UNIT IV

COMMAND LINE ARGUMENTS

It is possible to pass some values from the command line to C programs when they are executed. These values are called **command line arguments** and many times they are important for our program especially when we want to control our program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program.

Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly –

```
#include <stdio.h>

int main( int argc, char *argv[] )  {

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and**argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

We pass all the command line arguments separated by a space, but if argument itself has a space then we can pass such arguments by putting them inside double quotes "" or single quotes ".

RECURSION

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
  recursion(); /* function calls itself */
}

int main() {
  recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Example:

```
#include <stdio.h>

int fibonaci(int i) {
```

```
  if(i == 0) {
    return 0;
  }

  if(i == 1) {
    return 1;
  }
  return fibonaci(i-1) + fibonaci(i-2);
}

int  main() {

  int i;

  for (i = 0; i < 10; i++) {
    printf("%d\t%n", fibonaci(i));
  }

  return 0;
}
```

DECLARING VARIABLE LENGTH PARAMETERLISTS

Whenever a function is declared to have an indeterminate number of arguments, in place of the last argument you should place an ellipsis ('...'), so, int a_function (int x, ...); would tell the compiler the function should accept however many arguments that the programmer uses, as long as it is equal to at least one, the one being the first, x.

We'll need to use some macros from the stdarg.h header file to extract the values stored in the variable argument list--va_start, which initializes the list, va_arg, which returns the next argument in the list, and va_end, which cleans up the variable argument list.

To use these functions, we need a variable capable of storing a variable-length argument list--this variable will be of type va_list. va_list is like any other type. For example, the following code declares a list that can be used to store a variable number of arguments.

```
va_list a_list;
```

va_start is a macro which accepts two arguments, a va_list and the name of the variable that directly precedes the ellipsis ("..."). So in the function a_function, to initialize a_list with va_start, we would write va_start (a_list, x);

```
int a_function ( int x, ... )
{
   va_list a_list;
   va_start( a_list, x );
}
```

va_arg takes a va_list and a variable type, and returns the next argument in the list in the form of whatever variable type it is told. It then moves down the list to the next argument. For example, va_arg (a_list, double) will return the next argument, assuming it exists, in the form of a double. The next time it is called, it will return the argument following the last returned number, if one exists. Note that we need to know the type of each argument--that's part of why printf requires a format string! Once we're done, use va_end to clean up the list: va_end(a_list);

To show how each of the parts works, take an example function:

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

/* this function will take the number of values to average
  followed by all of the numbers to average */
double average ( int num, ... )
{
   va_list arguments;
   double sum = 0;

   /* Initializing arguments to store all values after num */
   va_start ( arguments, num );
   /* Sum all the inputs; we still rely on the function caller to tell us how
    * many there are */
   for ( int x = 0; x < num; x++ )
   {
      sum += va_arg ( arguments, double );
   }
   va_end ( arguments );                  // Cleans up the list

   return sum / num;
```

```
}

int main()
{
   /* this computes the average of 13.2, 22.3 and 4.5 (3 indicates the number of values to average) */
   printf( "%f\n", average ( 3, 12.2, 22.3, 4.5 ) );
   /* here it computes the average of the 5 values 3.3, 2.2, 1.1, 5.5 and 3.3
   printf( "%f\n", average ( 5, 3.3, 2.2, 1.1, 5.5, 3.3 ) );
}
```

STRUCTURES

Structure is the collection of variables of different types under a single name for better handling. For example: You want to store the information about person about his/her name, citizenship number and salary. You can create these information separately but, better approach will be collection of these information under single name because all these information are related to person.

Keyword struct is used for creating a structure.

Syntax of structure

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
```

```
    data_type memeber;
};
```

We can create the structure for a person as mentioned above as:

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
};
```

This declaration above creates the derived data type **struct person**.

