UNIT 5 - Functions and User-defined Data Types

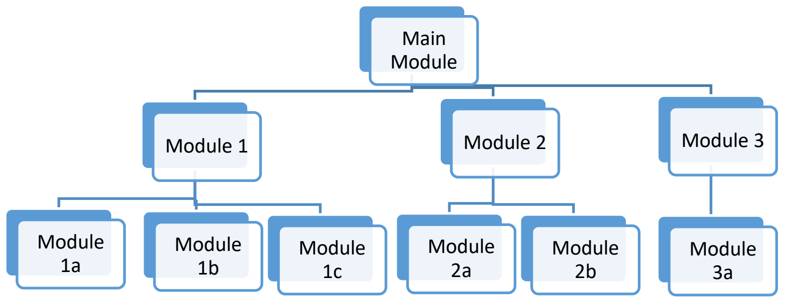
FUNCTIONS

Introduction

- C functions can be classified into two categories
  - Library functions
  - User- defined functions
- main is a user-defined function.
- printf and scanf are library functions.

Need for functions

- The program may be too large and complex to write in a single function **main**.
- The debugging, testing and maintenance becomes difficult.
- When a program is divided into functional parts, each part is independently coded and later combined to form a single unit.
- These subprograms are called as functions.
- They are easy to understand, debug and test.
- The planning done for a modular approach is top-down design.
  - Understand the problem as a whole.
  - Break it into simpler, understandable parts.
- The principle of top-down design and structured programming tells that
  - Program should be divided into a main module and its related module.
  - Each module can be further divided into submodules until the module consists of only elementary processes that are intrinsically understood and cannot be further subdivided.
  - This process is known as factoring



Advantages of Functions

- There are many advantages in using functions in a program.
- They are

- It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.
- The length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.
- It is easy to locate and isolate a faulty function for further investigation.
- A function may be used by many other programs this means that a c programmer can build on what others have already done, instead of starting over from scratch.
- A program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.
- Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program

Reasons for functions in C

- Many programs require that a specific function be repeated many times.
- Instead of writing the function code as many times as it is required it can be written as a single function.
- The same function can be accessed again and again as many times as it is required.
- To avoid writing redundant program code of some instructions again and again.
- Programs with using functions are compact & easy to understand.
- Testing and correcting errors is easy because errors are localized and corrected
- Understanding the flow of program and its code is easy since the readability is enhanced while using the functions.
- A single function written in a program can also be used in other programs also.

Elements of a user-defined function

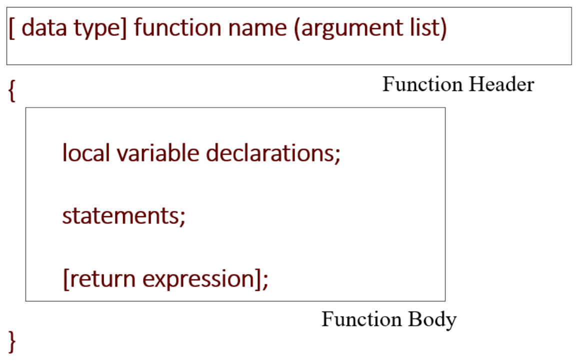
- Function definition
- Function call
- Function declaration

Function Definition

- It is also known as **function implementation**.
- It is an independent program module that is specially written to implement the requirements of the function.
- It should include
  - Function name
  - Function type
  - List of parameters

- Local variable declarations
- Function statements
- Return statement

- These are grouped into two parts
  - Function header
    - First three elements
  - Function body
    - Last three elements



```
[ data type] function name (argument list)
{
    local variable declarations;

    statements;

    [return expression];
}
```

Function Header

- It consists of three parts
  - Function type or return type
    - The function type specifies the type of value that it is expected to return the program calling the function.
    - If no type is specified, C assumes that it is an ***int*** type.
    - If the function is not returning any value the return type should be specified as ***void***.
  - Function name
    - The function name is a valid C identifier.
    - No duplication of library function names or operating system commands allowed.
  - Formal parameter list
    - It declares the variables that will receive the data sent from the calling program.
    - They are the input data to the function.
    - They can also be used to send values back to the calling programs.
    - The parameters are also called as arguments.
    - The list consists of declaration of variables separated by commas and enclosed within parenthesis.
- Example
  - int mul(int a,int b)
  - double sqroot(double x, int n)
- No semicolon to be added after the closing parenthesis.
- There may be functions which do not have formal parameters.
- Use the keyword void inside the parenthesis.

- Example
    - Ŧ void print(void)

Function Body

- It contains
    - Ŧ Local declarations that are need by the function
    - Ŧ Statements that perform the task of the function
    - Ŧ Return statement that returns the value computed by the function
- The order of the sub elements should be in the same order.
- If the function does not return any value the return statement can be omitted.
- It is better to use a return statement even if no value is returned.
- Examples

Ŧ double average(int x, int y)
{
    double sum;
    sum = x +y;
    return(sum/2);
}
	Ŧ int mul(int x, int y)
	{
	    return x*y;
	}
		Ŧ void display(void)
		{
		  printf("No Value inside");
		  return;
		}

