Discipline Courses-I
Semester-I
Paper: Programming Fundamentals
Unit-III
Lesson: Functions
Lesson Developer: Arpita Aggarwal
College/Department: PGDAV College, University of Delhi

Table of Contents

- 1.4: Functions - IV
  - 1.4.1: Learning objectives
  - 1.4.2: Scope of variables
    - 1.4.2.1: Local variables
    - 1.4.2.2: Global variables
  - 1.4.3: Overloaded Functions
  - 1.4.4: Friend functions and virtual functions

- 1.5: Functions - V
  - 1.5.1: Learning objectives
  - 1.5.2: Recursion - I
    - 1.5.2.1: Advantage of recursion over iteration?
    - 1.5.2.2: Characteristics of recursion
  - 1.5.3: Recursive Factorial function
  - 1.5.4: Recursive Fibonacci function

- 1.6: Functions - VI
  - 1.6.1: Learning objectives
  - 1.6.2: Recursion - II
  - 1.6.3: Recursive "GCD" function
  - 1.6.4: Ackermann
    - 1.6.4.1: Implementation using the formula directly
    - 1.6.4.2: Implementation using partially iterative method
    - 1.6.4.3: Implementation using the pre-computed values in formula

- Summary
- Exercises
- Glossary
- References

1.1 Functions - I

Welcome to the world of functions! Till now you must have written a number of programs solving problems from the world around you. Now you will learn to break big programs into smaller, manageable units called functions which can be called from anywhere and any number of times in the program.

1.1.1 Learning Objectives

After reading this chapter you should be able to:

1. Think of a problem in terms of smaller modules.
2. Write programs that call functions.
3. Write functions solving small problems.
4. Distinguish between function prototype and function definition.
5. Write function that return a value or receives parameters from calling program.

1.1.2 Introduction to functions

Imagine you are asked to write a program which does multiple tasks and whose code runs into 500 or more lines. Most probably you will find it difficult to write and correct (debug) such a big program. Functions come to your help in such situations. They are used to organize big or complex programs into smaller, easily manageable and independent units. You can think of a function as a user-defined operation which is represented by a name. Each function encapsulates a group of statements performing a particular task.

Function: It is a named group of program statements performing a particular task which can be invoked multiple times from different places in a program.

When a function is invoked from a program, control is transferred to the first statement inside the function body. Remaining lines of the function body are then executed and the control is then returned back to the next line of the calling program from where the function was invoked.

1.1.2.1 Why use Functions?

- You find it easy to write a correct small function.
- It is easy to read, write and debug a function.
- It is easier to maintain and modify a function as they tend to be self documenting and highly readable (by virtue of being small).
- Functions can be called any number of times, at any place, with different parameters (described later on).
- Another reason for using functions is to reduce the size of the program. Any sequence of instructions that is repeated in a program can be grouped together to

form a function. The function code is stored in only one place in the memory even though the function may be executed a repeated number of times.

Value addition: Did you know?
Heading text: Other names for 'Functions'
Body text: Some languages use the terms 'Subroutines' or 'procedures' in place of functions. 'Procedural programming' term is derived from the word 'procedures' and means a programming paradigm where procedures/functions are used to break down a big problem into smaller, easily manageable units.
Source:

1.1.3 User-Defined Functions

While writing your own functions, you should carefully think about what your function will do, and how it will interface with the rest of your program. Choose name of the function so as to describe the task performed by it and as far as possible, it should not modify any global data. The round brackets after function name are mandatory.

The **general form** of a function is:

```
datatype function_name (argument list)
{
        ……
        //Body of function
        ……
}
```

Datatype: You will often need to return a value to the calling program. Datatype is the type of the value which is being returned to the calling program. In case the function does not return any value, the datatype is void.

Function_name: It is the name by which the calling program will invoke/call the function. A function can be called any number of times by any program.

Argument list (optional): There is often some data which is required to be passed from calling program to the function. This data is passed to the called function through this argument list.

Example:

```
// Program to demonstrate a function

#include<iostream>

using namespace std;
```

```
// function declaration
void demo();

//function definition
void demo()
{
        cout << "I am inside function demo() ……endl"; // body of function
}

// main() Function
int main()
{
        cout << " I am inside main() …." <<endl;
        demo();
        cout << " I am inside main() …..  again…." <<endl;
        return 0;
}
```

Value addition: Common Mistake…
Heading text: Beware!
Body text: Beginners often put ~~semicolons after function definition~~ a semicolon after the function name in function definition because of which function ends there and execution control never enters the body. Moreover, no compiler error comes for this.
Source:

1.1.4 Example of a function

Example of a function: demo() is the name of the function which is invoked by the main() program. The body of the function demo is written outside the main() program.

```
//Program to demonstrate a function

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

```
// function declaration
void demo();

//function definition
void demo()
{
        cout << "Hello friend! Welcome to the world of functions";

}

// main() program
int main()
{
        demo(); //function is invoked
        return 0;
}
```

1.1.5 Passing values to the function

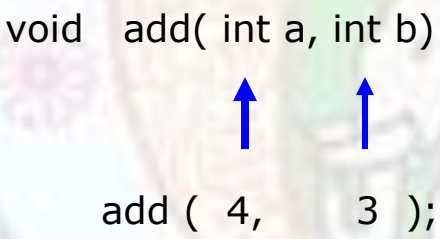
When you need to pass some values from the calling program to the called function, you do it with the argument list or parameter list. It is a comma separated list of variables (as many as needed) where each argument /parameter consists of a datatype followed by an identifier and act as regular local variable inside the function body. The parameters which are passed from the calling program (i.e. main() in example given below) are called actual parameters. On the other hand they are received in formal parameters inside the function body (i.e. add() in this case).

Example:

```
// Example of passing values to function
#include <iostream>
using namespace std;

// function declaration
void add (int, int);

//function definition
void add (int a, int b)          // function to add two values
```

```
{
        int r;
        r=a+b;
        cout << r;
}

// main() Function
int main ()
{
        add(4,3);
        return 0;
}
```

Now let us examine the above code. Execution always starts from the main() function. Right after that we see a function called add with two arguments i.e. 4 and 3 which are passed to the function add. These are our actual parameters. The function receives these two arguments (4 and 3) in two variables i.e. int a and int b respectively. They are the formal parameters. If you pay attention, you will see similarity between the structure of function call and the function declaration.



The arguments passed and the receiving parameters have a clear correspondence. Their number, datatype and sequence should always match. The values of both the arguments (4, 3) passed from the calling program are copied to the local variables int a and int b within the called function.

Note: Calling program can pass any number of parameters to the called function.

Value addition: Programming tip
Heading text: Actual and formal parameters
Body text: ○ Always keep the number, datatype and sequence of the actual and formal parameters same. ○ Some authors treat arguments and parameters separately, arguments are passed i.e. they are actual parameters, while parameters receive them i.e. they are formal parameters.

○ Parameters can also be passed as expressions such as 4+6, 2.5+1.5 etc
Source:

1.1.6 Returning value from the function

Usually the programs call functions to do some manipulation or calculation which need to be returned to the calling program. The datatype of the value to be returned by the function is specified before the function name in the declaration. After calculating, this value is returned using the return statement as the final statement of the function.

Look at the following code:

```
// Example of a function to return a value

#include <iostream>
using namespace std;

// function declaration
int add (int, int);

//function definition
int add (int a, int b)
{
        int r;
        r=a+b;
        return (r);     // returns the value of r to calling program
}
// main() Function
int main()
{
        int z;
        z = add (4, 3);
        cout << "The result is " << z;       // prints value returned by function add()
                                             //and stored in z
        return 0;

}
```

Notice that it is the same program as the earlier one with minor modifications: instead of printing the result i.e. value of variable r in the function add(), it returns the value of r to the main() program where it is received in variable z and then its value is printed.

