pointer variables always point to the corresponding type of data.
  - float a, b;
  - int sum, *p;
  - p = &a; // Wrong
- It is possible to declare combine declaration of simple variable, declaration and initialization pointer variable in one statement.
  - int a, *p = &a;
  - Declares a as an integer variable and p as a pointer variable and then initializes p to address of a.
- We can also define a pointer variable with an initial value of NULL or zero.
  - int *p = NULL; // same as int *p = 0;

Accessing a Variable Through its Pointer

- To access value of a variable using the pointer, we use the asterisk (*) operator, also known as the indirection operator or dereferencing operator.
  - int quantity, *p, n;
  - quantity = 250;
  - p = &quantity;
  - n = *p; // equivalent to n = *&quantity; which in turn
    // is equivalent to n = quantity;
  - Here *p returns the value of the variable quantity.

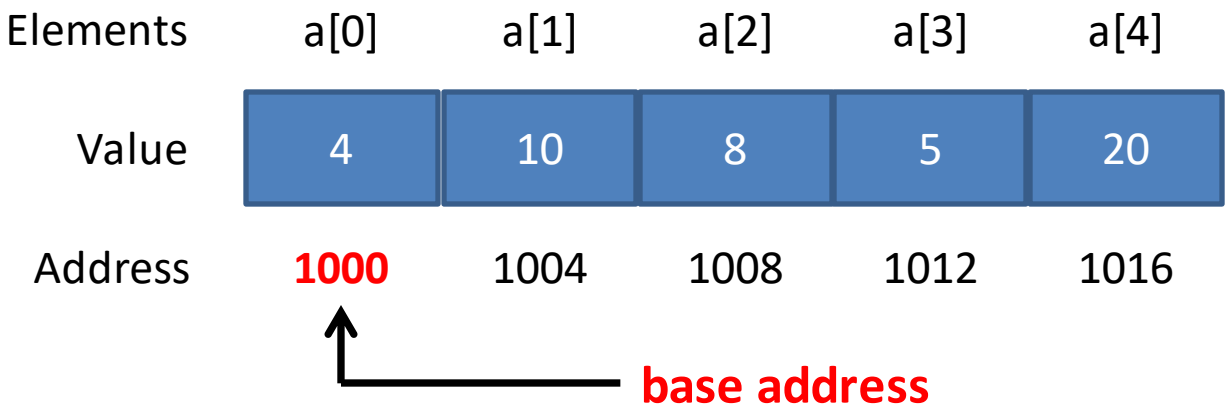
```
main()
{
   int a, b;
   int *p;
   a = 15;
   p = &a;
   b = *p;
   printf("Value of a is %d and address is %u\n", a, &a);
   printf("Value of a is %d and address is %u\n", *&a, &a);
   printf("Value of a is %d and address is %u\n", *p, p);
   printf("%d is the address pointed to by %u\n", p, &p);
   printf("Value of b is %d and address is %u\n", b, &b);
   *p = 25;
   printf("Now the value of a is %d", a);
}
```

```
Value of a is 15 and address is 6356748
Value of a is 15 and address is 6356748
Value of a is 15 and address is 6356748
6356748 is the address pointed to by 6356740
Value of b is 15 and address is 6356744
Now the value of a is 25
```

Stage	Values stored in locations and their addresses		
	a	b	p
Declaration	6356748	6356744	6356740
	a	b	p
a = 10	10		
	6356748	6356744	6356740
	a	b	p
p = &a	10		6356748
	6356748	6356744	6356740
	a	b	p
b = *p	10	10	6356748
	6356748	6356744	6356740
	a	b	p
b = *p	10	10	6356748
	6356748	6356744	6356740

Pointers and Arrays

- When an array is declared, the compiler allocates a base address and memory to hold the array elements in contiguous memory locations.
  - The base address is the location of the first element (index 0) of the array.
  - The compiler also defines the array name as a pointer to the first element.
  - int a[5] = {4, 10, 8, 5, 20};
  - The name a is the base address (pointer) that points to the first element, a[0] and therefore value of a is 1000.



- int *p;
- p = a; // equivalent to p = &a[0];
  - p is an integer pointer that points to array a.
  - We can access elements of a using p.
  - p = &a[0] (=1000)
  - p+1 = &a[1] (=1004)
  - p+2 = &a[2] (=1008)
  - p+3 = &a[3] (=1012)
  - p+4 = &a[4] (=1016)
- Address of a[3] = base address + (3 * sizeof(int))
  = 1000 + (3 * 4) = 1012

Program to add array elements using Pointers

```
main()
{
   int a[] = {3, 4, 8, 2, 7, 5};
   int *p, i, sum;
   p = a;
   sum = 0;
   for(i=0; i<6; i++)
   {
      sum = sum + *p;
      p++;
   }
   printf("Sum is %d", sum);
}
```

Swapping two numbers by Call by Value Method

```
#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
// printing the value of a and b in main
   swap(a,b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);
// The value of actual parameters do not change by changing the formal parameters in call
//by value, a = 10, b = 20
}
void swap (int a, int b)
{
   int temp;
   temp = a;
   a=b;
   b=temp;
   printf("After swapping values in function a = %d, b = %d\n",a,b);
// Formal parameters, a = 20, b = 10
}
```