Function Declaration

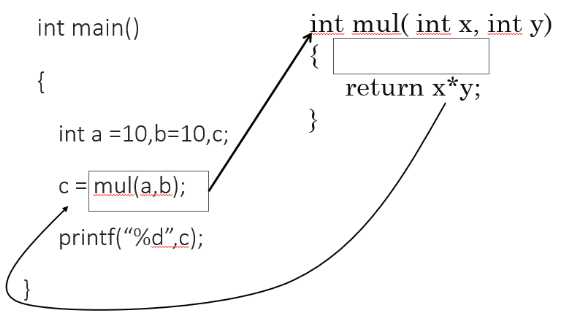
- It consists of only the function header.
- The header consists of three parts
    - Ŧ Return type
    - Ŧ Function name
    - Ŧ Formal parameter list
- The header should be terminated with a semicolon.
- **[data type] function name (argument list) ;**
    - Ŧ The argument list has to be separated by commas.
    - Ŧ The names of the arguments need not be the same in the function definition.
    - Ŧ The types must match the types of parameters in the function definition in number and order.
        - ϖ Use of parameter names in the declarations is optional.
    - Ŧ If the function has no formal parameters, the letter is written as (void).
    - Ŧ The return type is optional, when the function returns ***int*** type data.
    - Ŧ The return type must be ***void*** if no value is returned.

    - When the declared types do not match with the types in the function definition, compiler will produce an error.
    - When a function does not take any parameters, and does not return any value it is written as
        - void function_name(void);
    - The declaration can be placed in two places
        - Above all the functions
            - When the declaration is placed above all the functions it is referred as ***global prototype***.
            - These declarations are available for all the functions in the program.
        - Inside a function definition
            - When it is placed inside a function definition it is referred as ***local prototype***.

- Example
    - int multiply(int x, int y);
    - mul (int x, int y);
    - mul (int, int);
        - All are accepted formats

Function Call

- Use of the function name followed by a list of the actual parameters enclosed in parenthesis.
    - ***function-name(value1, value2,….);***
- When the compiler encounters a function call, the control is transferred to the function.
- The function is then executed line by line and a value is returned when a return statement is encountered.



```
int main()
{
  int a =10,b=10,c;
  c = mul(a,b);
  printf("%d",c);
}

int mul( int x, int y)
{
  return x*y;
}
```

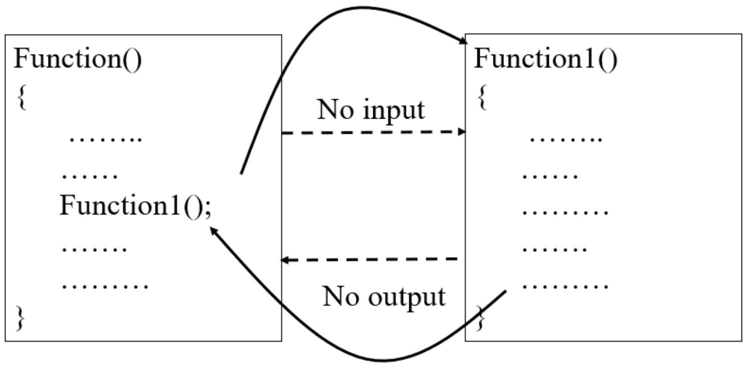
- Example
    - mul(6, 7);
    - mul(6, b);
    - mul(mul(a, b), 7);
    - mul(a, 7);
    - mul(a+6, 7);
    - mul(exp, exp);

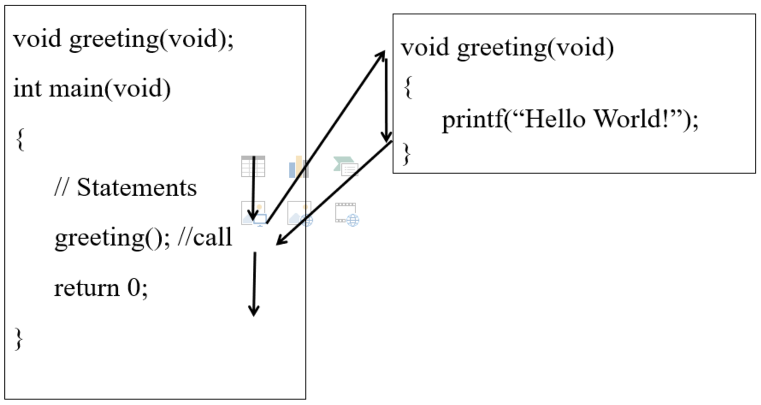
Category of Functions

- Function with no arguments and no return values
- Function with arguments and no return values
- Function with arguments and one return value

- Function with no arguments but returns a value
- Function that returns multiple values

void functions without parameters

- It does not receive any data from the calling function.
- Similarly the calling function does not receive any data from the called function.
- There is no data transfer between the calling function and called function.
- 