```
int add (int a, int b)
```


```
z   =   add (4, 3);
```

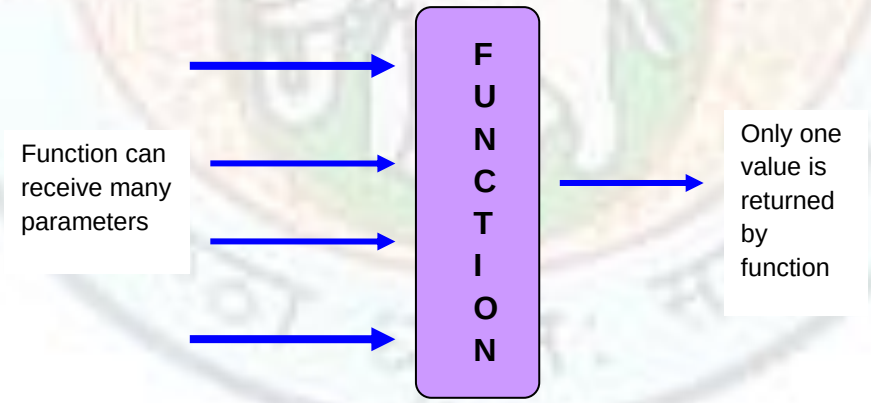
The following line of code is in main() program:

```
cout << "The result is " << z;
```

This code prints the result of the addition of 4 and 3 on screen.

Note: We can return only one value from the function to the calling program.

Value addition: Common Misconceptions
Heading text: Number of parameters passed to and from functions
Body text:



Notice only one value can be returned from function while any number of parameters can be passed from calling program to the function.

Source:

1.1.7 Function Prototyping

So far, you have seen a lot of activity on functions. Each function used in the sample programs has a declaration and a definition. The declaration, called a prototype, is the function's interface to the rest of the code in a program. It specifies the details about function such as number and type of arguments being received by the function and the datatype of value being returned by it.

In other words, function prototype is a declaration statement having the following form:

datatype function_name (argument_list);

where,

datatype: is the datatype of value returned by function.
function_name: is the name by which this function is called.
argument_list: it contains the datatypes and names of arguments that must be passed to the function.

1.1.7.1 For Creating Function Prototype...

You create a function prototype by deciding what types of data the function needs from calling program. The argument/parameter list for the function is based on this data. Then you decide on a descriptive name for the function and create a prototype for the function.

For Example:

Consider a function to multiply two integer numbers. These two numbers are sent as the arguments to the function and their product is returned back to the calling program.

You can name the function as multiplyNumbers(). The preliminary prototype for this function might look like this:

multiplyNumbers(int a, int b);

The product of the two integer numbers is going to be a long integer number. Therefore, the return type of the function is long. With this decision made, the completed function prototype looks like this:

long multiplyNumbers(int a, int b);

Value addition: Things to Note!
Heading Text: Programming tips
Body text: Please Note: • Each argument variable must be declared independently inside the paraentheses. long multiplyNumbers(int a, b); is illegal.

- In function prototype, names of arguments are dummy variables and are optional. Therefore,
  long multiplyNumbers(int, int);
  is acceptable.
- Function prototype should be encountered by compiler before the call for function.

Source:

1.1.7.2 Defining a Function

After specifying the function prototype, we need to implement the function – that is create the function definition. For this, we first write the skeleton for the function definition, using the prototype as a model. For the above function prototype, the function definition looks like this:

```
long multiplyNumbers( int a, int b)
{
        …….
        // write code for function
        ……
}
```

Value addition: Did you know?

Heading text: Programming tip

Body text: Function definition is automatically a function declaration. C++ makes prototyping essential i.e. its prototype must appear before a function's use. If the called function definition appears before its calling function's definition then the function's prototype may be skipped.

Source:

1.1.8 The main() function

Function main() is the starting point for the execution of any C++ program. It returns a value of type int to the operating system. The prototypes of main() are:

```
int main();

int main (int argc, char *argv[]);
```

Since the function main() returns a value, it should have the return statement for termination. Therefore, the definition of main() should look like:

```
// look of the main() function
int main()
{
        ……….
        ……..
        return 0;
}
```

Since in C++ return type of functions is int by default, the keyword int in the main() function is optional. But most C++ compilers will generate an error or warning if there is no return statement.

Value addition: You should know….
Heading text: return value of main()
Body text: Many operating systems test the return value (also called exit value) of program to determine if there is any problem. Normally an exit value of zero means the program ran successfully, while a nonzero value means there was a problem.
Source:

1.1.9 Example of a function returning value

Now we give the complete example of a program with a function prototype, function call and the function definition.

```
//function to multiply two numbers
#include<iostream>
using namespace std;
// function prototype
int multiply(int, int);
//main function that calls the function multiply
int main()
{
   int a, b, prod;
   cout << "Enter first number\n";
   cin >> a;
   cout << "Enter second number\n";
```

```
    cin >> b;
// Function call
    prod = multiply(a, b);
    cout <<"Product of" << a << "and" <<b << "is" << prod;
    fflush(stdin);
    getchar();
}
// Function Definition
int multiply(int x, int y)
{
    int mul;
    mul = x * y;
    return(mul);
}
```

Output of the program:

```
Enter first number
4
Enter second number
5
Product of 4 and 5 is 20
```

How do we get this output? If you notice, the function main() takes in two integer values, a and b, as inputs and sends these numbers as parameters to the function multiply(). Inside the function multiply(), they are received in integers x and y respectively. Here they are multiplied and the result is stored in the variable mul which is returned back to the function main().

1.2 Functions - II

Having had some exposure to functions, you are now ready for knowing more about different types of functions and their arguments. This section introduces some of such functions and arguments. C++ has a very rich library of functions that are built-in and can be readily used. You will be able to use such functions, created by others and having standard interface, in your programs.

1.2.1 Learning Objectives

After reading this section you should be able to:

1. Write Inline functions.
2. Take decision what functions should be written as inline.
3. Pass default arguments to the functions.
4. Define some function arguments as constant.
5. Use some built-in library functions in your program.

1.2.2 Inline Functions

Recalling from the previous sections, when you call a function from a program, a number of things happen. First, the caller function stores the information about the current instruction being executed, then locate and call the function, execution enters the function, function does its work, return value is sent back to caller, execution leaves the function and returns to the caller to execute next instruction.

All this takes time!

Consider the following min() function to return minimum of two values:

```
int min(int a, int b)
{
        return( a < b ? a : b );  // return a if smaller else return b
}
```

You will find many advantages of defining a function for such a small operation:

- One generally finds it easier to read a call for min() and interpret its meaning than to understand the code with conditional operator, especially if complex numbers are used.
- In case you need to change anything in code, you will find it easier to do so in one implementation instead of multiple occurrences in the program.
- Using a function, you are ensuring that each test in the function is implemented in the same manner i.e. semantics remain uniform.
- You can reuse the function for other applications instead of rewriting it.

So many benefits!!!!! But beware; there is one serious disadvantage too!!!!!

Calling the function is slower than directly evaluating the conditional operator in your program.

This is because two arguments must be copied, machine registers must be saved, program must branch to a new location etc.

So what is the solution?

C++ provides us with another solution…… **inline functions**.

Inline Functions: It is a function that is expanded in line when it is called. That is, the compiler replaces the function call with the corresponding function code. This way they avoid the runtime overheads involved with functions.

Inline functions are defined as follows:

```
inline datatype function_name(argument list)
{
        ……
        //function body
        …….
}
```

For example:

```
inline double cubeNumber(double x)
{
        return( x * x * x);
}
```

This inline function can be invoked by statements like:

```
a = cubeNumber (5.0);
b = cubeNumber (2.5+3.5);
```

The results produced are:

```
a = 125
b = 216
```

1.2.2.1 Example of an Inline Functions

Let us have a look at a complete program having an inline function.

```
// program that calls an inline function

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

```
int main()
{
   int a,b,z;

   //Function prototype
   int larger(int, int);

   cout << "Enter first number\n";
   cin >> a;
   cout << "\nEnter second number\n";
   cin >> b;

   //Function call
   z = larger(a, b);

   cout << " \nLarger number is  " << z;

   fflush(stdin);
   getchar();
}

inline int larger(int p, int q)
{
  return (p > q) ? p : q;
}
```

In the above program, main function calls a function larger that has been defined as inline. Function larger takes two parameters p and q as arguments from main() and returns the larger of the two to it. Actually, when the function is called, it gets expanded at the place of the function call by having the code of the function replacing the function call.

Output of the program:

```
Enter first number
4
Enter second number
3
Larger number is  4
```

What is actually happening here??? When the function larger() is called by the function main(), it gets replaced with the code of the function definition. The larger of the two values is evaluated using tertiary operator and the result is outputted on the screen.

Value addition: Did you know?
Heading text: Alternatives to inline functions
Body text: One more solution to the problem of functions taking lot of time is to use macro definitions, popularly known as macros. But the drawback of using macros is that since they are not really functions therefore they do not undergo usual error checking during compilation. Another problem is that if you are not careful while writing macros, results can be incorrect and go unchecked. For example: Look at the following macro: #define multiply(a, b) (a * b) If you use the macro as follows: X = multiply(2+1, 5); Macro is going to be substituted in a straightforward manner giving the result as 7 instead of 15. Of course this can be avoided by using parenthesis but no such situation arises using inline functions.
Source:

1.2.2.1 Are inline functions always expanded?

No!

Inline functions are not always expanded. It is the prerogative of the compiler to decide when to expand an inline. By writing an inline function you are simply recommending it to the compiler to expand it in line. And the compiler may choose to ignore this recommendation.

If the inline function definition is a long one or a complicated one, compiler can ignore the request for its expansion and compile it like a normal function.

Some situations where inline request can be ignored by the compiler:

- If a function is returning a values and also involves statements like loop, switch or goto.
- If the function contains some static variables.
- If there is a return statement in a function which does not return a value.
- If the function is recursive (you will learn later on).

Value addition: Common Mistake…
Heading text: Beware!
Body text: Inline functions make a program run faster because the overheads

involved in calling a function such as saving the current position of execution in caller program, locating the function, copying parameters, returning value etc. are eliminated. But inline functions make the program take up more memory because of the reproduction of function statements at all occurrences of the inline functions. Therefore, carefully trade-off between them!!!!
Source:

1.2.3 Default Arguments

Imagine you write a program calling a function with some arguments. And you forget to pass the values of the parameters!!! What would the function do now?

To deal with such situations, C++ provides us with the facility of providing default arguments in the functions.

A default argument is a value that is most appropriate argument value for a parameter in majority of the cases. If it is specified in function, one can call that function without specifying its arguments. In this case, function assigns the default value to the parameter which does not have a matching argument in the function call.

To specify default arguments, you simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. The format is:

Datatype function_name (datatype arg1 [= default_value], datatype arg2 [= default_value]...);

For Example:

int multiply (int a, int b = 5, int c = 2);

In this case, value of parameter a must be passed, passing value of b and c is optional. If they are passed, incoming values will be used otherwise the default values will be used.

For example, if you call the above function as follows:

x = multiply(3); //two arguments missing

Then, a = 3, b = 5, c = 2.

For a function call such as:

x = multiply (3, 4); //one argument missing

Here a = 3, b= 4 and c= 2.

Note: If you specify a default argument, all other arguments appearing after that must be default arguments. In other words, only trailing arguments can have default values.

An Example:

```
// default values in functions
#include <iostream>
using namespace std;
int divideNumber (int a, int b=2)
{
        int r;
        r=a/b;
        return (r);
}
int main ()
{
        cout << "Divide 20 by 2:-"<< divideNumber (20);
        cout << endl;
        cout << "Divide 40 by 4:-"<< divideNumber (40,4);
        return 0;
}
```

Here the **output** is:

```
Divide 20 by 2:- 10
Divide 40 by 4:- 10
```

Value addition: Common Mistake…
Heading text: Beware!
Body text: ○ Providing default arguments free the programmer from having to attend to every small detail of the function's interface. Moreover, they are useful if an argument is going to have the same value every time. For example, bank interest may remain same for all customers for a particular period of time. In this case default value of rate of interest can be used as default argument. ○ Default arguments are checked for type at the time of declaration and evaluated at the time of call.
Source:

1.2.4 const Arguments

When a function receives an argument, it can perform one of the following actions with regards to the value of the argument:

- modify the value itself by doing some manipulations
- **only** use the value of argument to modify another argument or another of its own variables i.e. the original value of argument remains intact.

If you know beforehand that the function is not supposed to modify the value of some particular argument, you should let the compiler know it. This is a good programming practice as it serves at least two purposes. First, the compiler will make sure that argument supplied stays intact. If the function tries to modify this argument, compiler would throw an error to let you know that an undesired operation is taking place. Second, this speeds up execution.

In such a case when it is required that the function should not change the value of an argument passed, you can pass that argument by making it constant with the use of keyword const prior to datatype of argument in the parameter list.

Syntax is:

datatype function_name ([datatype arg1,] const datatype arg2);

For example:

int sum (const int a, const int b);

Here parameters a and b are constant. By using qualifier const, you are telling the compiler that the function should not modify values of these parameters. The compiler will generate an error when this condition is violated.

