Chapter 9 "User-Defined Functions"

Need for User-Defined Functions

- C is a modular programming language.
- The program may become too large and complex and as a result the task of debugging, testing and maintaining becomes difficult.
- If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. These independently coded programs **subprograms** that are much easier to understand, debug, and test. In C, such subprograms are referred to as **“Functions”**.
- There are times when certain types of operations or calculations are repeated at many points throughout a program.

ADVANTAGES OF USING THE FUNCTION

- Separating the programs into functions increase the **ease of maintenance** and **enhancement** of the program.
- It will make the program **more understandable** for the user.
- It will **avoid the duplication of code** and **errors**.
- **Debugging** of the program will be **easier**.
- 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 function are addressed later.
- A function may be used by many other programs.

Types of functions

- **There are basically two types of functions.**
  1. Library Functions
  2. User Defined Functions

1) Library Functions: To perform some very commonly required task, ready-made functions are also available with C compiler. These functions are known as *Library function*.

- The library functions are declared in different files which are known as *header files*.
- **Header file: math.h**

  sin(x), cos(x), pow(x,y), sqrt(x) etc.

2) User Defined Functions: Functions defined by the user are called user defined functions.

Elements of User-Defined Functions: (1) Function definition

- It is an independent program module that is specially written to implement the requirements of the function.
- A function definition, also known as **function implementation** shall include the following **elements**:
  1. Function name
  2. Function type
  3. List of parameters
  4. Local variable declarations
  5. Function statements
  6. A return statement
- All the six elements are grouped into two parts, namely,
  - **Function header** (first three elements- 1,2 & 3)
  - **Function body** (Last three elements- 4,5 & 6)

Elements of User-Defined Functions:
(1) Function definition

- **Syntax:-**

```
Return_Type Function_Name(Arguments or Parameter list)
    {
    local variable declaration;
    executable statment1;
    executable statment2;
    ................
    ................
    return statement;
    }
```

Elements of User-Defined Functions:
(1) Function definition

- Example,

```
int add ( int x,  int y )     // definition of the function
{
int sum;
sum = x + y;             // x & y will be taken as input
                           from the calling func.
return(sum);             // returns integer value
}
```

Elements of User-Defined Functions: (2) Function Call

- In order to use this function we need to invoke it at a required place in the program. This is known as the function call.
- **The program (or a function) that calls the function is referred to as the *calling program or calling function*.**
  - The statement which activates the functions with the required input values (called arguments or *actual arguments*) is called calling of the function.
  - Ex - **answer = add (5,3);**
  - Here the function **add** is called in the main program with the **input values 5 and 3**. As defined earlier, add function add this two values and returns the **sum which will be assigned to answer**.

Elements of User-Defined Functions: (3) Function Declaration

- The calling program should declare any function (like declaration of a variable) that is to be used later in the program. This is known as the *function declaration or function prototype*.
- It consists of four parts:
  - Function type (return type)
  - Function name
  - Parameter list
  - Terminating semicolon
- Syntax:

  Return_type Function_Name (Arguments);
- Ex - **int add (int, int);**

Elements of User-Defined Functions: (3) Function Declaration

- **A prototype declaration** may be placed in **two places** in a program:
  1. Above all the functions (including main) in the global declaration section, this prototype is referred to as a ***global prototype***. Such declarations are available for all the functions in the program.
  2. Inside a function definition (in the local declaration section), the prototype is called a ***local prototype***. Such declarations are primarily used by the functions containing them.

Write a C program that perform addition of two numbers entered by the user.