Output

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

Swapping two numbers by Call by Reference Method

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
// printing the value of a and b in main
   swap(&a,&b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);
// The values of actual parameters do change in call by reference, a = 10, b = 20
}
void swap (int *a, int *b)
{
   int temp;
   temp = *a;
   *a=*b;
   *b=temp;
   printf("After swapping values in function a = %d, b = %d\n",*a,*b);
// Formal parameters, a = 20, b = 10
}
```

Output

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 20, b = 10

Difference between Call by Value and Call by Reference

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

File Management in C

- So far, we have used scanf and printf to read and write data from/to console.
- A file is a place on the disk where a group of related data is stored.
- C supports a number of functions to perform basic file operations such as,
  - Naming a file (while creating a new file),
  - Opening a file,
  - Reading data from a file,
  - Writing data to a file, and
  - Closing a file.

Defining and opening a file

- If we want to create a file and store data into it, we must specify
  - Filename
    - A string of characters that make a valid filename for the operating system.
  - Purpose of opening (whether to read or write)
- FILE is a data type.
- The general form for declaring and opening a file is:
  - FILE *fp;
  - fp = fopen(“Filename”, “mode”);
  - fp is a pointer to data type FILE.
  - mode can be
    - r – open the file for reading only
    - w – open the file for writing only
    - a – open the file for appending (or adding) data to existing file.

- When trying to open a file, one of the following things may happen:
  - When the mode is “w”, a file with the specified name is created if it does not exist. The contents are deleted if the file already exists.
  - When the mode is “a”, the file is opened with the current contents safe. A file with the specified name is created if it does not exist.
  - When the mode is “r”, and if it exists, the file is opened with the current contents safe. Otherwise an error occurs.

```
FILE *fp;
fp = fopen(“input.txt”, “r”);
if(fp == NULL)
    printf(“Such a file does not exist!”);
else
{
    while(!feof(fp))
    {
```

Closing a file

- A file must be closed as soon as all operations on it have been completed.
  - This prevents accidental misuse of the file.
- The function to close a file is:
  - fclose(fp);
  - This will close the file associated with the FILE pointer fp.
    - fclose(fp);

Input/Output Operations on Files

- The getc and putc functions
  - Analogous to getchar and putchar functions and handle one character at a time.
  - putc(ch, fp)
    - writes the character contained in variable ch to the file associated with fp.
  - ch = getc(fp)
    - reads a character from the file whose file pointer is fp.
  - The file pointer moves by one character for every operation of getc or putc.
    - The getc will return an end-of-file marker EOF, when the end of the file has been reached.

```
main()
{
   FILE *fp;
   char ch;
   fp = fopen("input.txt", "w");
   while((ch=getchar())!=EOF)
      putc(ch, fp);
   fclose(fp);
   fp = fopen("input.txt", "r");
   while(!feof(fp))
   {
      printf("%c", getc(fp));
   }
   fclose(fp);
}
```

The fprintf and fscanf Functions

- fprintf and fscanf can handle a group of mixed data simultaneously.
  - fscanf(fp, “control string”, list);
    - Reads the items in the list from the file specified by fp, according to the specifications contained in the control string.
    - When the end of file is reached, it returns the value EOF.
  - fprintf(fp, “control string”, list);
    - fp is the file pointer associated with a file that has been opened for writing.
    - The control string contains output specifications for the items in the list.

```
main()
{
   FILE *fp;
   int rollno, marks, i, n;
   char name[20];
   fp = fopen("student.txt", "w");
   printf("Enter number of students:");
   scanf("%d", &n);
   printf("Enter student data:\n");
   printf("Roll Number    Name    Marks\n");
   for(i=1; i<=n; i++)
   {
      scanf("%d %s %d", &rollno, name, &marks);
      fprintf(fp,"%d %s %d\n", rollno, name, marks);
   }
   fclose(fp);
   fp = fopen("student.txt", "r");
   printf("Roll No   Name    Marks\n");
   for(i=1; i<=n; i++)
   {
      fscanf(fp, "%d %s %d", &rollno, name, &marks);
      printf("%d\t%s\t%d\n", rollno, name, marks);
   }
   fclose(fp);
}
```

- Write a C program that creates a file reading contents that the user types from the keyboard till EOF. The text in this file must be in lowercase. There could be multiple blanks in between some words. Create another file in which the same content is copied in UPPERCASE and with only one blank in between the words that contained multiple blanks.

```
int main()
{
    FILE *f1, *f2;
    char ch, ch1, line[80];
    f1=fopen("input.txt","w");
     printf(“Enter text (Type END to terminate):\n”);
    do
    {
            gets(line);
            if(strcmp(line, "END") == 0)
                        break;
            fputs(line,f1);
            fputs("\n", f1);
    }while(1);
    fclose(f1);
```

Continued…

```
    f1=fopen("input.txt","r");
    f2=fopen("output.txt","w");
    while((ch=getc(f1))!=EOF)
    {
        if(ch==' ' && ch==ch1)
                continue;
        putc(toupper(ch),f2);
        ch1=ch;
    }
    fclose(f1);
    fclose(f2);
    printf("\n Copied to output file. \n\n");
    return 0;
}
```