An example:

Write a function that calculates and returns the perimeter of a rectangle if it receives the length and the width from the calling function, namely main(). Under no circumstances the values of length and width be modified inside function.

```
// Program with a function to find perimeter of a rectangle

#include <iostream>
using namespace std;
int Perimeter( const int len,  const int wid)
{
        long peri;
        peri = 2 * (len + wid);
        return peri;
}
int main()
{
        int length, width;
```

```
        cout << "Rectangle dimensions"<< endl;
        cout << "Enter the length: ";
        cin >> length;
        cout << endl;
        cout << "Enter the width: ";
        cin >> width;
        cout << endl;
        cout << "The perimeter of the rectangle is: "
                << Perimeter(length, width) << endl;

        fflush(stdin);      //clean input buffer
        getchar();
        return 0;
}
```

This program would produce the **output** as follows:

```
Rectangle dimensions
Enter the length: 5
Enter the width: 3

The perimeter of the rectangle is: 16
```

Note: The function Perimeter() does not change the values of the length or the width. Therefore, to reinforce the purpose of the assignment, you should make this clear to the compiler by adding reserved word const while receiving parameters.

Value addition: Programming tip
Heading text: Sequence of const parameters
Body text: You can make just one or more arguments constants, and there is no order on which arguments can be made constant.
Source:

1.2.5 Built-in Library Functions

After learning to create functions, as a smart programmer you will be creating any function to perform the desired job. To help you, C++ language provides a series of prewritten functions that can be used without caring about their internal implementation. All you need to know is what these functions do and what the interface to use them is.

These built-in functions are part of C++ language's huge library and are highly valuable, tested sufficiently and are completely reliable. They cover a wide range of applications also such as maths, strings, finance, input/output etc.

C++ provides standard library containing files that stores prewritten functions. These files are called header files and provide prototypes, definitions of library functions, declarations of datatypes and constants used in the library functions. A number of such headers are present in C++ containing numerous functions.

Broadly, we can divide the Standard C++ Library into following special-purpose libraries:

- The Language Support Library: It defines types and functions used by C++ programs employing features like operators such as new and delete, exception handling and runtime type information (RTTI). Examples of the headers under this category are <exception>, <limits>, <new>, <typeinfo>.
- The Diagnostics Library: This Library is used to detect and report error conditions in C++ programs. Example is <stdexcept>.
- The General Utilities Library: It contains components used by other Standard C++ Library components, such as the Containers, Iterators and Algorithms Libraries. Examples are <utility>, <functional>, <memory>.
- The Standard String Templates: This library provides facilities for manipulating characters. Example is <string>.
- Localization Classes and Templates: The cultural differences of its various users are addressed using this library. Example is <locale>.
- The Containers, Iterators and Algorithms Libraries (the Standard Template Library): This library manages and manipulates collection of objects. Examples are <algorithm>, <bitset>, <deque>, <iterator>, <list>, <map>, <queue>, <set>, <stack>, <unordered_map>, <unordered set>, <vector>.
- The Standard Numerics Library: Seminumerical operations are performed using this library. Examples are <complex>, <numeric>, <valarray>.
- The Standard Input/Output Library: Since the standard iostreams library may differ in different C++ products, compatibility can be achieved using this library. Examples are <fstream>, <iomanip>, <ios>, <iosfwd>, <iostream>, <istream>, <ostream>, <streambuf>, <sstream>.
- C++ Headers for the Standard C Library: Since the C International Standard specifies 18 headers which must be provided by a conforming hosted implementation, having the form *name*.h, the C++ Standard Library includes these 18 headers. Additionally, for each of them, it specifies a corresponding header that is functionally equivalent to its C library counterpart, but is located in std namespace. The name of each of these C++ headers is of the form c*name*, where *name* is the string obtained by removing ".h" extension from its equivalent C Standard Library header. For example, <stdlib.h> and <cstdlib> are both provided by the C++ Standard Library and are equivalent in function, with the exception that all declarations in <cstdlib> are located within the std namespace. Other examples are

<cassert>, <cctype>, <cerrno>, <cfloat>, <ciso646>, <climits>, <clocale>, <cmath>, <csetjmp>, <csignal>, <cstdarg>, <cstddef>, <cstdio>, <cstdlib>, <cstring>, <ctime>, <cwchar>, <cwctype>.

To give you some idea of the functions provided inside the header files, we list few of them here (list is not exhaustive in any way…):

Header	**Functions**
cmath	cos, sin, tan, acos, asin, atan,cosh, sinh, tanh, exp, frexp, log, ldexp, modf, pow, sqrt, ceil, fabs, fmod
cstring	strcpy, strncpy, memcpy, strcat, strncat, memcmp, strcmp, strcoll, strncmp, memchr, strchr, strstr, strchr, strpbrk, strlrn, memset
cstdlib	atof, atoi, atol, strtod, strtol, rand, srand, calloc, free, malloc, realloc, abort, exit, getenv, system, bsearch, abs, div, labs, ldiv
cassert	assert
cctype	isalnum, isalpha, iscntrl, isdigit, isgraph, islower, ispunct, isspace, isupper, isxdigit, tolower, toupper
cerrno	errno
csetjmp	longjmp, setjmp, jmp_buf
csignal	signal, raise
cstdarg	va_list, va_start, va_arg, va_end
cstdio	remove, rename, fclose, fflush, fopen, freopen, setbuf, setvbuf, fprintf, fscanf, printf, scanf, fgetc, fgets, fputc, fputs, getchar, putchar, fread, fwrite, fseek, feof
ctime	clock, difftime, mktime, time, asctime, ctime, gmtime, localtime, strftime

Now let's look at some examples that use these libraries and the functions contained in them. Since we all are most familiar with the iostream header, we'll start with an example using additional functions from this header:

For Example:

```
// modify precision

#include <iostream>
using namespace std;

int main ()
{

        double val = 3.65342;

        cout.precision(5);
```

```
        cout << "Value after precision is set to 5\n";
        cout << val << endl;

        cout.precision(10);
        cout << "Value after precision is set to 10\n";
        cout << val << endl;

        cout.setf(ios::fixed,ios::floatfield);          // floatfield set to fixed
        cout << "Value after floatfield set to fixed\n";
        cout << val << endl;

        fflush(stdin);
        getchar();
        return 0;
}
```

Output of the program is:

```
Value after precision is set to 5
3.6534
Value after precision is set to 10
3.65342
Value after floatfield set to fixed
3.6534200000
```

Now let us have another example where we use library cstring and the function strcpy contained in it:

```
// Example using header cstring and function strcpy

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

int main()
{
      char str1[80];
      char str2[80];

      str1[0]='\0';            //First string is NULL
      str2[0]='\0';            //Second string is NULL

      cout << "First String: " << str1 << endl;
      cout << "Second String: " << str2 << endl;

      cout << "Enter a string: ";
      cin.getline(str1,80);


      strcpy(str2,str1);
      cout << " After copying..." << endl;
```

```
        cout << "\nFirst String: " << str1 << endl;
        cout << "Second String: " << str2 << endl;

        fflush(stdin);
        getchar();
        return 0;
 }
```

Output of the program is:

```
First String:
Second String:
Enter a string: cpluspluslanguage
  After copying…
First String: cpluspluslanguage
Second String: cpluspluslanguage
```

Here is another example where we use cmath and the function log….

```
// Example using header cmath and function log

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

int main ()

{
        double number, ln;
        number = 4.5;

        ln = log (number);
        cout << "Logarithm of " << number << " is " << ln;

        fflush(stdin);
        getchar();
        return 0;
}
```

Output of the program is:

```
Logarithm of 4.5 is 1.50408
```

We hope these examples have given you fairly good idea of how to use a header and the functions contained in it. So start using the immense power provided to you by the enormous library of C++……..

