<<Unit-4>>

Table of Contents

- Variable Argument-Command Line Argument
- Enumeration
- Bitfield
- Bitwise operator
- Typedef
- Preprocessor,macro expansion
- Local and global variable
  - Header file creation file inclusion

1.Variable Arguments(argc,argv)

Sometimes, you may come across a situation, when you want to have A function which can take variable number of arguments, i.e., parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the definition of such a function.

```
int func(int, ... )
{     .
    .
    .
}

int main()
{
   func(1,2,3);
func(1, 2, 3, 4);
}
```

- It should be noted that function func() has last argument as ellipses i.e. three dotes (**...**) and the one just before the ellipses is always an int which will represent total number variable arguments passed. To use such functionality you need to make use header file which provides functions and macros to implement the functionality of **stdarg.h** of variable
- Define a function with last parameter as ellipses and the one just before the ellipses is always an **int** which will represent number of arguments.
- Create a **va_list** type variable in the function definition. This type is defined in **stdarg.h** header file.
- Use **int** parameter and **va_start** macro to initialize the **va_list** variable to an The macro va_start is defined in stdarg.h header file.
- Use **va_arg** macro and **va_list** variable to access argument list. each item in argument list.
- Use a macro **va_end** to clean up the memory assigned to **va_list** variable.

Now let us follow the above steps and write down a simple function which can take variable number of parameters and returns their average:

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

double average(int num,...)
{

    va_list valist;
     double sum = 0.0;
    int i;

    /* initialize valist for num number of arguments */

   va_start(valist, num);

    /* access all the arguments assigned to valist */
     for (i = 0; i < num; i++)
    {
       sum += va_arg(valist, int);
    }
    /* clean memory reserved for valist */
     va_end(valist);

    return sum/num;
}

int main()
{
   printf("Average of 2, 3, 4, 5 = %f\n",  average(4, 2,3,4,5));
 printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

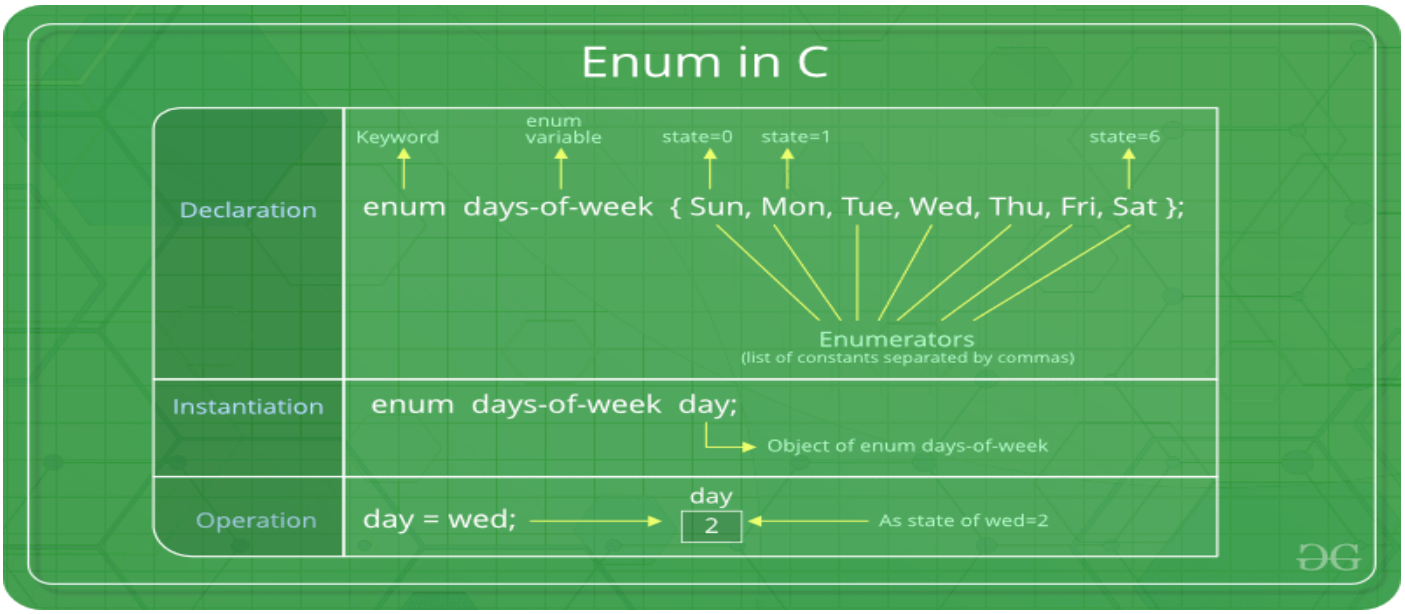
When the above code is compiled and executed, it produces the following result. It should be noted that the function average() has been called twice and each time first argument represents the total number of variable arguments being passed. Only ellipses will be used to pass variable number of arguments.

```
Average of 2, 3, 4, 5 = 3.500000

Average of 5, 10, 15 = 10.000000
```

