WEEK 5
FUNCTIONS

Dr. Yuzana Win

OUTLINES

- What is a Function?
- Define Function proto-types and Invoke Functions
- Methods of Passing Data among Functions
- Usage of auto, static, and extern storage qualifiers
- Recursion

WHAT IS A FUNCTION?

- A *function* is a sub-unit of a program which performs a specific task.
- A function can take any number of arguments mixed in any way.
- A function can return at most one argument.
- We can declare variables within a function just like we can within `main()` - these variables will be deleted when we return from the function

NEED FOR FUNCTIONS

- As programs become more complex and large, several problems arise. Most common are :

- Algorithms for solving more complex problems become more difficult and hence difficult to design

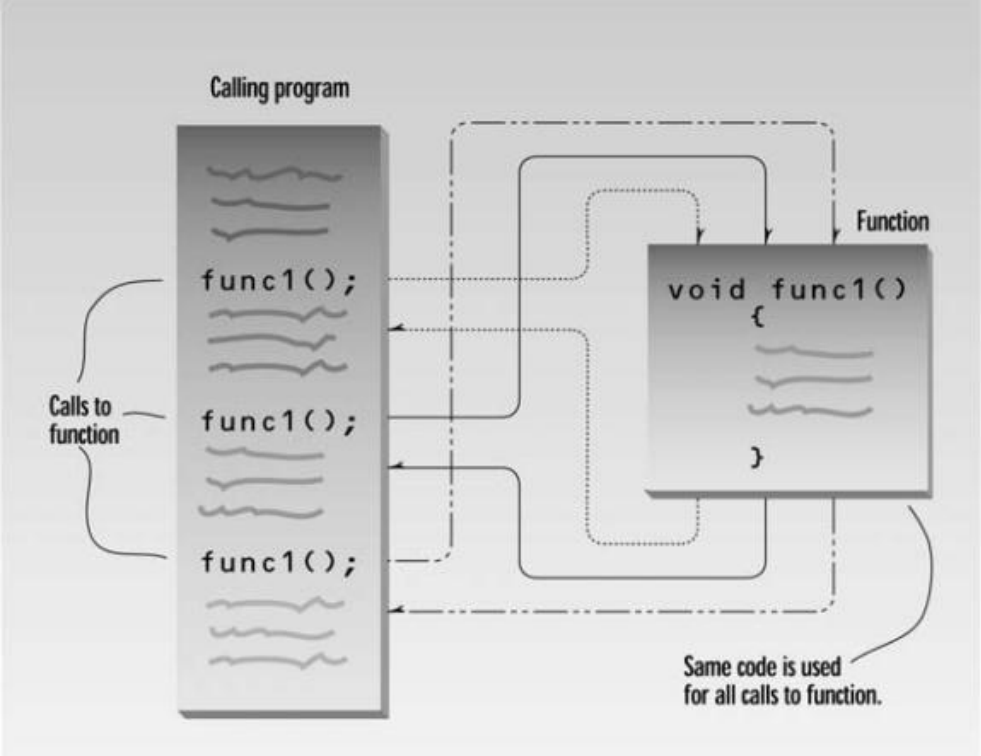
- Even if algorithms are known, its implementation becomes more difficult because the size of the program is longer

- As programs become larger and complex, debugging and testing becomes more difficult.

- As programs become larger and complex, more documentation is required to make the program understandable for people who will use and maintain the program

WHY USE FUNCTIONS?

- Avoids rewriting the same code over and over
- Easier to write programs and keep track of what they are doing



STRUCTURE OF FUNCTION

The general form of a function is as follows:

```
return-type  functionname(datatype arg1, datatype arg2,...)
        {
            local variables declaration;
            executable statement 1;
            executable statement 2;
                        :
             return(expression);
         }
```