Structure variable declaration

When a structure is defined, it creates a user-defined type but, no storage is allocated. For the above structure of person, variable can be declared as:

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
};
```

Inside main function:

struct person p1, p2, p[20];

Another way of creating sturcture variable is:

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
}p1 ,p2 ,p[20];
```

In both cases, 2 variables *p1*, *p2* and array *p* having 20 elements of type **struct person** are created.

Accessing members of a structure

There are two types of operators used for accessing members of a structure.

1. Member operator(.)
2. Structure pointer operator(->) (will be discussed in structure and pointers chapter)

Any member of a structure can be accessed as: structure_variable_name.member_name

Suppose, we want to access salary for variable *p2*. Then, it can be accessed as:

```
p2.salary
```

Example of structure

Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet.(Note: 12 inches = 1 foot)

```
#include <stdio.h>
struct Distance{
    int feet;
    float inch;
}d1,d2,sum;
int main(){
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d",&d1.feet);  /* input of feet for structure variable d1 */
```

```
    printf("Enter inch: ");
    scanf("%f",&d1.inch);  /* input of inch for structure variable d1 */
    printf("2nd distance\n");
    printf("Enter feet: ");
    scanf("%d",&d2.feet);  /* input of feet for structure variable d2 */
    printf("Enter inch: ");
    scanf("%f",&d2.inch);  /* input of inch for structure variable d2 */
    sum.feet=d1.feet+d2.feet;
    sum.inch=d1.inch+d2.inch;
    if (sum.inch>12){  //If inch is greater than 12, changing it to feet.
       ++sum.feet;
       sum.inch=sum.inch-12;
    }
    printf("Sum of distances=%d\'-%.1f\"",sum.feet,sum.inch);
/* printing sum of distance d1 and d2 */
    return 0;
}
```

Output

```
1st distance

Enter feet: 12

Enter inch: 7.9

2nd distance

Enter feet: 2

Enter inch: 9.8

Sum of distances= 15'-5.7"
```

ARRAYS OF STRUCTURES

Structure is used to store the information of One particular object but if we need to store such 100 objects then Array of Structure is used.

Example :

```
struct Bookinfo
{
    char[20] bname;
    int pages;
    int price;
}Book[100];
```

Explanation :

1. Here Book structure is used to Store the information of one Book.
2. In case if we need to store the Information of 100 books then Array of Structure is used.
3. b1[0] stores the Information of 1st Book , b1[1] stores the information of 2nd Book and So on We can store the information of 100 books.

book[3] is shown Below

Accessing Pages field of Second Book :

```
Book[1].pages
```

Live Example :

```
#include <stdio.h>

struct Bookinfo
{
    char[20] bname;
```

```
    int pages;
    int price;
}book[3];

int main(int argc, char *argv[])
{
int i;

for(i=0;i<3;i++)
   {
   printf("\nEnter the Name of Book    : ");
   gets(book[i].bname);
   printf("\nEnter the Number of Pages : ");
   scanf("%d",book[i].pages);
   printf("\nEnter the Price of Book   : ");
   scanf("%f",book[i].price);
   }

printf("\n--------- Book Details ---------- ");

for(i=0;i<3;i++)
   {
   printf("\nName of Book    : %s",book[i].bname);
   printf("\nNumber of Pages : %d",book[i].pages);
   printf("\nPrice of Book   : %f",book[i].price);
   }
```

```
return 0;
}
```

Output of the Structure Example:

```
Enter the Name of Book    : ABC
Enter the Number of Pages : 100
Enter the Price of Book   : 200
Enter the Name of Book    : EFG
Enter the Number of Pages : 200
Enter the Price of Book   : 300
Enter the Name of Book    : HIJ
Enter the Number of Pages : 300
Enter the Price of Book   : 500

--------- Book Details ------------

Name of Book    : ABC
Number of Pages : 100
Price of Book   : 200
Name of Book    : EFG
Number of Pages : 200
Price of Book   : 300
Name of Book    : HIJ
Number of Pages : 300
Price of Book   : 500
```

PASSING STRUCTURES TO FUNCTIONS

In C, structure can be passed to functions by two methods:

1. Passing by value (passing actual value as argument)
2. Passing by reference (passing address of an argument)

Passing structure by value

A structure variable can be passed to the function as an argument as normal variable. If structure is passed by value, change made in structure variable in function definition does not reflect in original structure variable in calling function.