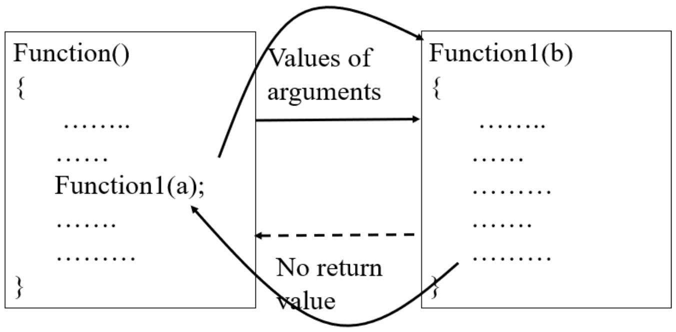
- 

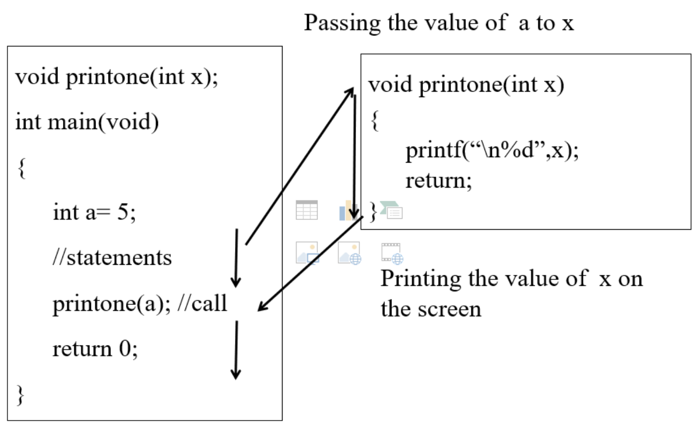
```
void greeting(void);
int main(void)
{
    // Statements
    greeting(); //call
    return 0;
}
```

```
void greeting(void)
{
    printf("Hello World!");
}
```

- Since a void function does not have a value it can be used only as a statement and cannot be used in an expression.

void functions with parameters

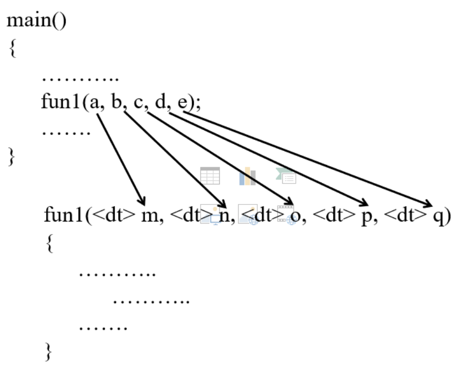
- The functions receive inputs from the calling function but does not send any data back to the called function.
- 

- 
Passing the value of a to x
Printing the value of x on the screen

```
void printone(int x);
int main(void)
{
      int a= 5;
      //statements
      printone(a); //call
      return 0;
}
```

```
void printone(int x)
{
      printf("\n%d",x);
      return;
}
```

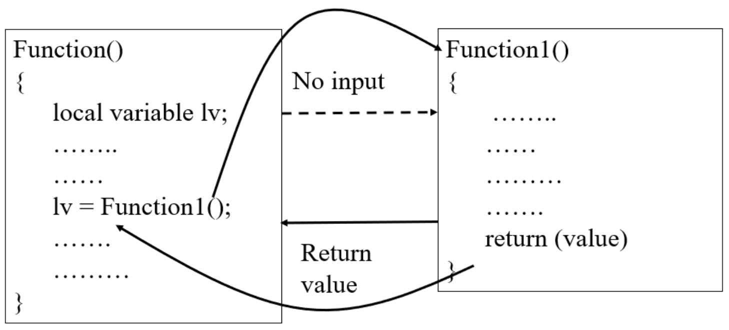
- Here ***a*** is known as the ***actual argument*** and ***x*** is known as the ***formal argument***.
- The formal and actual arguments must match in number, type and order.
- The values of the actual arguments are assigned to the values of the formal arguments on a one to one basis starting with the first argument.

- 

```
main()
{
      ……….
      fun1(a, b, c, d, e);
      …….
}

      fun1(<dt> m, <dt> n, <dt> o, <dt> p, <dt> q)
      {
            ……….
                  ……….
            …….
      }
```

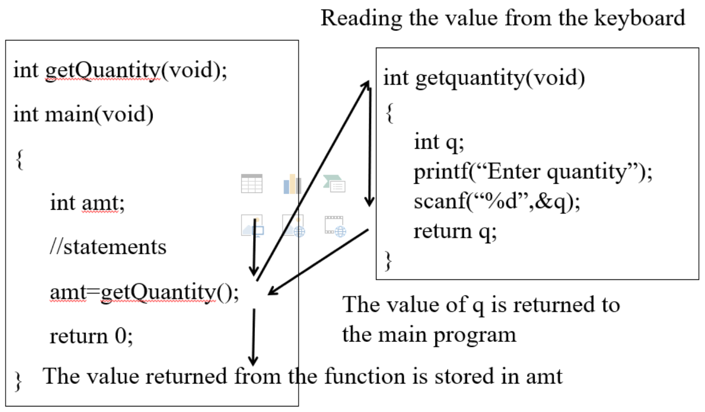
Non-void functions without parameters:

- Some functions return a value but do not have any parameters.
- Example
    - Read the data from the keyboard and return the data to the calling program.

- 

```
Function()
{
      local variable lv;
      ……..
      ……
      lv = Function1();
      …….
      ………
}
```

```
Function1()
{
      ……..
      ……
      ………
      …….
      return (value)
}
```