Value addition: Advantage of Header files
Heading text: Header files
Body text: Header files provide two safeguards ○ All files are guaranteed to contain the same declaration for a global object or function. ○ If a declaration requires updating, only one change to the header file needs to be made.
Source:

1.3 Functions - III

After having learnt about functions, inline functions and setting default arguments, it is time to look deep into different ways of passing arguments to the functions. You will find it interesting to know that arguments can be passed in more ways than one, such as call by values, call by reference and references.

1.3.1 Learning Objectives

After reading this chapter you should be able to:

1. Pass the values of parameters only.
2. Pass the variables themselves as parameters.
3. Use references to functions.

1.3.2 Call by Value

After having learnt about how to make functions, inline functions and specifying default arguments, it is time to know a little bit about internal working of a function.

Functions are allotted some memory storage to work called its activation record. This memory remains associated with the function until it terminates. At that point, storage is automatically available for reuse.

When the caller program passes arguments to the function, each function parameter is provided storage within the function's activation record. In other words, a copy of the original argument is created inside the function's memory storage area.

This method of parameter passing is called call by value. The caller program passes the copy of the parameters and not the original parameters to the function. In this method, actual parameters are never touched, only their values are used for working inside the function.

For Example:

```
// Program to add 100 to a number passed as argument
#include<iostream>
using namespace std;
// Function declaration
int addition(int n);
// main() function
int main()
{
        int num, result;
        num = 10;
        cout << " The initial value of number : " << num << endl;
        result = addition (num);
        cout << " The final value of number : " << num << endl;
        cout << " The result is : " << result << endl;
        return (0);
}
 // Function definition
int addition(int num)    // Note the same name given to the variable in function
{
        num = num + 100;
        return (num);
}
```

The result of the program is:-

The initial value of number : 10

The final value of number : 10

The result is : 110

From the output of the program, you can see that a new copy of the passed parameter num is created inside the function. That is why the change in its value inside the function is not reflected in the main program even after the function call. Having the same name of the arguments in main() and function addition() does not matter, since a separate copy is created by addition() for its use.

1.3.2.1 Advantage of call by value

Since in a call by value, the contents of the arguments are not changed, there is no need for a programmer to save and restore argument values when making a function call. Without a pass-by-value mechanism, each parameter not declared const would have to be considered potentially altered with each function call. Pass-by-value has the least potential for harm and requires the least work to be done by a general user. Pass-by-value is therefore, a reasonable default mechanism for argument passing since it provides security to the calling program.

1.3.2.2 When call by value is not suitable

Call-by value is not suitable under certain circumstances:

- When a large class object must be passed as an argument. This is because the time and space costs to allocate and copy the class object onto activation record are often too high for real-world applications.
- When the values of the arguments need to be modified inside the function body. In this case another way of calling the parameters is used i.e. call by reference.

Value addition: Did you know?
Heading text: Default way of passing arguments
Body text: Default way of passing arguments is call by value i.e. the value of the argument is copied in memory for the function to use.
Source:

1.3.3 Call by Reference

Imagine that you have created a program with a call to function where you are passing parameters. You intended to write a function to swap the values of two variables passed as parameters. Let us look at the following program:

```
// Program to call a function to swap values of two variables

#include <iostream>

using namespace std;

// Function declaration

void swap( int, int );
```

```
// main() Program

int main()
{

        int a =  5;
        int b = 8;

        cout << "Values Before Swapping:" <<endl;
        cout << "a:" << a << "\t b:"  << b << endl;

        //Function call
        swap ( a, b );

        cout << "Values After Swapping:" <<endl;
        cout << "a:" << a << "\t b:"  << b << endl;

        fflush(stdin);
        getchar();

        return 0;

}

//Function Definition

void swap( int i, int j )
{
        int temp = j;
        j = i;
        i = temp;

}
```

What do you expect as the output of the above program???
Did the function swap the values of the variables?
No!

The output is:

```
Values Before Swapping:
a: 5    b: 8
Values After Swapping:
a: 5    b:8
```

Was the function swap not invoked??? Indeed it **was** invoked!!!

But still the desired effect was missing. Why?

Actually the function swapped the values of the local copies of the variables passed and not the original ones. So what is the solution to this problem??

Pass the original variables themselves instead of copies. This is possible by call by reference method of parameter passing.
In call by reference, instead of passing the copies of the variables, you pass the address of the variables. In this case, function does not create a copy of the passed parameter but uses the original parameter itself. It now knows where the parameter resides in memory and can therefore change its value, if needed.

To pass the address of a variable, ampersand (&) is prefixed to the variable name in the function parameter list.

The above program gets modified in the following way:

```
// Program to call a function to swap values of two variables

#include <iostream>

using namespace std;

// Function declaration

void swap( int &, int & );      //Note the &

// main() Program

int main()
{
        int a =  5;
        int b = 8;
        cout << "Values Before Swapping:" <<endl;
        cout << "a:" << a << "\t b:"  << b << endl;
        //Function call
        swap ( a, b );
        cout << "Values After Swapping:" <<endl;
        cout << "a:" << a << "\t b:"  << b << endl;
        fflush(stdin);
        getchar();
        return 0;
}
//Function Definition

void swap( int &i, int &j )
{
        int temp = j;
        j = i;
        i = temp;
```

```
}
```

The output received now is:

```
Values Before Swapping:
a: 5    b: 8
Values After Swapping:
a: 8    b:5
```

1.3.3.1 When call by reference is appropriate

Call by Reference is appropriate in cases:

- When you wish to change the original values of the parameters passed. This was the case in the above example.
- Second use of call by reference is when you need to return more than one parameters from the function. Since any change in a referenced variable is reflected back in calling program, it is in effect similar to returning more than one value.
- Third use is to pass large class objects to a function. (You will be doing classes later on).

Value addition: Common Mistake...

Heading text: Beware!

Body text: swap() can be implemented using pointers also in the following way:

```
//swap using pointers
void swap( int *i, *j)
{
      int temp = *j;

      *j = *i;
      *i = temp;
}
```

Now main() also needs to modify the function call to pass addresses of the variables instead of values:

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

Source:

1.3.4 References as return type

The default way of returning values from a function is by value. When a value is returned by the function, it is copied in the receiving variable in the calling program and this calling program has no way of changing its value.

But this default behavior can be overridden. A function can be declared to return a reference. In this case the address or reference is returned to the calling program. Now the calling program can modify this value too. This can be done by post-fixing return datatype by ampersand sign (&).

For Example:

```
//Function to return reference

int& greater (int  &i, int &j)


{
        if ( i > j )
                return i;
        else
                return j;
}
```

This function returns reference to either i or j depending on what is greater. Since now a function call returns a reference to either i or j, this function call can appear on the left hand side also.

1.3.4.1 Pitfalls of return by reference

The programmer should take care of the following pitfalls of return by reference.

- If you are returning a local object from a function, you have to be careful since the lifetime of local object is till the function terminates. After that reference may point to an undefined memory.

- Second thing to take care is that any modification of the value returned changes the actual object being returned. And this might not always be the intention. In order to prevent unintended modification of a reference return type, it should be declared as const.