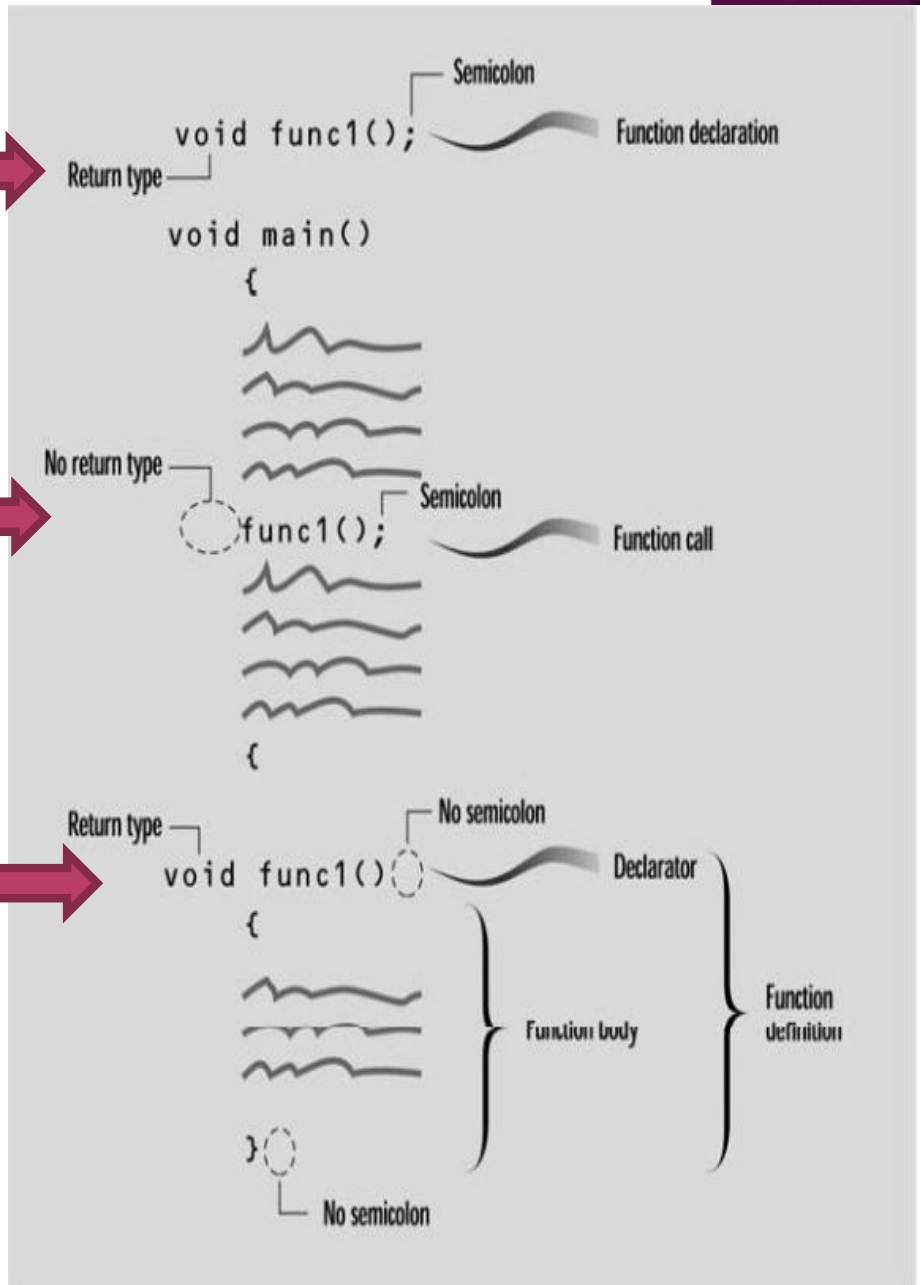
```
int getmax(int x, int y)
 { return x>y?x:y; }

int getsum(int x, int y)
{  return (x+y); }

int power(int x, int y)
{
     int p=1,i;
     for(i=0;i<y;i++)
     {  p=p*x;  }
     return p;
}
```

SIMPLE FUNCTION

(1) Function *Declaration*

(2) *Calling* a Function

(3) The Function *Definition*



FUNCTION PROTOTYPING

- Function prototype is a function declaration that specifies the return type and data types of the arguments.
- A function prototype/declaration informing the compiler the number and data types of arguments to be passed to the function, and the data type of the value returned by the called function.
- Prototype for the function find_big

```
int  find_big(int, int);
```

- Prototype for the swap function written

```
void  swap(int*, int*);
```

- Usually prototypes appear at the beginning of a Program.