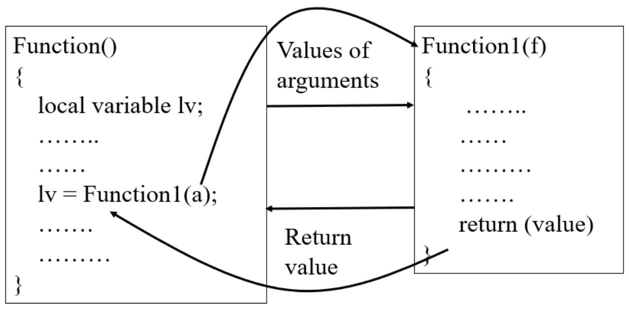

Reading the value from the keyboard
The value of q is returned to the main program
The value returned from the function is stored in amt

```
int getQuantity(void);
int main(void)
{
    int amt;
    //statements
    amt=getQuantity();
    return 0;
}
```

```
int getquantity(void)
{
    int q;
    printf("Enter quantity");
    scanf("%d",&q);
    return q;
}
```

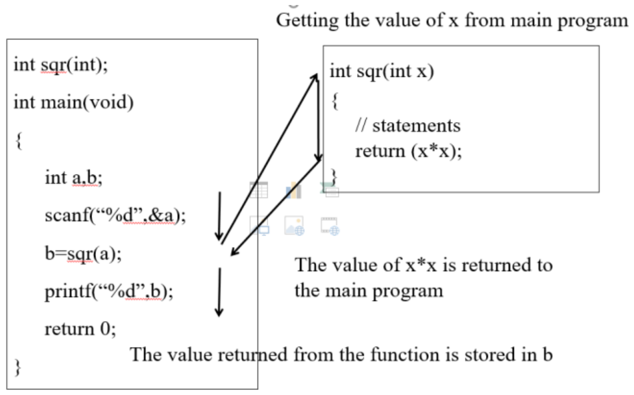
Non-void functions with parameters

- These are the functions that take inputs from the main program and returns a value back to it.
- These functions are treated as "black boxes" which are self-contained and independent modules.



```
Function()
{
   local variable lv;
   ……..
   ……
   lv = Function1(a);
   …….
   ………
}
```

```
Function1(f)
{
    ……..
   ……
   ………
   …….
   return (value)
}
```


Getting the value of x from main program
The value of x*x is returned to the main program
The value returned from the function is stored in b

```
int sqr(int);
int main(void)
{
    int a,b;
    scanf("%d",&a);
    b=sqr(a);
    printf("%d",b);
    return 0;
}
```

```
int sqr(int x)
{
    // statements
    return (x*x);
}
```

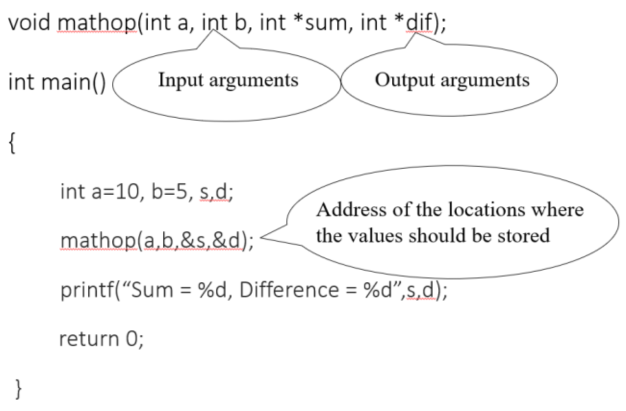
Returning float values:

- C function returns a value of the type ***int*** as the default case when no other type is specified explicitly.
- If the return type is not mentioned the integer part is sent to the main program.
- If it is assigned to a float value the truncated integer value is converted to a float.

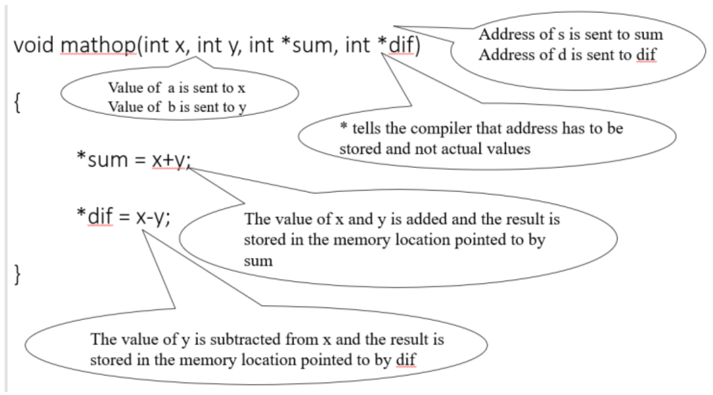
Functions that return multiple values

- A return statement can return only one value.

- To return multiple values we use the arguments not only to receive the input but also return back the output.
- The arguments that are used to send the data back are called as ***output arguments***.
- To send the data we use the ***address operator*** and ***indirection operator***.
- The address operator & and indirection operator * are used.



```
void mathop(int a, int b, int *sum, int *dif);

int main()

{

    int a=10, b=5, s,d;

    mathop(a,b,&s,&d);

    printf("Sum = %d, Difference = %d",s,d);

    return 0;

}
```



```
void mathop(int x, int y, int *sum, int *dif)

{

    *sum = x+y;

    *dif = x-y;

}
```

Address of s is sent to sum
Address of d is sent to dif
Value of a is sent to x
Value of b is sent to y
* tells the compiler that address has to be stored and not actual values
The value of x and y is added and the result is stored in the memory location pointed to by sum
The value of y is subtracted from x and the result is stored in the memory location pointed to by dif

  - The variables ****sum*** and ****dif*** are known as ***pointers***.
  - ***sum*** and ***dif*** are ***pointer variables*** of the type ***int***.