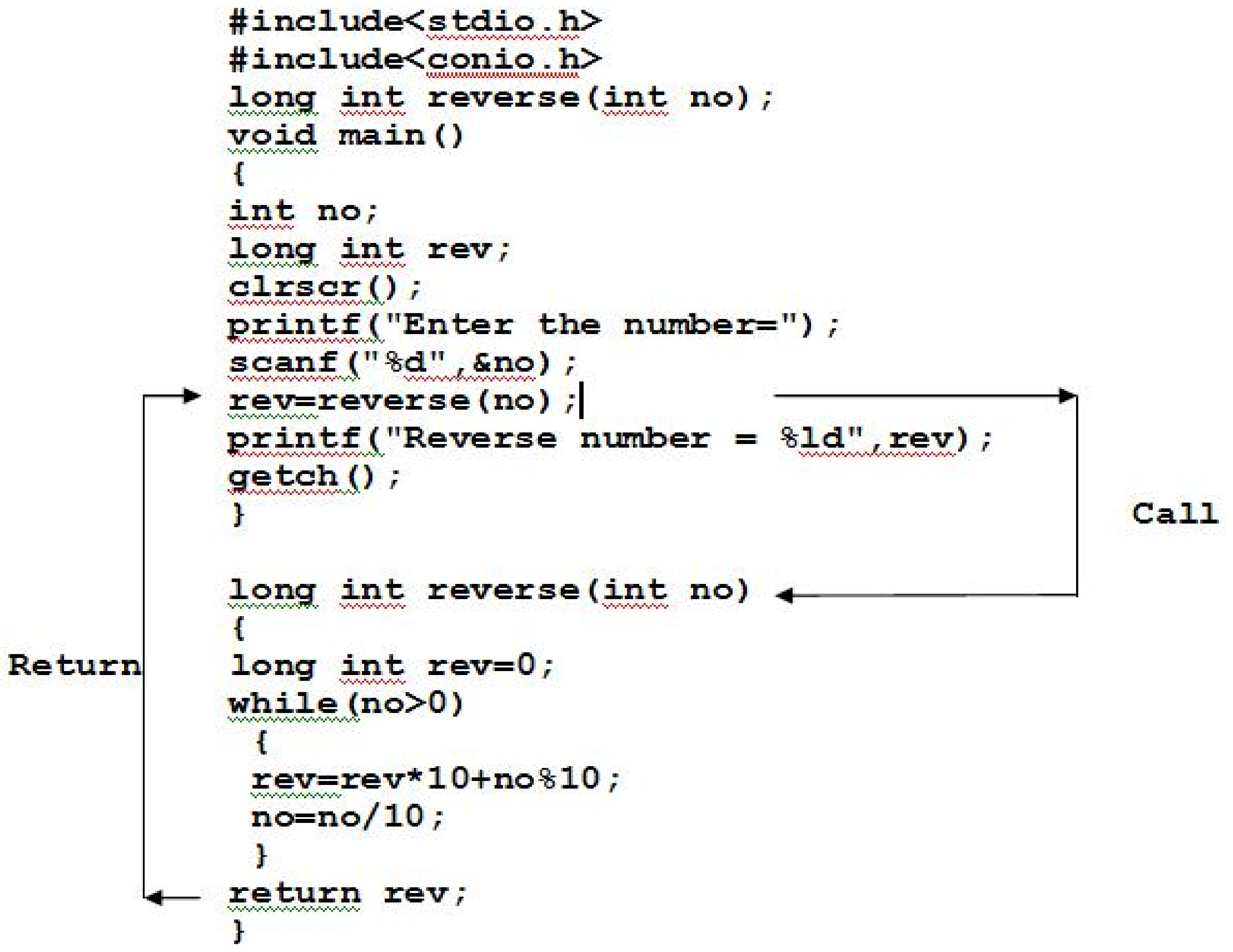
```
#include <stdio.h>

int main()
{
int a,b, answer;        /* Variable declaration */
int  sum(int,int);   //Function declaration
printf("Give the first integer number : \n");
scanf("%d",&a);
printf("Enter second integer number : \n ");
scanf("%d",&b);

answer=sum(a,b); /* Call to the
function sum */

printf("The sum is %d",answer);
return 0;
}
```

```
/* Function Definition  */
int sum(int p, int q)
{
int sum;
sum = p+q;
return(sum);
}
```

Output

```
Give the first integer number : 4
Enter second integer number : 8
The sum is 12
```