Write a C program to create a structure student, containing name and roll. Ask user the name and roll of a student in main function. Pass this structure to a function and display the information in that function.

```
#include <stdio.h>
struct student{
    char name[50];
    int roll;
};
void Display(struct student stu);
/* function prototype should be below to the structure declaration otherwise compiler
shows error */
int main(){
    struct student s1;
    printf("Enter student's name: ");
    scanf("%s",&s1.name);
    printf("Enter roll number:");
    scanf("%d",&s1.roll);
    Display(s1);   // passing structure variable s1 as argument
    return 0;
}
void Display(struct student stu){
  printf("Output\nName: %s",stu.name);
  printf("\nRoll: %d",stu.roll);
}
```

Output

Enter student's name: Kevin Amla

Enter roll number: 149

Output

Name: Kevin Amla

Roll: 149

Passing structure by reference

The address location of structure variable is passed to function while passing it by reference. If structure is passed by reference, change made in structure variable in function definition reflects in original structure variable in the calling function.

Write a C program to add two distances(feet-inch system) entered by user. To solve this program, make a structure. Pass two structure variable (containing distance in feet and inch) to add function by reference and display the result in main function without returning it.

```
#include <stdio.h>
struct distance{
   int feet;
   float inch;
};
void Add(struct distance d1,struct distance d2, struct distance *d3);
int main()
{
   struct distance dist1, dist2, dist3;
   printf("First distance\n");
   printf("Enter feet: ");
   scanf("%d",&dist1.feet);
   printf("Enter inch: ");
   scanf("%f",&dist1.inch);
   printf("Second distance\n");
   printf("Enter feet: ");
   scanf("%d",&dist2.feet);
   printf("Enter inch: ");
   scanf("%f",&dist2.inch);
   Add(dist1, dist2, &dist3);
```

```
/*passing structure variables dist1 and dist2 by value whereas passing structure
variable dist3 by reference */
   printf("\nSum of distances = %d\'-%.1f\"",dist3.feet, dist3.inch);
   return 0;
}
void Add(struct distance d1,struct distance d2, struct distance *d3)
{
/* Adding distances d1 and d2 and storing it in d3 */
    d3->feet=d1.feet+d2.feet;
    d3->inch=d1.inch+d2.inch;
    if (d3->inch>=12) {     /* if inch is greater or equal to 12, converting it to feet. */
       d3->inch-=12;
       ++d3->feet;
   }
}
```

Output

```
First distance

Enter feet: 12

Enter inch: 6.8

Second distance

Enter feet: 5

Enter inch: 7.5



Sum of distances = 18'-2.3"
```

Explaination

In this program, structure variables *dist1* and *dist2* are passed by value (because value of *dist1*and *dist2* does not need to be displayed in main function) and *dist3* is passed by reference ,i.e, address of *dist3* (&dist3) is passed as an argument. Thus, the structure pointer variable *d3* points to the address of *dist3*. If any change is made in *d3* variable, effect of it is seed in *dist3* variable in main function.

STRUCTURE POINTERS

Pointers can be accessed along with structures. A pointer variable of structure can be created as below:

```
struct name {
    member1;
    member2;
    .
    .
};
-------- Inside function -------
struct name *ptr;
```

Here, the pointer variable of type **struct name** is created.

Structure's member through pointer can be used in two ways:

1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation

Consider an example to access structure's member through pointer.

```
#include <stdio.h>
struct name{
   int a;
   float b;
};
int main(){
    struct name *ptr,p;
    ptr=&p;          /* Referencing pointer to memory address of p */
    printf("Enter integer: ");
    scanf("%d",&(*ptr).a);
    printf("Enter number: ");
    scanf("%f",&(*ptr).b);
    printf("Displaying: ");
```

```
    printf("%d%f",(*ptr).a,(*ptr).b);
    return 0;
}
```

In this example, the pointer variable of type **struct name** is referenced to the address of *p*. Then, only the structure member through pointer can can accessed.