Scope of a Function

- Each function is a discrete block of code.
- Function defines **block scope**.
  - Function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function.
  - Code and data defined within one function cannot interact with the code or data defined in another function.
    - Each function has a different scope.

Local Variable

- Variables that are defined within a function are **local variables.**
  - Comes into existence when the function is entered and is destroyed upon exit.

    - It cannot hold its value between function calls.
    - Only static local variable can hold its value.
        - Compiler treats the variable as if it were a global variable for storage specification but limits its scope to the function.

- Formal parameters to a function also fall within the function's scope.
    - The parameter is known throughout the function.
    - The parameter comes into existence when the function is called and destroyed when the function is exited.
- A function **cannot be defined** within another function.

Call by value and call by reference

- Arguments can be passed to a function in two ways
    - Call by value
        - Copies the value of an argument into the formal parameter of the subroutine.
        - Changes made to the parameter have no effect on the argument.
    - Call by reference
        - Address of an argument is copied into the parameter.
        - The address is used to access the actual argument used in the call.
        - Changes made to the parameter affect the arguments.

Passing of pointer as parameters

- The type of the actual and the formal parameter should be the same.
- The actual arguments must be the addresses of the variables that are local to the called function.
- The formal arguments in the header must be prefixed by the indirection operator.
- The arguments must be prefixed by the indirection operator even in the declaration.
- To access the value of the actual argument in the called function, the corresponding formal argument must be prefixed with the indirection operator.

Nesting of functions

- C permits nesting of two functions freely.
- There is no limit how deeply functions can be nested.

Recursion

- Recursive function is a function that calls itself.
- When a function calls another function and that second function calls the third function then this kind of a function is called nesting of functions.
- A recursive function is the function that calls itself repeatedly.

-
  ```
  int main()
  {
          printf("this is a use of recursive function");
          main();
    return 0;
  }
  ```
    - The line is printed repeatedly and indefinitely.
    - Terminate the execution abruptly.
- These functions are effectively used to solve the problems where the solution is expressed in terms of successive application of the same solution to the subsets of the problem.
- Whenever a recursive function is written we need to have a terminating function that forces the function to return without executing the recursive part.

Functions and Arrays

- An entire one-dimensional array of values can be passed into a function just as we pass individual variables.
- In this task it is essential to list the name of the array along with functions arguments without any subscripts and the size of the array as arguments.
- Example
    - smallest(a, n);
        - This will pass the whole array a to the called function smallest.
        - The header of this function should look like
        - int smallest(int arr[], int row)

It informs the compiler that arr is an array of integers

It informs the compiler that arr is an array

Rules

- The function must be called by passing only the name of the array.
- In the function definition the formal parameter must of the array type.
    - The size of the array need not be specified.
- The function prototype should also show that the argument is an array.

Functions and two-dimensional arrays

- The function must be called by passing only the array name.
- The array must be indicated as having two dimensions by including two sets of brackets in the function definition.
- The size of the second dimension must be specified.

- The prototype declaration must be similar to the function header.

```
• float average(int [][20], int, int);

int main()
{
        int m,n,i,j, arr[20][20];
         float mean;
        scanf("%d %d",&m,&n);
        for(i=0;i<m;i++)
           for(j=0;j<n;j++)
                scanf("%d",&arr[i][j]);
         mean = average(arr, m, n);
        printf("%f",mean);
}
```

```
• float average(int x[][20], int m, int n)
{
        int i,j; float sum=0.0;
        for(i=0;i<m;i++)
            for(j=0;j<n;j++)
                 sum=sum+x[i][j];
         return(sum/(m*n));
}
```

Functions and Strings

- The string must be declared as a formal argument of the function.
- The function prototype must show that argument is a string.
- A call to the function must have a string name without subscripts in its actual argument.

Arguments to main()

- C does not allow user defined parameters to be passed to main().
- Two system-defined arguments can be passed to main()
    - ₮ int argc and char *argv[]
        - ϖ argc counts the number of parameters passed.
        - ϖ argv is an array of strings
            - ℓ Each string represents a parameter that is passed to main

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char const *argv[])
{
    if(argc!=2)
    {
        printf("You forgot the name\n");
        exit(1);
    }
    printf("Hello %s\n", argv[0]);
    printf("Welcome %s", argv[1]);
    return 0;
}
```

```
#include <stdio.h>
int main(int argc, char const *argv[])
{
    printf("The number of arguments is %d\n", argc);
    printf("The name of the program is %s\n",argv[0]);
    for (int i = 1; i < argc; ++i)
        printf("User value No. %d is %s\n",i, argv[i]);
    return 0;
}
```

```
#include <stdio.h>
int main(int argc, char const *argv[])
{
    int t, i;
    for(t=0; t<argc; ++t)
    {
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char const *argv[])
{
    int disp, count;
```

```
        i=0;
        while(argv[t][i])
        {
            printf("String %d, char %d
is \t", t, i);
            putchar(argv[t][i]);
            ++i;
        }
        printf("\n");
    }
    return 0;
}
```