1.4 Functions - IV

Have you ever thought about the scope of recognition of the variables declared in the functions? Are they recognized only in the function they are declared in or other functions for example, the main() function, can also access them? What about the variables declared in the main() function? What is the visibility of variables declared outside any function??? What will happen if we give same name to more than one function? Is it possible??? For finding answers to these queries, read on….

1.4.1 Learning Objectives

After reading this chapter you should be able to:

1. Know the scope and lifetime of variables declared inside and outside of any function.
2. Give same name to more than one function.
3. Decide when to write functions with the same name.

1.4.2 Scope of Variables

Having learnt how to create functions, the next step is to know and understand where a variable can be used i.e. lifetime of variables. As you must have noticed, there are different types of variables: some can be used only in main() function, some in user made functions while others in both of them. The variables used in a program are always declared. The location where you declare a variable controls its "visibility" and role in a program. The area where a variable is declared and can be accessed is referred to as its scope.

Depending upon where a variable can be used, the scope of the variables can broadly be classified as

- Local Variables
- Global Variables

1.4.2.1 Local Variables

Names of variables appearing inside a block of statement enclosed in a pair of curly braces {…..} have a local scope. You can use these variables only within that block and only after the actual declaration. In other words, if a variable is declared after an opening curly bracket "}", that variable is available and accessible until the first and next closing bracket "}". Such a variable is referred to as local. The variables defined local to the block of the function would be accessible only within that block of the function and not outside the function. The scope of the local variables is limited to the function in which these variables are declared.

Let us see this with a small example:

```
#include <iostream>

using namespace std;
```

```
//Function prototype

int function1(int,int);

//main() function

int main( )

{

	int b;

	int s=5,u=6;

	b=function1(s,u);

	cout<<"\n The Output is:"<<b;

	fflush(stdin);

	getchar();

	return 0;

}

//function definition

int function1(int x,int y)

{

	int z;

	z=x+y;

	return(z);

}
```

In the above program the variables x, y, z are accessible only inside the function function1() and their scope is limited only to this function and not outside it. Thus the variables x, y, z are local to the function function1(). Similarly one would not be able to access variable b inside the function function1() as such. This is because variable b is local to function main.

Value addition: Programming Tip….
Heading text: Block scope and Function scope
○ You can also create and define your own scope. Such a scope starts with an opening curly bracket and ends with a closing curly bracket. This scope is called a block. ○ If you declare a variable inside of a function but outside of any block, such a

variable has a function scope because it is accessible by any other variable that is inside of the same function.

- You cannot define a function in a local scope. i.e. you cannot define one function inside the body of another.

Source:

1.4.2.2 Global Variables

Global variables are the ones that are declared outside any function or any block, thus making them visible in any part of the program code. They can be used within all functions and outside all functions used in the program. The method of declaring global variables is to declare the variable outside the function or block.

For instance:

```
// Function to show the scope of global variable

 #include <iostream>
using namespace std;

 // global variable
 int g;

 //Function prototype
 void func1();

 //main() function
 int main( )
 {
        int a;       //local variable for main()
    {
          int b;  //local variable for block
          b=25;
          a=45;
          g=65;
          cout << "inside main() block"<< endl;
          cout << "b" << b<<endl;
          cout << "a" << a<<endl;
          cout << "g" << g<<endl;

     }
        a=50;
```

```
        cout << "inside main() after block"<< endl;
        cout << "a" << a <<endl;

        func1( );
        cout << "inside main() after function call"<< endl;
        cout << "g" << g<<endl;

        fflush(stdin);
        getchar();
        return 0;
   }

void func1( )
{
        cout << "inside function" << endl;
        g = 30;         //Scope of g is throughout the program and so is used between
                        //function calls
        cout << "g"<< g<< endl;
 }
```

In the above example, global variable g is declared outside any function and therefore is recognized throughout the program, in both the functions: main() and func1(). Scope of variable b inside main() is till the first braces shaded as yellow, while scope of variable a is till the end of main brace shaded as red.

So you see the scope of global variables is between the point of declaration and the end of program or compilation unit whereas scope of local variables is between the point of declaration and the end of innermost enclosing compound statement.

Value addition: Did you know?
Heading text: Scope supported by C++
Body text: C++ supports three forms of scope: local scope, namespace scope, and class scope. Local scope: It is a portion contained within a function definition or a function block. Actually each function is a distinct local scope. Local scope can also be specified within a function in the form of compound statements or the use of braces. Namespace scope: A programmer can define user-declared namespaces nested within the global scope using namespace definitions. Each such user-declared namespace is a different scope. The outermost namespace scope of a program is called global scope or global namespace scope. Identifiers declared in a portion of program that is not contained within any function declaration, function definition or a

class definition is recognized in that namespace.

Look at the example below that declares a namespace with variable i before the function main().

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

namespace one
{
  int i = 5;
}


int main ()
 {

  cout << "variable in namespace one: " << i << endl;       //ERROR!!!!
                                          //i is not recognized here
                                          //as it is local to namespace one


  fflush(stdin);
  getchar();
  return 0;
}
```

This program does not compile!!!

Now look at the variable z declared globally

```
// namespaces

#include <iostream>
using namespace std;

namespace one
{
  int i = 5;
}

int z = 2;

int main ()
 {

  //cout << "variable in namespace one: " << i << endl;
  cout << "variable in global scope: " << z;

  fflush(stdin);
  getchar();
  return 0;
}
```

Output is:
variable in global scope: 2

Note: We can use scope resolution operator to make i visible in function main(). Instead of writing simply i, we should write one::i to make i visible in main().

Class scope: Each class definition is a separate or distinct scope. An identifier declared in a class is recognized within a class. You will read more about classes in the coming chapters.

1.4.3 Overloaded Functions

So now you know that we cannot use variables with the same name in the same scope. But have you ever thought what about the functions? Can we use two or more functions with the same name in the same scope???

Imagine a situation where you need to write functions to calculate volume of a cube, a cylinder and a rectangular box. Since all three functions calculate volume, you do **not** want to give them separate names, instead would like that correct volume is returned depending upon the number of parameters passed. In other words, if only one parameter is passed, volume of cube is returned, if two, volume of cylinder is returned and if three parameters are passed, volume of rectangular box is returned. In other words, we want to use the same function name to create functions that perform a variety of different tasks. Is it possible to write such functions???

The answer is YES, we can have two or more functions of the same name by a method known as function overloading and the functions having the same name are known as overloaded functions.

Why do we need Function Overloading?

Function overloading is one of the most powerful features of C++ programming language and forms the basis of compile time polymorphism (you'll be learning more about polymorphism later on).

Many a times you'll be overloading the constructor function (covered later on) of a class.

How do we overload functions?

Function overloading allows multiple functions that provide a common operation on different parameter types to share a common name. Now if we write more than one function with the same name, how will the compiler know when to call which function? Actually you have to make those similarly-named functions differ in their signatures in one way or the other.

And this can be done by differing the functions in terms of either the number of parameters or datatypes of the parameters passed to them. Compiler will invoke correct function by looking at the parameter list in the function call.

So you see, nothing special needs to be done, you just need to declare two or more functions having the same name but either having different number of parameters or having parameters of different types.

Look at the following examples to show overloading in different ways:

Example 1: Overloaded Functions differing in NUMBER OF PARAMETERS