Structure pointer member can also be accessed using -> operator.

(*ptr).a is same as ptr->a

(*ptr).b is same as ptr->b

Accessing structure member through pointer using dynamic memory allocation

To access structure member using pointers, memory can be allocated dynamically using malloc() function defined under "stdlib.h" header file.

Syntax to use malloc()

ptr=(cast-type*)malloc(byte-size)

Example to use structure's member through pointer using malloc() function.

```
#include <stdio.h>
#include<stdlib.h>
struct name {
   int a;
   float b;
   char c[30];
};
int main(){
   struct name *ptr;
   int i,n;
   printf("Enter n: ");
   scanf("%d",&n);
   ptr=(struct name*)malloc(n*sizeof(struct name));
/* Above statement allocates the memory for n structures with pointer ptr pointing to
base address */
   for(i=0;i<n;++i){
      printf("Enter string, integer and floating number  respectively:\n");
      scanf("%s%d%f",&(ptr+i)->c,&(ptr+i)->a,&(ptr+i)->b);
   }
   printf("Displaying Infromation:\n");
```

```
    for(i=0;i<n;++i)
        printf("%s\t%d\t%.2f\n",(ptr+i)->c,(ptr+i)->a,(ptr+i)->b);
    return 0;
}
```

Output

```
Enter n: 2

Enter string, integer and floating number  respectively:

Programming

2

3.2

Enter string, integer and floating number  respectively:

Structure

6

2.3

Displaying Information

Programming      2      3.20

Structure      6      2.30
```

STRUCTURES WITHIN STRUCTURES

Structures can be nested within other structures in C programming.

```
struct complex

{

 int imag_value;
```

```
 float real_value;
};
struct number{
   struct complex c1;
   int real;
}n1,n2;
```

Suppose you want to access *imag_value* for *n2* structure variable then, structure member*n1.c1.imag_value* is used.

UNIONS

Unions are quite similar to the structures in C. Union is also a derived type as structure. Union can be defined in same manner as structures just the keyword used in defining union in **union** where keyword used in defining structure was **struct.**

```
union car{
  char name[50];
  int price;
};
```

Union variables can be created in similar manner as structure variable.

```
union car{
  char name[50];
  int price;
}c1, c2, *c3;
```

OR;

```
union car{
  char name[50];
  int price;
};
-------Inside Function-----------
union car c1, c2, *c3;
```

In both cases, union variables *c1*, *c2* and union pointer variable *c3* of type **union car** is created.

Accessing members of an union

The member of unions can be accessed in similar manner as that structure. Suppose, we you want to access price for union variable *c1* in above example, it can be accessed as c1.price. If you want to access price for union pointer variable *c3*, it can be accessed as (*c3).price or as c3->price.

Difference between union and structure

Though unions are similar to structure in so many ways, the difference between them is crucial to understand. This can be demonstrated by this example:

```
#include <stdio.h>
union job {        //defining a union
   char name[32];
   float salary;
   int worker_no;
}u;
struct job1 {
   char name[32];
   float salary;
   int worker_no;
}s;
int main(){
```

```
    printf("size of union = %d",sizeof(u));
    printf("\nsize of structure = %d", sizeof(s));
    return 0;
 }
```

Output

```
 size of union = 32

 size of structure = 40
```

There is difference in memory allocation between union and structure as suggested in above example. The amount of memory required to store a structure variables is the sum of memory size of all members.

Fig: Memory allocation in case of structure

But, the memory required to store a union variable is the memory required for largest element of an union.

