Agenda of This Session

- Conditional looping control statements: while, do-while, for
- Unconditional control statements: break, continue, goto, Functions
- Arrays, Strings, Pointers, Call by value and reference
- Compound Data Types: Struct, Union, enum, typedef

Switch – Case

- Used to make selections from a number of choices
- Syntax:

```
switch (variable or an integer expression)
{
    case constant:
    //Statements;
    break;
    case constant:
    //Statements;
    break;
    default:
    //Statements;
    break;
}
```

```
int main()
{
   int num=2;
   switch(num+2)
   {
      case 1:
         printf("Case1 ");
         break;
       case 2:
         printf("Case2 ");
         break;
       case 3:
         printf("Case3 ");
         break;
       case 4:
         printf("Case4 ");
         break;
       default:
         printf("Default ");
   }
   return 0;
}
```

Looping Control Statement

- Performs looping operations until given condition becomes false.
- while loop
- do…while loop
- for loop

- Conditional statement executes only once in the program where as looping statements executes repeatedly several number of time.

- Infinite loop: when loop condition is never false.

While Loop

- repeatedly evaluates *expression* and, if non-zero, executes *statement*
- Syntax:

```
while (expression)
                    statement;


OR


while (expression)
          {
                    block of statements;
          }
```

```
#include <stdio.h>
main()
{
   int i = 10;

   while ( i > 0 )
   {
     printf("%d\n", i );
     --i;
   }
}
```

Do…while Loop

- Like while but test condition is checked at the end of the loop rather than the start resulting in atleast one execution of statement.
- Syntax:

```
do
{
   Single statement;
   or
   Block of statements;
}while(expression);
```

```
#include <stdio.h>
main()
{
   int i = 10;

   do{
      printf("%d\n", i );
      --i;
   }while ( i > 0 );
}
```

For Loop

- Most commonly used and most popular because of simplicity.
- Syntax:

```
for ( initialization ; condition ; update )
{
   Single statement;
   or
   Block of statements;
}
```

- initialization: Initialises variables.
- condition: Condtional expression, as long as this condition is true, loop will keep executing.
- update: simple increment/decrement of a variable.

```
#include <stdio.h>
int main()
{
   int i; //Used as a counter variable.

   for( i=1 ; i<=10 ; i++)
   {
      printf("%d\n",i);
   }

   return 0;
}
```

Unconditional control Statements

- C provides two commands to control how we loop:

- break -- exit form loop or switch.
- continue -- skip 1 iteration of loop.

- and 2 commands to transfer control to another part of the program.
- goto label -- jump to a labelled statement
- return -- used only in functions

Unconditional Control Statement cont'd

break

- Break statement is used to terminate loops & switch case statements from the subsequent execution.
- Syntax: break;
- Example:

```
for(i=0;i<10;i++)
   {
    if(i==5)
    {
       printf("\nComing out of for loop when i
= 5");
       break;
    }
    printf("%d ",i);
   }
```

continue

- Continue statement is used to continue the next iteration of loops, after skipping the current iteration.
- Syntax : continue;
- Example:

```
for(i=0;i<10;i++)
   {
    if(i==5 || i==6)
    {
      printf("\nSkipping %d from display using
" \
      "continue statement \n",i);
      continue;
    }
    printf("%d ",i);
   }
```

Unconditional Control Statement cont’d

goto

- goto statements is used to transfer the normal flow of a program to the specified label in the program.
- Example:

```
#include <stdio.h>
int main()
{
  int i;
  for(i=0;i<10;i++)
    {
  if(i==5)
  {
   printf("\nWe are using goto statement when i = 
5");
   goto HAI;
  }
  printf("%d ",i);
}
HAI : printf("\nNow, we are inside label name \"hai\"
\n");
}
```

return

- The return statement ends the current function and returns control in the point of invocation
- Syntax: return; OR return value;
- Example:

```
void main()
{
    int sum = sumDigits();
    printf("%d\n", sum);
    return;
}
```

Arrays

- collection of data items of the same type having a common name
- 2 types
  - One dimensional array
  - Multi-dimensional array
- Visually a one-dimensional array is like a list
- A multi dimensional array is like a table.