INVOKING A FUNCTION

- The user-defined functions are accessed from a function simply by its name and actual arguments / parameters enclosed with in parentheses.
- These actual arguments are used to pass values of formal arguments defined in the function.
- When the function call is encountered, the control is transferred to the called function.
- The formal arguments are copied by actual arguments and the execution of the function is carried out.
- When the return statement is executed or last statement has finished its execution, the control is transferred back to the place of function call in the calling function.

FUNCTION DEFINITIONS

- Defining a Function

```
return_type function_name(parameter-list)
{
        body of the function
}
```

```
int getsum(int x, int y)
    {
        return (x +y);
    }
```

Return type: A function may return a value.
A data type of the result (default **int**)
void - function returns nothing

Function Name: The actual name of the function

Parameters: The value is referred to as actual parameter or argument

Function body: The function body contains a collection of statements that define what the function does

LOCAL AND GLOBAL VARIABLES

- The variables declared inside a function are local to that function. It can be accessed only with in that function.
- Memories for the local variables are allocated only when the function is invoked and de-allocated when the control returns to the calling function.
- Local variables are also known to be automatic variables.
- The variables declared outside of all function are global variables. These global variables are visible to all functions.

LOCAL AND GLOBAL VARIABLES

Example : Usage of Global Variables

```
#include <iostream>
using namespace std;

int  a=0;            /* Global variable */
float  b;            /* Global variable */
void fun();

int main()
{
     int a=10;                  /* Local variable*/
     float b=1.2;
     fun();
     cout << a;
     return 0;
}
void fun()
{
    a+=10;
}
```

What will be the output?

LOCAL AND GLOBAL VARIABLES

```
#include <iostream>
using namespace std;

int a, b;                          /*Global variables*/
void fun(int a);

int main()
{          a = 1; b = 2;          /*Local variable*/
           fun(a);
           cout << a << b << endl;
           return 0;
}
void  fun(int a)
{          a = 3;                  /*local variable*/
           {    int b = 4;         /*local variable*/
                cout << a << b << endl;
           }
           cout << a << b << endl;
           b = 5;
}
```

Output

```
3 4
3 2
1 5
```

```
#include <iostream>
using namespace std;
main()
{
      cout << "\nI am in main";
      italy();
      cout << "\nI am finally back in main”

}
italy()
{
       cout << "\nI am in italy";
       brazil();
       cout << "\nI am back in italy";
}
brazil()
{
        cout << "\nI am in brazil";
        argentina();
}
argentina()
{
        cout << "\nI am in argentina";
}
```

Outline

- Call function
  - Any function can be called from other functions

Program Output

```
I am in main
I am in italy
I am in brazil
I am in argentina
I am back in italy
I am finally back in main
```

```
#include <iostream>
using namespace std;

main()
{
      message();
      message();

}
message()
{
       cout << "\nJewel Thief!";
}
```

Outline

- Call function
- A function can be called any number of times

Program Output

```
Jewel Thief!
Jewel Thief!
```

PASSING VALUES BETWEEN FUNCTIONS

```
// Finding the sum of three integers
#include <iostream>
using namespace std;
main()
{
      int a, b, c, sum;
      cout << "\nEnter any three integers:";
      cin >> a >> b >> c;
      sum = calsum(a,b,c);  // calling function
      cout << "\nSum = " << sum;
}
// Function definition

calsum(int x, int y, int z)
{
      int d;
      d = x + y + z;
      return(d);
}
```

Outline

- Input values
- Call function
- Function definition

- Program Output

```
Enter any three numbers: 10 20 30
Sum = 60
```

```
// Finding the biggest of two integers

#include <iostream>
using namespace std;
int find_big(int,int);      // find_big function
prototype
int main( )
{     int  num1, num2, big;
      cout << “Enter two integers:\n” ;
      cin >> num1 >> num2;
      big=find_big(num1,num2);   // find_big
function call
      cout << " The biggest is : " << big ;
      system("pause");
      return 0;
 }
int  find_big(int  a, int b)
 {        if ( a > b) return a;
          else return b;
 }
```

```
Enter two integers: 22 85
The biggest is: 85
```

Outline

- Function declaration (two parameters)
- Input values
- Call function
- Function definition

PASSING VALUES BETWEEN FUNCTIONS

- The mechanism used to pass data to a function is via argument list. There are two approaches to passing arguments to a function. These are
  - **Call by Value**
  - **Call by Reference**

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.
- When an array is used as a function argument, its address is passed to a function. This is an exception to the call-by-value parameter passing convention.