```
//Example Program in C++
#include<iostream>
using namespace std;
//Function Prototypes
int func(int i);
int func(int i, int j);
int main()
{
        cout << "Call for function with one parameter: "<<func(5)<< endl;
                                                        //func(int i) is invoked
        cout << "Call for function with two parameters: "<< func(5,7)<<endl;
                                                         //func(int i, int j) is invoked

        fflush(stdin);
        getchar();
        return 0;
}
int func(int i)
{
        return i;
}
int func(int i, int j)
{
        return i+j;
}
```

Output of the program is:

Call for function with one parameter: 5

Call for function with two parameters: 12

Example 2: Overloaded Functions differing in TYPE OF PARAMETERS

```
//Example Program in C++
#include<iostream>
using namespace std;
 //Funtion Prototypes
 int func(int i);
 double func(double i);
 int main()
 {
        cout<<"Call for function with integer datatype:  " << func(10)<< endl;
                                                          //func(int i) is invoked
        cout<<"Call for function with double datatype :  "<< func(10.201)<<endl;
                                                         //func(double i) is invoked
        fflush(stdin);
        getchar();
        return 0;
 }
 int func(int i)
 {
        return i;
 }
 double func(double i)
 {
         return i;
 }
```

Output of the program:

Call for function with integer datatype: 10
Call for function with double datatype : 10.201

Value addition: Common Mistake...
Heading text: return type of function
Body text: Please note that the return type of a function is not considered for differentiating two function calls from the calling program.
Source:

1.4.4 Friend functions and Virtual functions

Apart from the types of functions stated above, C++ supports two more types of functions: Friend functions and Virtual functions. These functions are for handling some specific tasks related to the class objects. Since classes will be covered in the next few units, we reserve the discussion of these functions till then.

1.5 Functions - V

Do you know what will happen if a function calls itself??? Is it possible??? How will the compiler deal with this situation where a function calls itself? In case it is possible for a function to call itself, how will the compiler keep track of the "instance" of the function called? Are there any problems which can be solved using this phenomenon??? Actually this process of a function calling itself is called "recursion". To know more about recursion, read on….

1.5.1 Learning objectives

After reading this chapter you should be able to:

1. Write a function that can call itself i.e. recursion is used.
2. Decide in what situation it is beneficial for a function to call itself.
3. Understand and write functions to find factorial of a function using recursion.
4. Understand and write functions to find terms of Fibonacci sequence.

1.5.1 Recursion – I

Discussion of functions is incomplete without talking about recursion! A very useful and interesting feature provided by C++ is that of recursion where a function calls itself. Instead of performing calculations by calling other functions, functions call themselves by passing in different parameters each time. This forms an almost loop-like effect, where each time a simpler calculation is performed by the function. This is repetitively done until a base-case is met.

A function that calls itself is referred to as a recursive function. Let us look at the following recursive function that acts as a counter.

Example:

```
// Program having a recursive counter function
#include<iostream>
using namespace std;
 //Function prototype
```

```
int count(int x);
// main() Function
int main()
{
        int x = 5;
        cout << count(x) <<endl; // Recursive function call
        fflush(stdin);
        getchar();
        return 0;
}
//Recursive function definition
int count(int x)
{
        if (x == 0)
        return 0;
        else
        {
                cout << x<< "    ";
                return count(x - 1);  // note: count(x) calls count(x-1)
        }
}
```

Do you feel it is a never ending loop as a dog chasing its tail? So it may look like. But actually it is not like this.

Well, sometimes it may become true also, in case you forget to have a terminating condition. In practice we should always check to see if a certain condition is true and in that case exit (return from) our function. Such a case where we end our recursion is called a **base case.** Additionally, just as in a loop, we must change some value and advance closer to our base case with every fresh call.

So what is the output of the above program???

It is:

5 4 3 2 1 0

How does this recursive program works which counts back from 5 to 0. It operates recursively, as the count function calls itself. Step-by-step procedure of how the recursive stack builds up, and then how it eventually executes itself back, down to determine an

answer, is given below:

1. The variable x is declared as 5.
2. x is passed into the count function.
3. x is not equal to 0 (since it is 5), so the else statement is executed.
4. The value of x, i.e. 5, is printed.
5. The count function returns count(5-1) or count(4).
6. The value of count(4) is not known, so the variable x with value 4 is passed into the count function.
7. The procedure above is repeated.
8. The value of x, i.e. 4, is printed out.
9. Again we don't know the value of count(3), so to figure it out, the above procedure is repeated until x is equal to 1.
10. 1 is printed out.
11. We don't know the value of count(0), so to find its value, variable x with value 0 is passed into the count function.
12. Now x == 0 i.e. first condition is true, so the function returns a 0.
13. Now that we know that the value of count(0) = 0, we can deduce the value of count(1), which we said was equal to count(0). Thus, count(2), count(3), count(4), and count(5) all return 0 as integers. However, the function has been constructed in such a way that it prints out the value of x before calling itself recursively with value one less than itself, therefore, it results in 5 4 3 2 1 0 being printed to the screen.

1.5.2.1 Advantage of Recursion over iteration?

If you notice carefully, the above recursion is essentially a loop like a "for loop" or a "while loop". So when do we prefer recursion to an iterative loop?

A recursive function is likely to run slower than its nonrecursive or iterative counterparts because of the overheads associated with the invocation of a function. We use recursion when we see that our problem can be reduced to a simpler problem that can be solved after further reduction. The recursive function is likely to be smaller and more easily understood.

1.5.2.2 Characteristics of Recursion

Every recursion should have the following characteristics.

1. A simple base case for which we have a solution and a return value. Sometimes there can be more than one base case.
2. A way of getting our problem closer to the base case, i.e. a way to reduce part of the problem to get a somewhat simpler problem.
3. A recursive call which passes the simpler problem back into the function.

The key to thinking recursively is to see the solution of the problem as a smaller version of the same problem. The key to solving recursive programming requirements is to imagine that the function does its job and then use it to solve the more complex cases. Here is how:

- Identify the base case(s) and what the base case(s) do. A base case is the simplest possible problem (or case) your function could be passed.
- Return the correct value for the base case. Your recursive function will then be comprised of an if-else statement where the base case returns one value and the non-base case(s) recursively call(s) the same function with a **smaller** parameter or set of data.
- Thus you decompose your problem into two parts: (1) The simplest possible case for which you have an answer (and return for), and (2) all other more complex cases which you will solve by returning the result of a second calling of your function. This second calling of your function (recursion) will pass on the complex problem but reduced by one value. This decomposition of the problem will actually be a complete, accurate solution for the problem for all cases other than the base case. And this case will eventually end in the base case at the last recursive call.

Value addition: Did you know?
Heading text: Use of Stack
Data structure Stack (last in first out) is used to keep track of all the function calls in a recursion.
Source:

1.5.3 Recursive Factorial Function

Let's look at another example of recursion. Suppose we have to write a function to calculate factorial of a number. For example factorial of 5 or 5! equals 5*4*3*2*1.

Can we also say that factorial of 5 is equal to 5*4!? Looking at the problem in this way, we have gained a valuable insight. We now can see our problem in terms of a simpler version of our problem and also know how to make our problem progressively simpler along with the problem defined in terms of itself. Don't we?

Now let's write the factorial function recursively.