- i.e. *one-dimensional arrays are vectors, two-dimensional arrays are matrices*

One Dimensional Array

- syntax:
- data_type array_name[array_size];

- Example:

float floatArray[1000];

OR

const int NROWS = 100;   // (OR #define NROWS 100)

const int NCOLS = 200;   // (OR #define NCOLS 200)

float matrix[NROWS][NCOLS];

- Array initialization during declaration, example:

- int mark[5] = {19, 10, 8, 17, 9};
- int mark[] = {19, 10, 8, 17, 9};

Multi- Dimensional Array

- Syntax:

data_type array_name[size1][size2]...[sizeN];

Example:

- float x[3][4];
- float y[2][4][3];

- Array initialization during declaration, example:
- int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
- int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
- int c[2][3] = {1, 3, 0, -1, 5, 9};
- int test[2][3][4] = {

{ {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },

{ {13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9} }

};

Strings

- Array of characters that ends with a null character.
- Example:
- String “abc” is actually stored as

‘a’ ‘b’ ‘c’ ‘\0’

- Technically, in a fifty char array you could only hold 49 letters and one null
- Example:

char string[50];

- Initialization of Strings, Example:

char c[] = "abcd";

OR,

char c[50] = "abcd";

OR,

char c[] = {'a', 'b', 'c', 'd', '\0'};

OR,

char c[5] = {'a', 'b', 'c', 'd', '\0'};

Example Program:

```
#include <stdio.h>
int main()
{
    /* A nice long string */
    char string[256];

    printf( "Please enter a long string: " );

    /* notice stdin being passed in */
    fgets ( string, 256, stdin );

    printf( "You entered a very long string, %s",
string );

    getchar();
}
```

Manipulating Strings

- **Header: <string.h>**
- strcat – concatenate two strings
- strchr – string scanning operation
- strcmp – compare two strings
- strcpy – copy a string
- strlen – get string length
- strncat – concatenate one string with part of another
- strncmp – compare parts of two strings
- strncpy – copy part of a string
- strrchr – string scanning operation
- strlwr – Converts string to lowercase
- strupr – Converts string to uppercase
- strstr – Find a substring

- **String I/O:**
- Scanf():

```
char c[20];
scanf("%s", c);
```

the scanf() function takes only a single string before the white space

- Gets():

```
char name[30];
gets(name);
```

the gets() function i used to read a line of text.

- Puts()

```
puts(name);
```

Function to display string.

String Manipulation Function Examples

- strcat() function:

```
char name[50];
char lastname[50];
char fullname[100];
strcat( fullname, name );     /* Copy name into full name */
strcat( fullname, " " );      /* Separate the names by a space */
strcat( fullname, lastname ); /* Copy lastname onto the end of fullname */
```

- Note: use strncat() or strlcat() instead of strcat, in order to avoid buffer overflow.

- strlen() funtion

```
char str[20] = "Hello";
printf("Length of string str1: %d", strlen(str1));
```

- strcmp function

```
if (strcmp(s1, s2) ==0)
        printf("string 1 and string 2 are equal");
```

- strcpy function

copies the string str2 into string str1

```
strcpy(s1,s2);
```

- strchr function

searches string str for character

```
strchr(mystr, 'f');
```

- strstr function

searches string str for string

```
strchr(mystr, 'and');
```

Pointers

- A variable that stores the address of another variable.
- Syntax:

data-type *variable_name;

- Example:

int *pointer;

- Pointer Initialization:

int c=22;

int *pc;

pc=&c;

or,

int *pc = &c ;   //initialization and declaration together

- Common mistakes:

int c, *pc;

// Wrong! pc is address whereas, c is not an address.

pc = c;

// Wrong! *pc is the value pointed by address whereas, %amp;c is an address.

*pc = &c;

// Correct! pc is an address and, %amp;pc is also an address.

pc = &c;

// Correct! *pc is the value pointed by address and, c is also a value.

*pc = c;

Benefit of using pointers

- Pointers are more efficient in handling Array and Structure.
- Pointer allows references to function and thereby helps in passing of function as arguments to other function.
- It reduces length and the program execution time.
- It allows C to support dynamic memory management.

Pointer and Arrays

- We can declare a pointer of type int to point to the array arr.
- arr is equal to &arr[0]

```
int *p;
p = arr;
or p = &arr[0];  //both the statements are equivalent.
```

```
int i;
int a[5] = {1, 2, 3, 4, 5};
int *p = a;  // same as int*p = &a[0]
for (i=0; i<5; i++)
{
 printf("%d", *p);
 p++;
}
```

- Pointer variables of char type are treated as string.

```
char *str = "Hello";
or
char *str;
str = "hello";   //thi is Legal
```

- Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3]={
                "Adam",
                "chris",
                "Deniel"
         };
//Now see same array without using pointer
char name[3][20]= {
                  "Adam",
                  "chris",
                  "Deniel"
           };
```

Functions

- Two types of functions
- Standard library functions

standard library functions are built-in functions in C programming to handle tasks

- User defined functions

functions created by the user are called user-defined functions.

- Structure of function

```
datatype functionName(datatype arguements);
//funtion prototype
int main()
{
...
   functionName();  // function call
...
}
datatype functionName(datatype arguements)
             //function definition
{
   block of statements; //function body
}
```

Types of User Defined Function

Function with no arguments and no return value