```
    if(argc<2)
    {
        printf("Enter the length of the count on the command
line\n");
        exit(1);
    }
    if(argc==3 && !strcmp(argv[2], "display"))
        disp =1;
    else
        disp =0;
    for(count=atoi(argv[1]);count;--count)
        if(disp)
               printf("%d\n", count);
    putchar('\a');
    printf("Done");
    return 0;
}
```

Programming Examples

- Explain the significant needs of user-defined functions.
- Write the general structure of a user defined function and explain the working principle.
- Define a function to calculate $\sqrt{a_0b_0 + a_1b_1 + a_2b_2 + \ldots\ldots + a_{n-1}b_{n-1}}$
- Write a recursive program to find the GCD of 2 numbers.
- Write a recursive program to find the binary equivalent of an integer.
- Write a C program using functions to evaluate the expression $3x^2 + \sin x$. The value of x is passed to the function.

Questions

- What is recursion.
- Explain the categories of functions with examples.
- What is recursion.

User-Defined Data Types

STRUCTURES

- Arrays are used to store large set of data and manipulate them but the disadvantage is that all the elements stored in an array are to be of the same data type.
- If a collection of different data type items needs to be used it is not possible using an array.
- To use a collection of different data items of different data types a structure needs to be used.
- Structure is a method of packing data of different types.
- A structure is a convenient method of handling a group of related data items of different data types.

- Structure is a collection of variables referenced under one name providing a convenient way of keeping related information together.
- A structure declaration forms a template that can be used to create structure objects.
- Variables that make up a structure are called members.
    - They are also called as elements or fields.

Defining a structure

- C has two ways to declare a structure
    - Tagged structure
    - Type-defined structures.

Tagged Structure

- The structure starts with the keyword ***struct***.
- The second element of the declaration is the ***tag***.
    - It is the identifier for the structure that allows us to use it for other purposes.
- If the structure is enclosed with a semicolon after the closing braces no variables are defined.
    - Such a structure is a type template with no associated storage.

struct tag_name
{
    data type member1;
    data type member2;
    …
};
	struct STUDENT
	{
	    char id[10];
	    char name[20];
	    int marks;
	};

Type-defined Structures

- The *typedef* is used to declare a structure.
- The keyword typedef is needed in the beginning of the declaration and an identifier is required at the end of the block.
    - The identifier is the type definition name.
    - typedef struct {field list} TYPE;
    - typedef struct{char id[10]; char name[20]; int marks;} STUDENT;

Variable Declaration

- After a structure has been declared, variables can be declared using it.
- The type is normally declared in the global area to make it visible to all functions.
- The variables are usually declared in the functions.

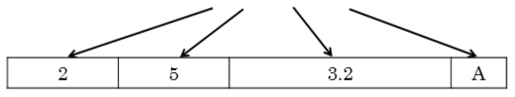
struct STUDENT{
char id[10];
char name[20];
int marks;
};
struct STUDENT stu;
	typedef struct
	{
	char id[10];
	char name[20];
	int marks;
	} STUDENT;
	STUDENT stu;

Initialization

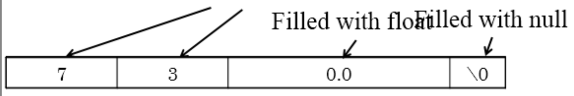
- The initializers are enclosed in braces and separated by commas.
- The initializers must match their corresponding types in the structure definition.