2.Enumeration (or enum) 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.



The keyword 'enum' is used to declare new enumeration types in C and C++. Following is an example of enum declaration.

]

The name of enumeration is "flag" and the constant are the values of the flag. By default, the values of the constants are as follows:

constant1 =0, constant2 =1, constant3 =2 and so on.

```
enum flag{constant1, constant2, constant3, ..... };
```

Variables of type enum can also be defined. They can be defined in two ways:

In both of the below cases, "day" is defined as the variable of type week.

```
enum week{Mon, Tue, Wed};
enum week day;
```

```
// Or

enum week{Mon, Tue, Wed}day;
```

```
// An example program to demonstrate working
of enum in C

#include<stdio.h>

enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};

int main()

{

    enum week day;

    day = Wed;

    printf("%d",day);

    return 0;

}
```

Output:

2

In the above example, we declared “day” as the variable and the value of “Wed” is allocated to day, which is 2. So as a result, 2 is printed.

Another example of enumeration is:

```
// Another example program to demonstrate
working of enum in C

#include<stdio.h>

enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,

          Aug, Sep, Oct, Nov, Dec};

int main()

{

   int i;

   for (i=Jan; i<=Dec; i++)

   {

      printf("%d ", i);

   }

   return 0;

}
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

In this example, the for loop will run from i = 0 to i = 11, as initially the value of i is Jan which is 0 and the value of Dec is 11.

Interesting facts about initialization of enum.

1. Two enum names can have same value. For example, in the following C program both 'Failed' and 'Freezed' have same value 0.

```
#include <stdio.h>//5

enum State {Working = 1, Failed = 0,
Freezed = 0};



int main()

{

   printf("%d, %d, %d", Working, Failed,
Freezed);

   return 0;

}
```

Output:

```
1, 0, 0
```

2. If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0. For example, in the following C program, sunday gets value 0, monday gets 1, and so on.

```
#include <stdio.h>//6

enum day {sunday, monday, tuesday,
wednesday, thursday, friday,
saturday};



int main()

{

    enum day d = thursday;

    printf("The day number stored in d
is %d", d);

    return 0;

}
```

Output:

```
The day number stored in d is 4
```

3. We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

```
#include<stdio.h> //7

enum day {sunday = 1, monday, tuesday = 5,

          wednesday, thursday = 10, friday, saturday};

int main()

{

    printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,

             wednesday, thursday, friday, saturday);

    return 0;

}
```

Output:

```
1 2 5 6 10 11 12
```

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'
```

Enum vs Macro

We can also use macros to define names constants. For example we can define 'Working' and 'Failed' using following macro.

```
#define Working 0

#define Failed 1

#define Freezed 2
```

There are multiple advantages of using enum over macro when many related named constants have integral values.

a) Enums follow scope rules.

b) Enum variables are automatically assigned values. Following is simpler

```
enum state  {Working, Failed, Freezed};
```

3.Bit Fields

Suppose your C program contains a number of **TRUE/FALSE** variables grouped in a structure called status, as follows:

```
struct
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

This structure requires 8 bytes of memory space but in actual we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situation. If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, above structure can be re-written as follows:

```
struct
{
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status;
```

Now, the above structure will require 4 bytes of memory space for status variable but only 2 bits will be used to store the values. If you will use up to 32 variables each one with a width of 1 bit , then also status structure will use 4 bytes, but as soon as you will have 33 variables, then it will allocate next slot of the memory and it will start using 64 bytes. Let us check the following example to understand the concept:

```
#include <stdio.h> //8
#include <string.h>

/* define simple structure */ struct
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
```

```
struct
{
  unsigned int widthValidated : 1;
unsigned int heightValidated : 1;
} status2;

int main( )
{
   printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

   return 0;
}
```

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

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

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

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

Below the description of variable elements of a bit field:

Elements	Description
type	An integer type that determines how the bit-field's value is interpreted. The type may be int, signed int, 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.

The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit for example if you need a variable to store a value from 0 to 7 only then you can define a bit field with a width of 3 bits as follows:

```
struct
{
  unsigned int age : 3;
} Age;
```

The above structure definition instructs C compiler that age variable is going to use only 3 bits to store the value, if you will try to use more than 3 bits then it will not allow you to do so. Let us try the following example:

```
#include <stdio.h> //9
#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;
}
```

When the above code is compiled it will compile with warning and when executed, it produces the following result:

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

4.Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

P	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101 A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60, which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000

>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

Try the following example to understand all the bitwise operators available in C programming language

```
#include <stdio.h> //10
 main()
{
    unsigned int a = 60;
/* 60 = 0011 1100 */      unsigned int
b = 13;
/* 13 = 0000 1101 */
int c = 0;

   c = a & b;
/* 12 = 0000 1100 */
 printf("Line 1 - Value of c is %d\n",c);

   c = a | b;       /* 61 = 0011 1101 */
printf("Line 2 - Value of c is %d\n", c );

   c = a ^ b;       /* 49 = 0011 0001 */
printf("Line 3 - Value of c is %d\n", c );

   c = ~a;          /*-61 = 1100 0011 */
printf("Line 4 - Value of c is %d\n", c );

   c = a << 2;     /* 240 = 1111 0000 */
printf("Line 5 - Value of c is %d\n", c );

   c = a >> 2;     /* 15 = 0000 1111 */
   printf("Line 6 - Value of c is %d\n", c );

 }
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Defining Constants

There are two simple ways in C to define constants:

1. Using **#define** preprocessor.

The #define Preprocessor

Following is the form to use #define preprocessor to define a constant:

```
#define identifier value
```

Following example explains it in detail:

```
#include <stdio.h> //11

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main()
{
    int
area;

   area = LENGTH
*WIDTH;
   printf("value of
area : %d", area);
printf("%c", NEWLINE);

   return 0;
}
```

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

```
value of area : 50
```

5.Typedef

The C programming language provides a keyword called typedef, which you can us to give a type a new name.

Following is an example to define a term BYTE for one-byte numbers:

```
typedef unsigned char BYTE;
```

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;
```

You can use typedef to give a name to user defined data type as well. For example you 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> //12
#include <string.h>

typedef struct Books
{    char  title[50];    char  author[50];
     char  subject[100];    int   book_id;
} Book;

int main( )
{
```

```
   Book b;

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

   printf( "Book title : %s\n",b.title);
   printf( "Book author : %s\n",b.author);
printf( "Book subject : %s\n",b.subject);
printf( "Book book_id : %d\n",b.book_id);
    return 0;
}
```

It produces the following result:

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

typedef vs #define

The #define is a C-directive which is also used to define the aliases for various data types similar to typedef but with three differences:

- The **typedef** is limited to giving symbolic names to types only where as **#define** can be used to define alias for values as well, like you can define 1 as ONE etc.

- The **typedef** interpretation is performed by the compiler where as **#define** statements are processed by the pre-processor.