```
#include<stdio.h>
void area();  // Prototype Declaration
void main()
{
area();
}

void area()
{
   float area_circle;
   float rad;
   printf("\nEnter the radius : ");
   scanf("%f",&rad);
   area_circle = 3.14 * rad * rad ;
   printf("Area of Circle = %f",area_circle);
}
```

Function with no arguments and a return value

```
#include <stdio.h>
int getInteger(); //function prototype
int main()
{
   int n;
n = getInteger();
      printf("Number entered by user is %d.", n);
   return 0;
}
int getInteger()
{
   int n;
   printf("Enter a positive integer: ");
   scanf("%d",&n);
   return n;
}
```

Types of User Defined Function Cont’d

Function with arguments but no return value

```
#include <stdio.h>
void swap(int,int);
void main()
{
 int a,b;
 printf("Enter 2 numbers: ");
 scanf("%d %d",&a,&b);
 swap(a,b);
}
void swap(int a,int b)
{
  int temp;
  temp=a;
  a=b;
  b=temp;
  printf("Numbers after swapping are : %d, %d",a,b);
}
```

Function with arguments and a return value.

```
#include<stdio.h>
float calculate_area(int);
int main()
{
   int radius;
   float area;
   printf("\nEnter the radius of the circle : ");
   scanf("%d",&radius);
   area = calculate_area(radius);
   printf("\nArea of Circle : %f ",area);
   return(0);
}
float calculate_area(int radius)
{
   float areaOfCircle;
   areaOfCircle = 3.14 * radius * radius;
   return(areaOfCircle);
}
```

Function Call

Call by Value

```
int calc(int x);
int main()
{
 int x = 10;
 x = calc(x);
 printf("%d", x);
}

int calc(int x)
{
 x = x + 10 ;
 return x;
}
```

Call by Reference

```
void calc(int *p);
int main()
{
 int x = 10;
 calc(&x);     // passing address of x as
argument
 printf("%d", x);
}

void calc(int *p)
{
 *p = *p + 10;
}
```

Structures

- User defined data type that can hold data items of different kind.
- used to represent a record
- syntax:

```
struct struct_name {
  type member1;
  type member2;
  /* declare as many members as desired. */
};
```

- structure variable syntax:

```
struct  struct_name  var_name;
```

- assign values to struct members

  struct struct_name var_name = {value for memeber1, value for memeber2 …so on for all the members}

or

```
var_name.memeber_name = value;
```

```
struct database {
  int id_number;
  int age;
  float salary;
};

int main()
{
  struct database employee;

   employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

Structures as Function Arguments

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

/* function declaration */
void printBook( struct Books book );

int main( ) {

   struct Books Book1={"C Programming",
"Livewire", "C Programming Tutorial",
6495407};
/* print Book1 info */
   printBook( Book1 );
}
```

```
void printBook( struct Books book ) {

   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);
}
```

Pointers to Structers

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
/* function declaration */
void printBook( struct Books *book );
int main( ) {

  struct Books Book1={"C Programming",
"Livewire", "C Programming Tutorial",
6495407};
/* print Book1 info */
   printBook(&Book1 );
}
return 0;
}
```

```
void printBook( struct Books *book ) {

   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);
}
```

Unions in C

- unions are similar to structure
- difference in memory allocation between union and structure.
- The amount of memory required to store a structure variable is the sum of memory size of all members.
- the memory required to store a union variable is the memory required for the largest element of an union.

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

int main()
{
 union car car1, car2, *car3;
 return 0;
}
```

Aliasing using Typedef

- used to create an alias name for another data type
- Syntax:

typedef datatype alias

- examples:

```
typedef int km_per_hour ;
typedef int points ;

km_per_hour current_speed ;
points high_score ;
```

- used to simplify the declaration of a compound type (struct, union) or pointer type

```
typedef struct MyStruct {
    int data1;
    char data2;
} newtype;
```

Or

```
typedef struct {
    int data1;
    char data2;
} newtype;

newtype a;
```

Using typedef with pointers

- example:

```
typedef int *intptr;   // type name: intptr
                       // new type: int*

intptr ptr;            // same as: int *ptr
```

- with structure pointer:

```
typedef struct Node* NodePtr;
...
NodePtr startptr, endptr, curptr, prevptr, errptr, refptr;
```

Enumerated datatype

- a data type consisting of a set of named values

- Syntax

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

- Example of Enumeration in C

```
enum week {sun, mon, tue, wed, thu, fri, sat};
enum week today;
```

Enumerated data type cont'd

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

enum week {sun, mon, tue, wed, thu,
fri, sat};
void main()
{
 enum week today;
 today=tue;
 printf("%d day",today+1);
 getch();
}
```

```
enum cardsuit {
   Clubs,
   Diamonds,
   Hearts,
   Spades
};

struct card {
   enum cardsuit suit;
   short int value;
} hand[13];
```