```
typedef struct
        {
                int x;
                int y;
                float t;
                char u;
        } SAMPLE;
```

SAMPLE sample1{2, 5, 3.2, ‘A’};



SAMPLE sample2{7,3};



Rules

- C language does not permit the initialization of individual structure members within the template.
- The order of values enclosed in braces must match the order of members in the structure definition.
- Partial initialization is allowed.
    - The uninitialized members should be only at the end of the list.
- The uninitialized members will be assigned default values as follows:
    - Zero for integer and floating-point numbers.
    - ‘\0’ for characters and strings

Accessing Structure Members

- Referencing individual fields.
    - Each field in the structure can be accessed and manipulated using expressions and operators.

- To refer to a field in a structure refer both the structure and the field.
    - First use the structure variable-identifier and then the field identifier.
    - The structure variable identifier is separated from the field identifier by a dot.
    - The dot is a direct selection operator that is a postfix operator.

- While using scanf to read the values form the keyboard
    - scanf("%s%s%d",stu.id, stu.name, &stu.marks);
- To access the members of the structure STUDENT.
- Example 1

typedef struct { char id[10]; char name[20]; int marks;} STUDENT;

STUDENT stu;

stu.id, stu.name, stu.marks

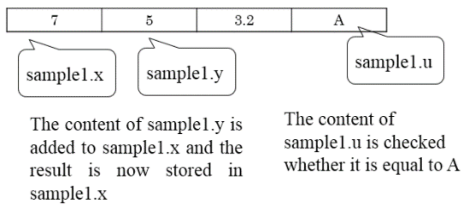
- Example 2

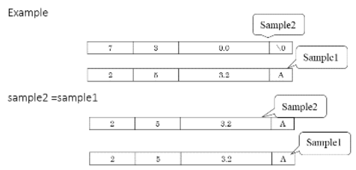
typedef struct {int x; int y; float t; char u; } SAMPLE;

SAMPLE sample1;

- Use a selection operator on sample1 to evaluate the field u.
    - If it is an A then add the two integer fields and store the result in the first field.
        - if(sample1.u==‘A’)

sample1.x += sample1.y


7
5
3.2
A
sample1.x
sample1.y
sample1.u
The content of sample1.y is added to sample1.x and the result is now stored in sample1.x
The content of sample1.u is checked whether it is equal to A

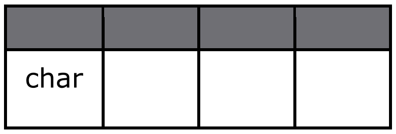
- Information contained in one structure can be assigned to another structure.
- Two variables of the same structure can be copied in the same way as ordinary variables.
- Example
    - stu1 = stu2;
    - 

Comparing structure variables

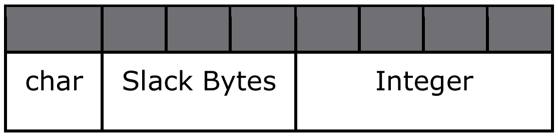
- C does not permit any logical operations on structure variables.
- Comparison has to be done on individual members.

Word boundaries and Slack bytes

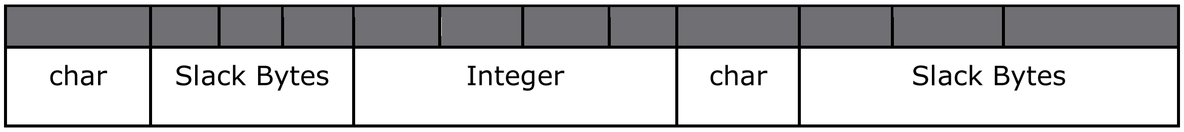
- To store any type of data in structure there is minimum fixed byte which must be reserved by memory.
- This minimum byte is known as **word boundary**.
- Word boundary depends upon machine.
    - Ŧ On a 32-bit machine it can be 4 bytes.
    - Ŧ On a 64-bit machine it can be 8 bytes.
        - ϖ So any data type reserves at least 8 bytes space.
        - ϖ In an X86 machine it uses half word – 4 bytes
- Example
    - Ŧ struct word1{char a; int b; char c;};
        - ϖ First of all char a will reserve four bytes and store the data in only first byte (size of char is one byte).



        - ϖ Now int b(size of int four byte) will search four bytes and since there are remaining only three bytes it will search for new four bytes.



        - ϖ The remaining three bytes of the half word that is useless is called as slack bytes.
        - ϖ Now char c requires one byte and will occupy another half word of four bytes.
        - ϖ It uses one byte and the remaining three bytes are useless and called as slack bytes.



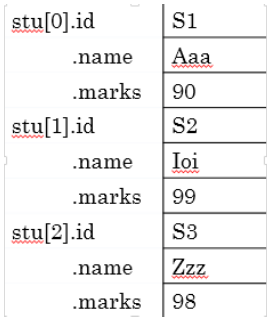
Operations on Individual members

- The assignment operator will work on the entire structure entity only.

- Each individual structure member can be compared.
- Increment and decrement operators can be applied to the numeric members of the structure.
- The precedence of the member operator is higher than all the arithmetic and relational operators.

Array of Structures

- It is possible to define a array of structures.
- For example if we are maintaining information of all the students in the college and if 50 students are studying in the college.
- STUDENT stuarr[50];
    - This is an array that contains the name, id and marks of 50 students.



stu[0].id	S1
.name	Aaa
.marks	90
stu[1].id	S2
.name	Ioi
.marks	99
stu[2].id	S3
.name	Zzz
.marks	98

- C allows the use of arrays as structure members.

Structures within structures

- When a structure includes another structure it becomes a nested structure.
- There is no limit to the number of structures that can be nested.
    - Normally the limit is three.

Declaring Nested Structures

- It is easier and simpler to follow if each structure is declared separately and then grouped into a high-level structure.
- The nesting must be done from inside out.
    - The nesting is done from lowest level to most inclusive level.
    - Declare the innermost structure and then move to the outer structure.
- Structure stamp stores date and time.
- Date is a structure that stores month, day and year.
- Time is a structure that stores hour, minute and second.
-

typedef struct
{ int mon, day, year;
}DATE;
	typedef struct
	{ int hours, min, sec;
	}TIME;
		typedef struct
		{ DATE date;
		TIME time;
		}STAMP;
		STAMP stamp;

- It is possible to nest the same structure type more than once in a declaration.

- Structure that contains start and end times fo a job.
- typedef struct

  {   STAMP start; STAMP end;   } JOB;

  JOB job;

Referencing Nested Structures

- To access a nested structure, each level from the highest to the component being referenced must be included.
- Example
  - stamp.date
  - stamp.time.hour;
  - job.start.time.hour

Nested Structure Initialization

- Each structure must be completely initialized before proceeding to the next member.
- Each structure must be initialized in a set of braces.
- The definition and initialization for stamp is given as
- STAMP stamp = {{5,12,2009},{1,30,47}};

Structures and Functions

- A function can access the members of the structure in three ways.
  - Individual members can be passed to the function.
  - The whole structure can be passed and the function can access the members by using pass by value.
  - The address of the structure or member can be passed and the function can access the members through indirection and indirect selection.

Sending individual members

- Example
  - r.num = mul(fr1.num,fr2.num);
  - r.den = mul((fr1.den,fr2.den);

    mul(int x, int y)

    { return x*y; }

Sending whole structures

- Specify the type in the formal argument of the called function.
- The function can return a structure.
- C will copy the values to a local structure.

- Passing structures is call by value.

Structure Pointers

- Pointers to structures can be created.
- Syntax
    - struct <struct_name> *<struct_pointer>;
- Use of structure pointers
    - To pass a structure to a function call by reference
    - To create linked lists and other dynamic data structures
- When a pointer to a structure is passed to a function, only the address of the structure is sent.

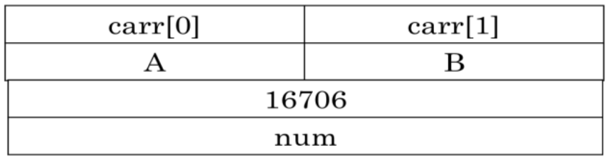
UNIONS

- It is a construct that allows memory to be shared by different types of data.
- union tag_name