CALL BY VALUE

```
#include <iostream>
using namespace std;
void swapv(int x, int y);
main()
{
      int a=10, b=20;
      swapv(a,b);
      cout << "a= " << a << " " << "b= " << b << endl;
}
// Function definition
void swapv(int x, int y)
{
          int t;
          t = x;
          x = y;
          y = t;
          cout << "x= " << x << " " << "y= " << y << endl;
}
```

```
x = 20 y = 10
a = 10 b = 20
```

Outline

- Values of a and b remain unchanged even after exchanging the values of x and y

CALL BY REFERENCE

Call by Reference

- In this approach, the addresses of actual arguments are used in the function call. The formal arguments should be declared as pointer variables.

- This approach is of practical importance while passing arrays and structures among functions and also for passing back more than one value to the calling function.

CALL BY REFERENCE

```
#include <iostream>
using namespace std;
void swapr(int *x, int *y);
main()
{
      int a=10, b=20;
      swapr(&a,&b);
      cout << "a= " << a << " " << "b= " << b << endl;
}
// Function definition
void swapr(int *x, int *y)
{
          int t;
          t = *
          *x = *y;
          *y = t;
          cout << "x= " << *x << " " << "y= " << *y << endl;
}
```

```
x = 20 y = 10
a = 20 b = 10
```

Outline

- To exchange the values of a and b using their addresses stored in x and y

CALL BY REFERENCE

```
#include <iostream>
using namespace std;
float areaperi(int r, float *a, float *p);
int main()
{
    int radius;
    float area, perimeter;

    cout << "Enter radius of a circle\n";
    cin >> radius;

    area, perimeter = areaperi(radius, &area,
&perimeter);

    cout << "Area=" << area << endl;
    cout << "Perimeter= " << perimeter << endl;
    return 0;
}
```

```
float areaperi(int r,
float *a, float *p)
{
    *a = 3.14*r*r;
    *p = 2 * 3.14 * r;
    return(*a, *p);
}
```

```
Enter radius of a
circle 5
Area = 78.5
Perimeter = 31.4
```