Write a C program to find reverse number using UDF.



```
#include<stdio.h>
#include<conio.h>
long int reverse(int no);
void main()
{
int no;
long int rev;
clrscr();
printf("Enter the number=");
scanf("%d",&no);
rev=reverse(no);
printf("Reverse number = %ld",rev);
getch();
}

long int reverse(int no)
{
long int rev=0;
while(no>0)
  {
  rev=rev*10+no%10;
  no=no/10;
  }
return rev;
}
```

Category of Functions:

- A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:
  - **Category 1:** Functions with no arguments and no return values.
  - **Category 2:** Functions with arguments and no return values.
  - **Category 3:** Functions with arguments and one return value.
  - **Category 4:** Functions with no arguments but return a value.
  - **Category 5:** Functions that return multiple values.

Category 1:
Functions with no arguments and no return values.

- When a function has **no arguments**, it does **not receive any data from the calling function**.
- Similarly, when **it does not return a value**, the **calling function does not receive any data from the called function**.
- In effect, there is **no data transfer** between the calling function and the called function.

Category 1:
Functions with no arguments and no return values.

For example, Write a C program to find addition of two integer numbers using UDF.

```
#include <stdio.h>

void add(void);

int main()
{
add();
return 0;
}
```

```
void add(void)
{
int a,b,sum;
printf("Enter the value
of a & b.");
scanf("%d%d",&a,&b);
sum = a+b;
printf("Addition of a and
b is %d",sum);
}
```

Category 2:
Functions with arguments and no return values

- Here called function receives the data from calling function.
- Similarly, when it does not return a value, the calling function does not receive any data from the called function.
- In effect, there is one way data transfer between the calling function and the called function.

Category 2:
Functions with arguments and no return values

For example, Write a C program to find addition of two integer numbers using UDF.

```
#include <stdio.h>

void add(int a,int b);

int main()
{
int a,b,sum;
printf("Enter a & b.");
scanf("%d%d",&a,&b);
add(a,b); // Argument are
called actual arguments
return 0;
}
```

```
void add(int a,int b)
//Argument calls formal
arguments
{
int sum;
sum = a+b;
printf("Addition of a and b
is %d",sum);
}
```

Category 3:
Functions with arguments and one return value

- Here called function receives the data from calling function.
- Similarly, when it does return a value, the calling function does receive any data from the called function.
- In effect, there is two way data transfer between the calling function and the called function.

Category 3:
Functions with arguments and one return value

For example, Write a C program to find addition of two integer numbers using UDF.

```
#include <stdio.h>

int add(int a,int b);
int main()
{
int a,b,sum;
printf("Enter the value of a
& b.");
scanf("%d%d",&a,&b);
sum=add(a,b);
printf("Addition of a and b is
%d",sum);
return 0;
}
```

```
int add(int a,int b)
{
int sum;
sum = a+b;
return sum;
//return(a+b);
}
```

Category 4: Functions with no arguments but return a value

- When a function has no arguments, it does not receive any data from the calling function.
- Similarly, when it does return a value, the calling function does receive any data from the called function.

Category 4:
Functions with no arguments but return a value.

For example, Write a C program to find addition of two integer numbers using UDF.

```
#include <stdio.h>
int add(void);
int main()
{
int sum;
sum=add();
printf("Addition of a and b is
%d",sum);
return 0;
}
```

```
int add(void)
{
int a,b,sum;
printf("Enter the value of
a & b.");
scanf("%d%d",&a,&b);
sum = a+b;
return sum;
//return(a+b);
}
```

Category 5:
Functions that return multiple values

- The return statement can return only one value.
- Suppose, however, that we want to get more information from a function. We can achieve this in C using the arguments not only to receive information but also to send back information to the calling function. The arguments that are used to "send-out" information are called ***output parameters***.
- The mechanism of sending back information through arguments is achieved using what are known as the **address operator (&)** and **indirection operator (*)**.