```
{
        data_type member1;
        data_type member1;
        …..
 };
```

- Each piece of data starts at the same memory location and occupies at least a part of the same memory.
- When a union is shared by two or more different data types only one piece of data can be in memory at one time.
    - union sharedata

```
{
    char carr[2];
    short num;
};
```



Referencing a Union

- The rules for referencing a union are identical to that of the structures.
- To reference the individual fields within the union, we use direct selection operator.
- Each reference must be fully qualified from the beginning of the collection to the element being referenced.
- When a union is being referenced through a pointer the selection operator can be used.
    - sharedata.num
    - sharedata.carr[0]

Initializers

- Only the first type declared in a union can be initialized when the variable is defined.
- The other types can be initialized by assigning values or reading values into the union.
- When initializing enclose the values in a set of braces.

Bit Fields

- Bit field is a built-in feature of C that allows the user to access a single-bit.
- Reasons for bit fields
  - Ŧ If storage is limited, several Boolean variables can be stored in one byte.
  - Ŧ Certain devices transmit status information encoded into one or more bits within a byte.
  - Ŧ Certain encryption routines need to access the bits within a byte.
- An int or unsigned member of a structure or union can be declared to consist of a specified number of bits.
- Such a member is called as a bit field.
- The number of associated bits is called its width.
- The width is specified by a non-negative constant integral expression following a colon.
- The width is at most the number of bits in a machine word.
- Bit fields are normally declared as consecutive members of the structure.
- The compiler packs them into a minimal number of machine words.
- typedef struct

```
{
        unsigned data_type : bit size;
        unsigned data_type : bit size;
}tag_name;
```

- Whether the compiler assigns the bits in left-to-right order is machine dependent.
- Arrays of bit fields are not allowed.
- The address operator & cannot be applied to bit fields.
- Pointers cannot be used to address a bit field directly.
- The use of the member access -> is acceptable.

#include <stdio.h>
// A simple representation of date
struct date
{
        unsigned int d;
        unsigned int m;
        unsigned int y;
};
	#include <stdio.h>
	// A structure without forced alignment
	struct test1
	{
	   unsigned int x: 5;
	   unsigned int y: 8;
	};
	// A structure with forced alignment

```
struct date1
{
// d has value between 1 and 31, so 5 bits are
//sufficient
    unsigned int d: 5;
// m has value between 1 and 12, so 4 bits are
//sufficient
    unsigned int m: 4;
    unsigned int y;
};
int main()
{
    printf("Size of date is %d bytes\n",
    sizeof(struct date));
    struct date dt = {29, 12, 2017};
    printf("Date is %d/%d/%d\n", dt.d, dt.m,
    dt.y);
    printf("Size of date is %d bytes\n",
    sizeof(struct date1));
    struct date1 dt1 = {29, 12, 2017};
    printf("Date is %d/%d/%d\n", dt1.d,
    dt1.m, dt1.y);
    return 0;
}
/*
    Size of date is 12 bytes
    Date is 29/12/2017
    Size of date is 8 bytes
    Date is 29/12/2017
*/
```

```
struct test2
{
  unsigned int x: 5;
  unsigned int: 0;
  unsigned int y: 8;
};
int main()
{
  printf("Size of test1 is %d bytes\n",
sizeof(struct test1));
  printf("Size of test2 is %d bytes\n",
sizeof(struct test2));
  return 0;
}
/*
Size of test1 is 4 bytes
Size of test2 is 8 bytes
*/
```

•

Enumeration

- Enumeration is a set of named integer constants.
- They are defined like structures.
- **enum tag {enumeration list} variable_list;**
    - Ŧ Tag and variable_list are optional.
        - ϖ One of the two should always be present.

```
#include <stdio.h>

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

{

        enum choc {toffee, candy, dark, milk, exotic, bitter, nuts};

        enum choc chocolate;

        int c;

        printf("Enter any number between 0 and 6 (both included)\n");
        scanf("%d", &chocolate);
        printf("Your choice of chocolate is ");
        switch(chocolate)
        {
                case toffee :printf("Toffee\n"); break;
                case candy :printf("Candy\n"); break;
                case dark : printf("Dark\n"); break;
                case milk : printf("Milk\n"); break;
                case exotic :printf("Exotic\n"); break;
                case bitter :printf("Bitter\n"); break;
                case nuts :printf("Nuts in chocolate\n"); break;
                default : printf("Unknown\n");
        }
        return 0;
}
```

Programming Examples

- Create a structure of employee having the following info – employee id, employee name, date of joining, salary. Write a C program to input info of 20 employees and display the details of the specified employee given the employee id.
- Declare a structure to represent a complex number. Write a function to add the two complex numbers using this structure.

Questions

- What is a structure? Give the syntax for defining a structure in C. explain how the individual members of the structure are accessed.
- Explain unions briefly.
- Explain in brief how structures are different from unions.
- Define a structure. Give syntax and example of structure declaration.