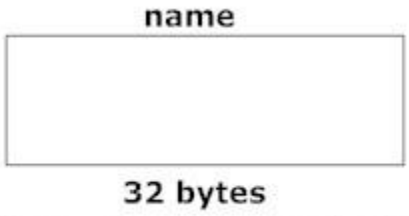

Fig: Memory allocation in case of union

What difference does it make between structure and union?

As you know, all members of structure can be accessed at any time. But, only one member of union can be accessed at a time in case of union and other members will contain garbage value.

```
 #include <stdio.h>

 union job {

    char name[32];
```

```
    float salary;
    int worker_no;
 }u;
 int main(){
    printf("Enter name:\n");
    scanf("%s",&u.name);
    printf("Enter salary: \n");
    scanf("%f",&u.salary);
    printf("Displaying\nName :%s\n",u.name);
    printf("Salary: %.1f",u.salary);
    return 0;
 }
```

Output

```
 Enter name
 Hillary
 Enter salary
 1234.23
 Displaying
 Name: f%Bary
 Salary: 1234.2
```

Note: You may get different garbage value of name.

Why this output?

Initially, *Hillary* will be stored in u.name and other members of union will contain garbage value. But when user enters value of salary, 1234.23 will be stored in u.salary and other members will contain garbage value. Thus in output, salary is printed accurately but, name displays some random string.

BIT FIELDS

In C, structure members can be specified with size in number of bits, and this feature is known as bit-fields. Bit-fields are important for low-level (i.e., for systems programming) tasks such as directly accessing systems resources, processing, reading and writing in terms of streams of bits (such as processing packets in network programming), cryptography (encoding or decoding data with complex bit-manipulation), etc.

Bit Field Declaration

The declaration of a bit-field has the following form inside a structure −

```
struct {
   type [member_name] : width ;
};
```

The following table describes the variable elements of a bit field −

Elements	Description
Type	An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
member_name	The name of the bit-field.
Width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

Example:

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

```
#include <string.h>

struct {
  unsigned int age : 3;
} Age;

int main( ) {

  Age.age = 4;
  printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
  printf( "Age.age : %d\n", Age.age );

  Age.age = 7;
  printf( "Age.age : %d\n", Age.age );

  Age.age = 8;
  printf( "Age.age : %d\n", Age.age );

  return 0;
}
```

ENUMERATIONS

An enumeration is a user-defined data type consists of integral constants and each integral constant is give a name. Keyword enum is used to defined enumerated data type.

```
enum type_name{ value1, value2,...,valueN };
```

Here, *type_name* is the name of enumerated data type or tag.

And *value1*, *value2*,....,*valueN* are values of type *type_name*.

By default, *value1* will be equal to 0, *value2* will be 1 and so on but, the programmer can change the default value.

```
// Changing the default value of enum elements
enum suit{
    club=0;
    diamonds=10;
    hearts=20;
    spades=3;
};
```

Example of enumerated type

```
#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};
int main(){
    enum week today;
    today=wednesday;
    printf("%d day",today+1);
    return 0;
   }
```

Output

4 day

TYPEDEF

The C programming language provides a keyword called **typedef**, which we can use to give a type, a new name. Following is an example to define a term **BYTE** for one-byte numbers −

```
typedef unsigned char BYTE;
```

After this type definition, the identifier BYTE can be used as an abbreviation for the type **unsigned char, for example.**.

```
BYTE  b1, b2;
```

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows −

```
typedef unsigned char byte;
```

We can use **typedef** to give a name to our user defined data types as well. For example, We can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows −

```
#include <stdio.h>

#include <string.h>

 typedef struct Books {

   char title[50];

   char author[50];
```

```
   char subject[100];

   int book_id;

} Book;


int main( ) {

   Book book;

   strcpy( book.title, "C Programming");
   strcpy( book.author, "Nuha Ali");
   strcpy( book.subject, "C Programming Tutorial");
   book.book_id = 6495407;

   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);

   return 0;
}
```

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

```
Book  title : C Programming
Book  author : Nuha Ali
Book  subject : C Programming Tutorial
Book  book_id : 6495407
```