Parameter passing methods

- In all the function prototype and definition examples we have seen thus far, parameters are specified as if they were simple variables, and the process of parameter passing is comparable to the process of assigning a value to a variable:
  - Each parameter is assigned the value of its corresponding argument
  - Although the value of a parameter may change during the course of a function, the value of the corresponding argument is not affected by the change to the parameter

Passing by value

- The process described on the previous slide, and illustrated in all examples thus far, is called ***passing by value*** - this is the default method for parameter passing
- When arguments are passed by value:
  - a ***copy*** of each argument ***value*** is passed to its respective parameter
  - the parameter is stored in a separate memory location from the storage location of the argument, if the argument has one
  - Any valid expression can be used as an argument

EXAMPLE

The program below illustrates what happens when arguments are passed by value. A tracing of the changes in the program's variables is shown on the right.

```
int multiply (int, int);

int main()
{
        int a, b, c;
        a = 2;
        b = 5;
        c = multiply(a,b);
        a = multiply(b,c);
        return 0;
}

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

a	b	c	x	y
2	5	10	2	5
50			10	10
			5	
			50	

When the program ends, the variables remaining in memory have the values shown in red

Limitations of pass by value

- Recall that a function can have either one return value or no return value
- If we want a function's action to affect more than one variable in the calling function, we can't achieve this goal using return value alone – remember, our options are one or none
- The next example illustrates this problem

Example – swap function

Suppose we want to write a function that swaps two values: that is, value a is replaced by value b, and value b is replaced by the original value of a. The function below is an attempt to achieve this goal.

```
void swap (int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}
```

The function appears to work correctly. The next step is to write a program that calls the function so that we can test it:

```
int main()
{
        int a=2, b=6;
        printf("Before swap, a=%d and b=%d",a,b);
        swap(a,b);
        cout << "After swap, a=%d and b=%d",a,b);
        return 0;
}
```

Output:

```
Before swap, a=2 and b=6
After swap, a=2 and b=6
```

What went wrong?

- In the swap function, parameters x and y were passed the values of variables a and b via the function call swap(a, b);
- Then the values of x and y were swapped
- When the function returned, x and y were no longer in memory, and a and b retained their original values
- Remember, when you pass by value, the parameter only gets a copy of the corresponding argument; changes to the copy don’t change the original

Building a better swap function: introducing reference parameters

- C offers an alternative parameter-passing method called ***pass-by-reference***
- When we pass by reference, the data being passed is the ***address*** of the argument, not the argument itself
- The parameter, rather than being a separate variable, is a reference to the same memory that holds the argument – so any change to the parameter is also a change to the argument

Revised swap function

We indicate the intention to pass by reference by appending an ampersand (&) to the data type of each reference parameter. The improved swap function illustrates this:

```
void swap (int& x, int& y)
{
        int tmp = x;
        x = y;
        y = tmp;
}
```

The reference designation (&) means that x and y are not variables, but are instead references to the memory addresses passed to them

If we had the same main program as before, the function call:

```
swap(a,b);
```

indicates that the first parameter, x, is a reference to a, and the second parameter, y, is a reference to b

How pass-by-reference works

- In the example on the previous slide, x and y referenced the same memory that a and b referenced
- Remember that variable declaration does two things:
  - Allocates memory (one or more bytes of RAM, each of which has a numeric address)
  - Provides an identifier to reference the memory (which we use instead of the address)
- Reference parameters are simply additional labels that we temporarily apply to the same memory that was allocated with the original declaration statement
- Note that this means that arguments passed to reference parameters must be variables or named constants; in other words, the argument must have its own address

Example

Earlier, we looked at a trace of the program below, on the left. The program on the right involves the same function, this time converted to a void function with an extra reference parameter.

```
int multiply (int, int);
int main()
{
        int a, b, c;
        a = 2;
        b = 5;
        c = multiply(a,b);
        a = multiply(b,c);
        return 0;
}

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

```
void multiply (int, int, int&);
int main()
{
        int a, b, c;
        a = 2;
        b = 5;
        multiply(a,b,c);
        multiply(b,c,a);
        return 0;
}

void multiply (int x, int y, int& z)
{
        z = x * y;
}
```

Nesting of functions

- C permits nesting of functions freely.
- main() can call function1, which calls function2, which calls function 3, which can call function 4, … … … and so on.
- There is in principle no limit on as to how deeply functions can be nested.