PASSING A DISTANCE STRUCTURE

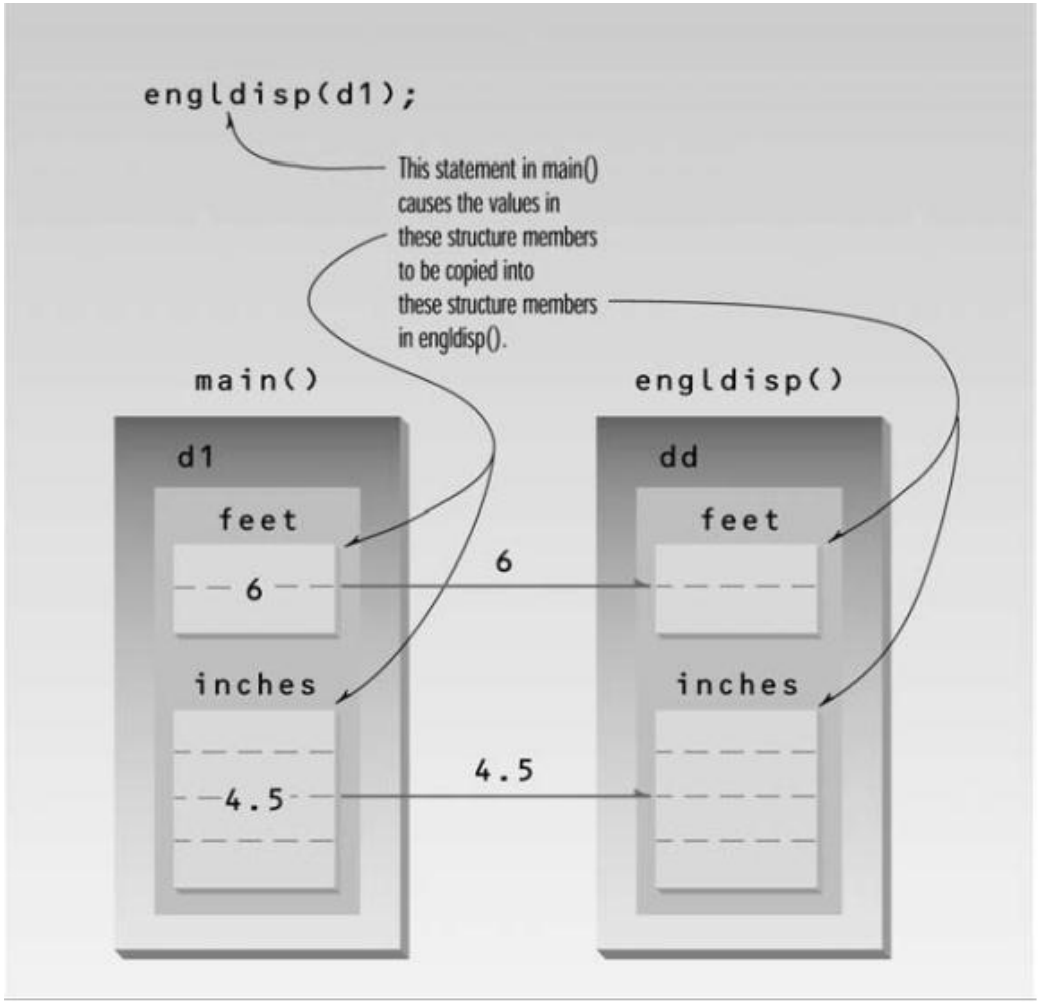
```
// engldisp.cpp
// demonstrates passing structure as argument
#include <iostream>
using namespace std;

struct Distance      //English distance
{
int feet;
float inches;
};
```

```
void engldisp( Distance dd )
{
cout << dd.feet << "\'-" << dd.inches
<< "\"";
}
```

```
void engldisp( Distance );
int main()
{
Distance d1, d2;
cout << "Enter feet: ";
cin >> d1.feet;
cout << "Enter inches: ";
cin >> d1.inches;

cout << "\nEnter feet: ";
cin >> d2.feet;
cout << "Enter inches: ";
cin >> d2.inches;
cout << "\nd1 = ";
engldisp(d1);
cout << "\nd2 = ";
engldisp(d2);
cout << endl;
return 0;
}
```

OUTPUT

Enter feet: 6
Enter inches: 4.5
Enter feet: 5
Enter inches: 4.25
d1 = 6’-4.5”
d2 = 5’-4.25”