Following is a simplest usage of #define:

```
#include <stdio.h> //13

#define TRUE  1
#define FALSE 0

int main( )
{
   printf( "Value of TRUE : %d\n", TRUE);
printf("Value of FALSE : %d\n", FALSE);
   return 0;
}
```

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

```
Value of TRUE : 1
Value of FALSE : 0
```

6.Preprocessors

The C Preprocessor is not part of the compiler, but is a separate step in the compilation process.

In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP.

All preprocessor commands begin with a pound symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column. Following section lists down all important preprocessor directives:

Directive	Description
#define	Substitutes a preprocessor macro
#include	Inserts a particular header from another file
#undef	Undefines a preprocessor macro
#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Tests if a compile time condition is true
#else	The alternative for #if
#elif	#else an #if in one statement
#endif	Ends preprocessor conditional
#error	Prints error message on stderr
#pragma	Issues special commands to the compiler, using a standardized method

Preprocessors Examples :-Predefined Macros

ANSI C defines a number of macros. Although each one is available for your use in programming, the predefined macros should not be directly modified.

Macro	Description
__DATE__	The current date as a character literal in "MMM DD YYYY" format
__TIME__	The current time as a character literal in "HH:MM:SS" format
__FILE__	This contains the current filename as a string literal.
__LINE__	This contains the current line number as a decimal constant.
__STDC__	Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example:

```
#include<stdio.h> //14
 int main()
```

```
{
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );
}
```

When the above code in a file test.c is compiled and executed, it produces the following result:

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

```c
#include <stdio.h> //15

#define PI 3.1


int main()

{

    float radius, area;

    printf("Enter the radius: ");

    scanf("%f", &radius);


    // Notice, the use of PI

    area = PI*radius*radius;


    printf("Area=%.2f",area);

    return 0;

}
```

7. C Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed. There are three places where variables can be declared in C programming language:

- Inside a function or a block which is called local variables,
- Outside of all functions which is called global variables.
- In the definition of function parameters which is called formal parameters.

Let us explain what are local and global variables and formal parameters.

Local Variables

Variables that are declared inside a function or block are called **local variables**. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using **local variables**. Here all the variables a, b and c are local to **main()** function.

```c
#include <stdio.h> //16

int main ()
{
  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global Variables

Formal Parameters

Function parameters, so called **formal parameters**, are treated as local variables within that function and they will take preference over the global variables. Following is an example:

```
#include <stdio.h> //17

/* global variable declaration */ int
a = 20;

int main ()
{
  /* local variable declaration in main function */
int a = 10;   int b = 20;   int c = 0;

  printf ("value of a in main() = %d\n",  a);
c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);
   return
0;
}

/* function to add two integers */ int
sum(int a, int b)
{
    printf ("value of a in sum() = %d\n",  a);
printf ("value of b in sum() = %d\n",  b);

    return a + b;
}
```

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

```
value of a in main() = 10 value of a in sum() = 10 value

of b in sum() = 20 value of c in main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows:

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly otherwise, your program may produce unexpected results because uninitialized variables will take some garbage value already available at its memory location.

8.Header Files

A header file is a file with extension **.h** which contains C function declarations and macro definitions and to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that come with your compiler.

You request the use of a header file in your program by including it, with the C preprocessing directive **#include** like you have seen inclusion of **stdio.h** header file. which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be very much error-prone and it is not a good idea to copy the content of header file in the source files, specially if we have multiple source file comprising our program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in header files and include that header file wherever it is required.

Include Syntax

Both user and system header files are included using the preprocessing directive #include. It has following two forms:

```
#include <file>
```

This form is used for system header files. It searches for a file named file in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

```
#include "file"
```

This form is used for header files of your own program. It searches for a file named file in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

Include Operation

The **#include** directive works by directing the **C preprocessor** to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** directive. For example, if you have a header file header.h as follows:

and a main program called program.c that uses the header file, like this:

```
#include"header.c"
void main()
{
	add(3,5);
}
```

the compiler will see the same token stream as it would if program.c read

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

void add(int x ,int y)
{
int add=x+y;
printf("sum=%d",add);
}
```