Nesting of functions

For example, Write a C program to find delta. [Hint. Delta = b2 – 4*a*c]

```
#include<stdio.h>

int delta(int,int,int);
int square(int);

void main()
{
int a,b,c,d;
printf("Enter the value of a,b
& c");
scanf("%d%d%d",&a,&b,&c);
d=delta(b,a,c);
printf("\nD=%d",d);
return 0;
}
```

```
int delta(int y,int x,int z)
{
int d;
d=square(y)-4*x*z;
return d;
}

int square(int b)
{
return b*b;
}
```

Example: Fing factorial of a number

Write a C program to find factorial of a given number [Hint: Factorial of 3 = 3 * 2 * 1 = 6]

```
#include <stdio.h>

int main()
{
  int c, n, fact = 1;
  printf("Enter a number to
calculate it's factorial\n");
  scanf("%d", &n);

for (c = 1; c <= n; c++)
    fact = fact * c;

  printf("Factorial of %d =
%d\n", n, fact);
  return 0;
}
```

Output:

```
Enter a number to
calculate it's factorial
5
Factorial of 5 = 120
```

Recursion

- When a called function in turn calls another function a process of 'chaining' occurs. **Recursion is a special case of this process, where a function calls itself.**
- A very simple example of recursion is presented below:

```
main()
{
printf("This is an example of recursion.\n");
main();
}
```

Recursion

Write a C program to find factorial of a given number using concept of recursion. [Hint: Factorial of 3 = 3 * 2 * 1 = 6]

```
#include<stdio.h>
long int factorial(int no);
void main()
{
int no;
long int fact;
printf("Enter the number=");
scanf("%d",&no);
fact=factorial(no);
printf("Factorial of number=
%ld",fact);
return 0;
}
```

```
long int factorial(int no)
{
long int fact;
if(no==1)
    return 1;
else
    fact=no*factorial(no-1);
return fact;
}
```

Factorial with Recursion - Explained

$$\begin{tikzcd}
5! \arrow[d] & & & & & & 120 & & & \\
5 * 4! \arrow[dr] & & & & & & 5 * 24 \arrow[u, bend right] & & & \\
 & 4 * 3! \arrow[dr] & & & & & & 4 * 6 \arrow[ul, bend right] & & \\
 & & 3 * 2! \arrow[dr] & & & & & & 3 * 2 \arrow[ul, bend right] & \\
 & & & 2 * 1! \arrow[dr] & & & & & & 2 * 1 \arrow[ul, bend right] \\
 & & & & 1 & & & & &
\end{tikzcd}$$

Passing arrays to functions: One-Dimensional arrays:

- Like the values of simple variables, it is also possible to pass the values of an array to a function.
- To pass a one-dimensional an array to a called function, it is sufficient to list the name of the array, <u>without any subscripts</u>, and the size of the array as arguments.
- For example,

```
    int a[10],n=10;
    largest(a,n);
```

  will pass the whole array a to the called function.
- Otherwise we have to pass the first element address to called function.
- Like,

```
        largest(&a[0],n);
```

Passing arrays to functions: One-Dimensional arrays:

- The called function expecting this call must be appropriately defined. The function header might look this:

```
int largest(int a[], int size)
```

- **Three rules to pass an array to a function:**
  1. The function must be called by passing only the name of the array.
  2. In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.
  3. The function prototype must show that the argument is an array.