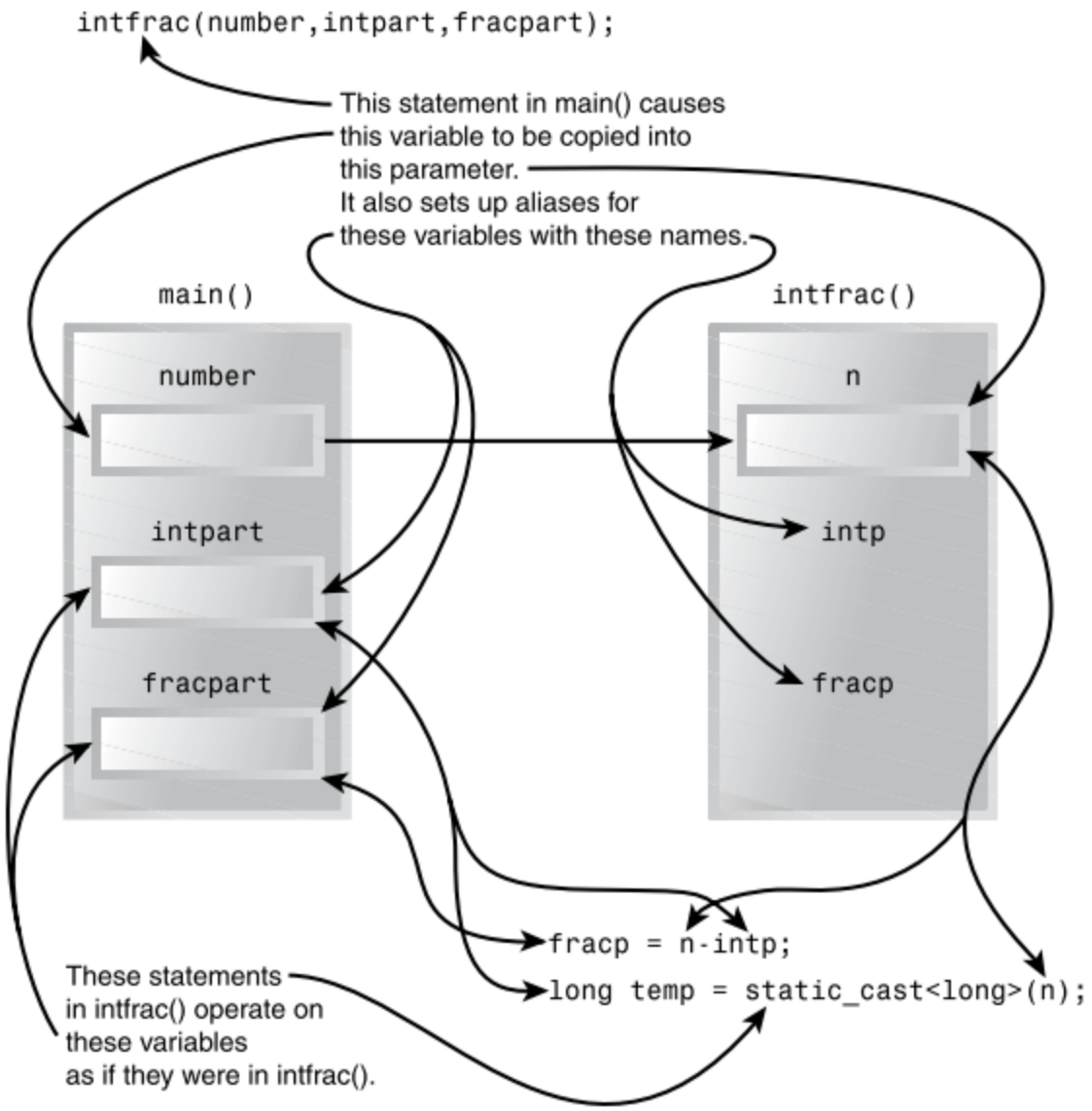
PASSING SIMPLE DATA TYPES BY REFERENCE

```
// ref.cpp
// demonstrates passing by reference
#include <iostream>
using namespace std;
int main()
{
void intfrac(float, float&, float&);
float number, intpart, fracpart;

do {
   cout << "\nEnter a real number:";
   cin >> number;
   intfrac(number, intpart, fracpart);
   cout << "Integer part is" << intpart  <<
", fraction part is" << fracpart << endl;
} while( number != 0.0 );
   return 0;
}
```

```
void intfrac(float n, float& intp,
float& fracp)
{
   long temp = static_cast<long>(n);
   intp = static_cast<float>(temp);
   fracp = n - intp;
}
```

OUTPUT

Enter a real number:
99.44
Integer part is 99,
fractional part is 0.44



STORAGE CLASSES

- The storage class precedes the variable's declaration and tells the compiler how the variable should be stored in memory. C++ supports following four storage classes:
    1. Auto
    2. Static
    3. Extern
    4. Register

- If the storage class is omitted at declaration, it is assumed of type auto storage class.

AUTO STORAGE CLASS

- Auto storage class
- Stored in Memory

- If not initialized in the declaration statement, their initial value is unpredictable value often called garbage value

- Local (visible) to the block in which the variable is declared. If the variable is declared with in a function, then it is only visible to that function.

- It retains its value till the control remains in that block. As the execution of the block/function is completed, it is cleared and its memory destroyed.