Write a C program to find out largest number in array using function.

```
#include<stdio.h>

int largest(int [],int);

void main()
{
int i,j,large;
int a[10],n;
printf("Enter the elements of the
array=\n");
for(i=0;i<10;i++)
{   printf("a[%d]=",i+1);
    scanf("%d",&a[i]);
}
large=largest(a,10);
printf("Largest = %d",large);
return 0;
}
```

```
int largest(int a[],int n)
{
int large,i;
large=a[0];
for(i=0;i<n;i++)
 {
 if(large<a[i])
          large=a[i];
 }
return large;
}
```

Passing strings to functions

- The strings are treated as character arrays in C and therefore the rules for passing strings to functions are very similar to those for passing arrays to functions.
- **Basic rules:**

1. The string to be passed must be declared as a formal argument of the argument of the function when it is defined.

   Example,

```
            void display(char item_name[])
            {
            … … …
            … … …
   }
```

Passing strings to functions

- **Basic rules:**

2. The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

```
void display(char str[]);
```

3. A call to the function must have a string array name without subscripts as its actual argument.

```
display(names);
```

where names is a properly declared string array in the calling function.

- We must note here that, like arrays, strings in C cannot be passed by value to functions.

The scope, visibility and lifetime of variables

- In C not only do all variables have a data type, they also have a *storage class*. The following variable **storage classes** are most relevant to functions.
    1. Automatic variables
    2. External variables
    3. Static variables
    4. Register variables

The scope, visibility and lifetime of variables

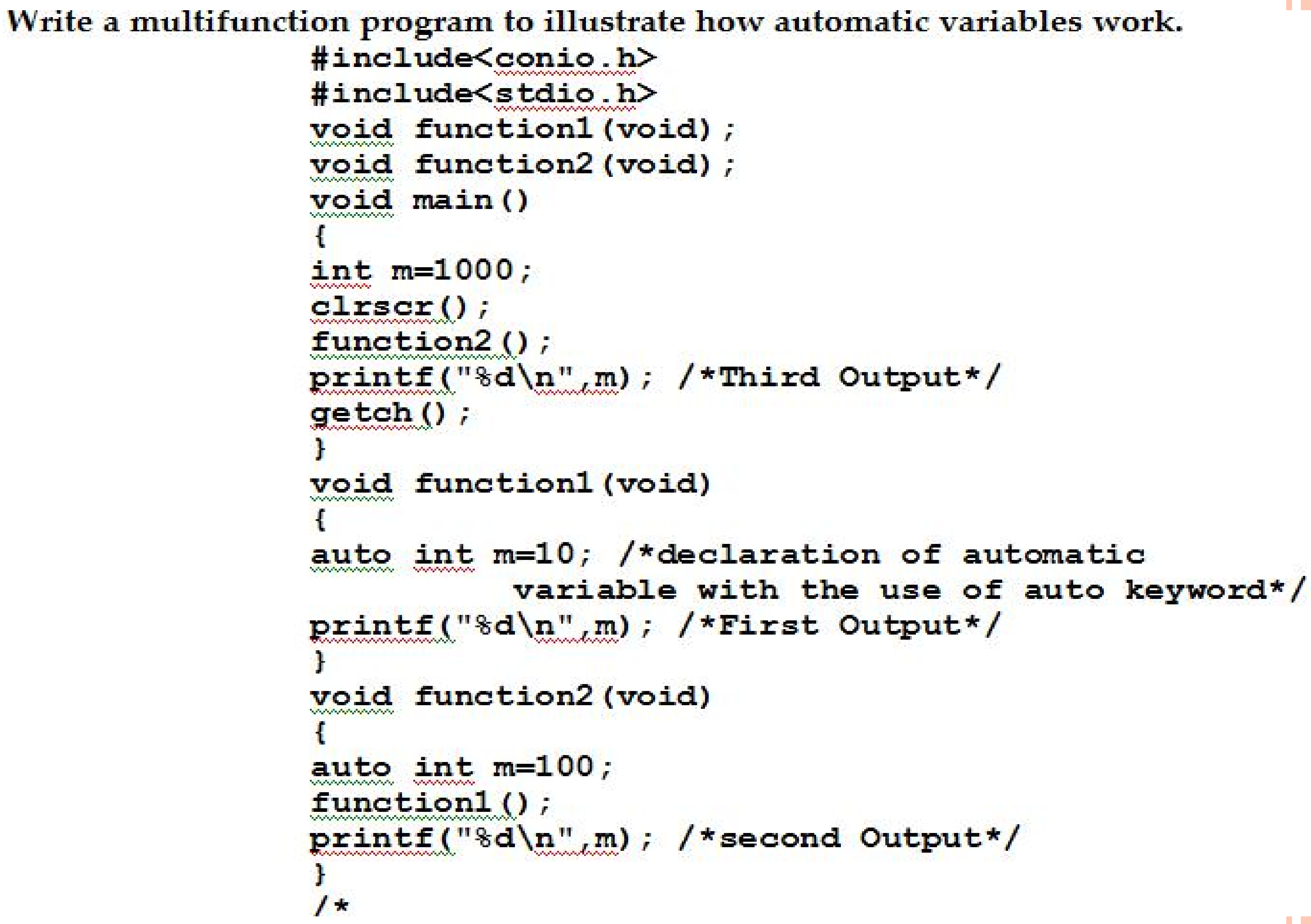
- The **scope of variable** determines over what region of the program a variable is actually available for use ('active').
- The **Longevity** refers to the period during which a variable retains a given value during execution of a program ('alive').
- The **visibility** refers to the accessibility of a variable from the memory.

- The variables may also be broadly categorized, depending on the place of their declaration, as **internal (local)** or **external (global)**.
- **Internal variables** are those which are declared within a particular function.
- **External variables** are declared outside of any function.

The scope, visibility and lifetime of variables:
1. Automatic variables

- Automatic variables are inside a function in which they are to be utilized. They are **created** when the **function is called** and **destroyed automatically** when the **function is exited**, hence the name automatic.
- Automatic variable is also known as **local** variable or **internal** variable.

The scope, visibility and lifetime of variables:
1. Automatic variables



Write a multifunction program to illustrate how automatic variables work.

```
#include<conio.h>
#include<stdio.h>
void function1(void);
void function2(void);
void main()
{
int m=1000;
clrscr();
function2();
printf("%d\n",m); /*Third Output*/
getch();
}
void function1(void)
{
auto int m=10; /*declaration of automatic
          variable with the use of auto keyword*/
printf("%d\n",m); /*First Output*/
}
void function2(void)
{
auto int m=100;
function1();
printf("%d\n",m); /*second Output*/
}
/*
```

The scope, visibility and lifetime of variables: 2. External variables

- Variables that are both **alive** and **active** throughout the entire program are known as external variables.
- They are also known as **global** variables.
- In a case local variable and a global variable have the same name, the local variable will have precedence over the global variable one in the function where it is declared.
- **global variables are initialized to zero by default.**

Write a multifunction program to illustrate how global variables work.

```
#include<conio.h>
#include<stdio.h>
int function1(void);
int function2(void);
int function3(void);
int a; /*global or external
variables*/
void main()
{
clrscr();
a=50; /*global a*/
printf("a=%d\n",a);
printf("a=%d\n",function1());
printf("a=%d\n",function2());
printf("a=%d\n",function3());
getch();
}
```

```
function1(void)
{
a=a-10;
}
int function2(void)
{
int a;    /*automatic or local
variable*/
a=10;
return(a);
}
function3()
{
a=a+20; /*global a*/
}
```