AUTO STORAGE CLASS

Example: Program illustrating auto variables

```
#include <iostream>
using namespace std;

void fun();
int main()
{
    fun();   //will output as 21
    fun();   // will output as 21
    fun();   // will output as 21
    return 0;
}
void fun()
{
    auto int  b = 20;
    /*Auto (Local) variable to fun() */
    b++;
    cout << " in Fun :  \n" << b << endl;
}
```

STATIC STORAGE CLASS

Static Storage Class

- Stored in Memory
- If not initialized in the declaration, it is initialized to zero.
- Local(Visible) to the block in which the variable is declared. If the variable is declared inside a function, then it is visible to that function. But, if the variable is declared outside the function, then it is visible to all functions following its declaration.
- It retains its value between different function calls. That is, when for the first a function is called, a static variable is created with initial value zero, and in subsequent calls it retains its present value.

STATIC STORAGE CLASS

Example: Program illustrating static variables

```
#include <iostream>
using namespace std;
void fun();
int main()
{
    fun();   //will output as 21
    fun();   // will output as 22
    fun();   // will output as 23
     return 0;
}
void fun()
{
    static int  b = 20;
    /*Auto (Local) variable to fun() */
    b++;
    cout << " in Fun :  \n" << b << endl;
}
```

REGISTER STORAGE CLASS

- Stored in Processor registers, if register is available. If no register is available, the variable is stored in memory, and works as if its storage class is auto.
- If not initialized in the declaration, the variable is initialized to zero.
- It retains its value till the control remains in that block. As the execution of the block/function is completed, it is cleared and its memory destroyed.

REGISTER STORAGE CLASS

- Operations on a **register** variable could occur much faster than on a normal variable
- Can only apply the **register** specifier to local variables and to the formal parameters

```
int power(register int m, register int e)
{
register int temp;
temp = 1;
for(; e; e--) temp = temp * m;
return temp;
}
```

EXTERN STORAGE CLASS

- The Scope of external variables is global.
- External variables are declared outside all functions, usually in the beginning of the program file.
- As all the function will be defined after these variables declaration, these variables are visible to all functions of the program.
- This storage class is useful for a multi files program.
- Stored in memory
- If not initialized in the declaration, it is initialized to zero.

EXTERN STORAGE CLASS

Example

Program File a.cpp

```
int val;                                        /*Global */
int main( )
 {
      cin >> val;
      compute();
      cout << val;
      return 0;
  }
```

Program File b.cpp

```
void  compute()
{
     extern int val;
     /*This implies that, the variable val is defined in another file*/
     val++;
 }
```

INLINE FUNCTION

- To cause a function to be expanded in line rather than called, precede its definition with the **inline keyword.**
- Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code.
- For this reason, it is best to **inline only very small functions.**
- Further, it is also a good idea to inline only those functions that will have significant impact on the performance of your program.

RECURSION

- If a function is having a self-reference, it is recursion.
- In order that the function should not continue indefinitely, a recursive function must have the following properties:

  - There must be certain criteria, called base criteria, for which a function does not call itself.

  - Each time, when a function calls itself, it must be close to the base criteria. That is, it uses an argument(s) smaller than the one it was given at previous reference.

RECURSION

Direct recursion: Function F contains a statement that invokes itself.

Indirect recursion: Function F invokes some other function, which invokes another function and so on until function F is again invoked.

RECURSION

Example Illustrative examples of Recursive function

Factorial Function : n! = n * (n-1)!

Base Criteria is 0! =1

```
int  factorial( int n)
{
      if(n = = 0)                          // base criteria
      return 1;
      else
      return( n* factorial(n-1));  // recursive call
}
```

ASSIGNMENTS

1. Write a program that is implemented with sub functions and provides menu based option like, Menu:
   - Addition
   - Subtraction
   - Multiplication
   - Division
   - Exit
2. Write a program to generate Fibonacci number with the recursive function call.
3. Find the sum of 1!/1^1+2!/2^2+3!/3^3+….n!/n^n

ASSIGNMENTS

4. Write a program to demonstrate a function called *lbstokg*() that returns a weight in kilograms after being given a weight in pounds. The program should get input value from the user, call *lbstokg*(), and print out the result.

5. Construct a program to create a function called *reversit()* that reverses a character-string (an array of char). The program should get a string from the user, call *reversit()*, and print out the result. For example, the phrase “I saw Elba” reverse to “Able was I”.