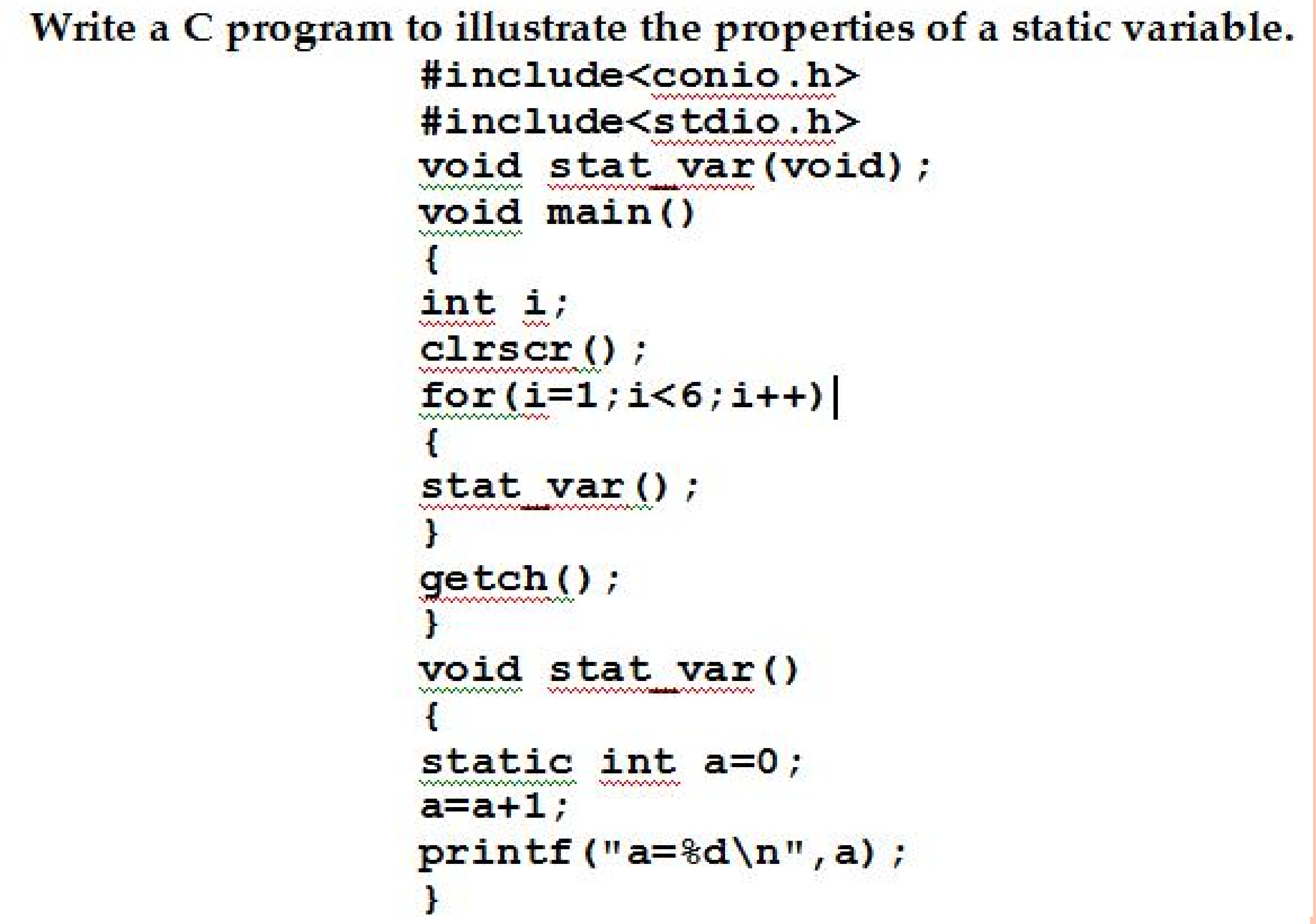
The scope, visibility and lifetime of variables: 3. Static variables

- As the name suggests, the value of static variables persists until the end of the program.
- A variable can be declared static using the keyword **static** like

```
static int x;
static float y;
```

- **A static variable is initialized only once, when the program is complied. It is never initialized again.**

The scope, visibility and lifetime of variables:
3. Static variables



Write a C program to illustrate the properties of a static variable.

```
#include<conio.h>
#include<stdio.h>
void stat_var(void);
void main()
{
int i;
clrscr();
for(i=1;i<6;i++)
{
stat_var();
}
getch();
}
void stat_var()
{
static int a=0;
a=a+1;
printf("a=%d\n",a);
}
```

The scope, visibility and lifetime of variables: 4. Register variables

- We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory.
- Since a register **access is much faster** than a memory access, keeping the frequently accessed variables (e.g., loop control variables) in the register will lead to faster execution of programs. This is done as follows:

```
register int x;
```

Thank You