```
// Function to calculate factorial if a number
int Fact( int n)
{
if( n == 1)
    return 1;
else
```

```
    {
    return ( n * (Fact( n-1)));
    }
}
```

In the function written above, note that the base case i.e. factorial of 1 is known and therefore, its return value is given in the first part of if condition. Now let us imagine that our function actually works, so we can use it to find factorial of other cases. If we are finding factorial of 5, we will simply return 5 * the result of factorial of 4. So we actually have the exact answer for all cases in the top level recursion. Notice that our problem is getting smaller on each recursive call because each time we call the function we give it a smaller number.

Now try running this program mentally to find factorial of number 2. Does it give the right answer? If it works for 1 (base case), then it must work for two also, since 2! merely returns 2 * factorial of 1. Yes it works and calculates 2! as 2 * 1 = 2. Does this work for 3! as well? Well, 3 must return 3 * factorial of 2. Now since we know that factorial of 2 works, factorial of 3 also works. Similarly it works for 4! and so on.

Winding Process:

Function called Function return

Fact(5) 5*Fact(4)
Fact(4) 4*Fact(3)
Fact(3) 3* Fact(2)
Fact(2) 2* Fact(1)
Terminating Point
Fact(1) 1

Unwinding Process

Fact(1) 1
Fact(2) 2*1
Fact(3) 3*2*1
Fact(4) 4*3*2*1
Fact(5) 5*4*3*2*1

Note: Make it a habit of writing the base case in the function as the first statement. Since forgetting the base case leads to infinite recursion.

Have you been able to write the main() function for the above written factorial function???

Value addition: Programming tip...
Heading text: when you forget base case

Body text: If you do not write the base case, in fact, your code won't run forever like an infinite loop, instead, you will eventually run out of stack space (memory) and get a run-time error or exception called a stack overflow.
Source:

1.5.4 Recursive Fibbonacci Function

To make the concept of recursion clearer, let us look at another example. You must be knowing, Fibonacci series is the series where the next term is the sum of last two terms.

Fibonacci Series: 1,1,2,3,5,8,13,21,34...

Each number, after the second, is the sum of the two numbers before it. And this keeps repeating. So what can be better way to find nth term of the series other than the recursion?

It is important to note that when a function calls itself, a new copy of that function is run. The local variables in the second version are independent of the local variables in the first, and they cannot affect one another directly, any more than the local variables in main() can affect the local variables in any function it calls. This has been illustrated by the example given below:

Look at the following example of the function which returns the nth term of the series:

```
// Function to calculate nth Fibonacci term
   int fibonacci (int n)
   {
         if (n < 3 )
    {
            return (1);
    }
    else
    {
          return( fibonacci(n-2) + fibonacci(n-1));
    }
 }
```

Can you write the main() function for the above function? Note the double call to the function fibonacci().

Have you been able to write it?

Good!

If not, look at the following code….

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

// Function Prototype
int fibonacci(int n);
// main() Function
int main()
{
    int n, result;
    cout << "Enter the term to find: " << endl;
    cin >> n;
    result = fibonacci(n);
    cout << result << " is the " << n << "th Fibonacci number\n";
    fflush(stdin);
    getchar();
    return 0;
}
```

Output of the program:

Enter the term to find:

10

55 is the 10^{th} Fibonacci number

How did we do it?

Examine the series carefully. The first two numbers are 1. Each subsequent number is the sum of the previous two numbers. Thus, the seventh number is the sum of the sixth and fifth numbers. More generally, the nth number is the sum of n - 2 and n - 1, as long as n > 2.

Recursive functions need a stop condition i.e. a base case. In the Fibonacci series, n < 3 is a stop condition.

The algorithm is:

1. Ask the user for a position or term in the series to find.
2. Call the fibonacci() function with that position, passing the value that the user entered.
3. The fibonacci() function examines the argument (n). If n < 3 it returns 1; otherwise, fibonacci() calls itself (recursively) passing in n-2, calls itself again passing in n-1, and returns the sum.

If you call fibonacci(1), it returns 1. If you call fibonacci(2), it returns 1. If you call fibonacci(3), it returns the sum of calling fibonacci(2) and fibonacci(1). Because fibonacci(2) returns 1 and fibonacci(1) returns 1, fibonacci(3) will return 2.

If you call fibonacci(4), it returns the sum of calling fibonacci(3) and fibonacci(2). We've established that fibonacci(3) returns 2 (by calling fibonacci(2) and fibonacci(1)) and that fibonacci(2) returns 1, so fibonacci(4) will sum these numbers and return 3, which is the fourth number in the series. And so on….

Value addition: Warning!!!
Heading text: Be Careful…
When you run this program, use a small number (less than 15). Because this program uses recursion, it can consume a lot of memory.
Source:

1.6 Functions - VI

After having learnt something about recursion, it is time to explore some more problems that need recursion to solve them. "GCD" is one such problem while "Ackermann" is another one that can be solved using recursion. To know more about these problems and their solutions, read on…

1.6.1 Learning Objectives

After reading this chapter you should be able to:

1. Have more insight into recursion.
2. Understand and write functions to find GCD of two numbers using recursion.
3. Understand and write functions to solve Ackermann equation using iterative, and recursion.

1.6.2 Recursion – II

Recursion is an interesting phenomenon for any language! A function calling itself… isn't it quite similar to human way of thinking? When we face a big problem, we start thinking in

terms of breaking it into simpler one that can be solved in a similar way…. Same as it is done in recursion.

Let us look at a few more problems that can be solved using recursion. Here we'll also have a feel of some other ways in which recursion can be used such as nested recursion.

1.6.3 Recursive "GCD" Function

Another example of a situation in which recursion is useful is the Euclidean algorithm, used to compute the greatest common divisor of two integers.

Function definition is:

$$gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, x \% y) & \text{if } x \geq y \text{ and } y > 0 \end{cases}$$

As you can see from the function definition, we can write the recursive program in the following way:

```
// Program to find Greatest Common Divisor using recursive function
#include<iostream>
using namespace std;

// Function definition
int GCD(int x, int y)
{
      if(y==0)           // base case, the programs stops if y reaches 0.
            return x;     //it returns the GCD as x
      else
            return GCD(y, x % y);    //recursion continues till y doesn't reach 0
}

//main() function
int main()
{
   int a,b;
   cout << "Enter first number" << endl;
```

```
    cin >> a;
    cout << "Enter second number" << endl;
    cin >> b;
    if (a < b)
    {
        int temp = a;
        a = b;
        b = temp;
     }

     cout << "Greatest Common Divisior (GCD) is " << GCD(a,b) << endl;
     fflush(stdin);
     getchar();
     return 0;
}
```

Output of the program:

```
Enter first number
24
Enter second number
6
Greatest Common Divisor (GCD) is 6
```

1.6.4 Ackermann Function

Ackermann function is defined recursively for non-negative integers m and n as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

This simple looking recursive function actually is not so primitive recursion. Notice the nested recursive call for the function!

This recursive function can be programmed in more than one ways. We will try to implement it in some of these ways.

1.6.4.1 Implementation using the formula directly

The most basic implementation is based directly on the formula specified above, resembling it almost identically. The only changes are that the greater-than-zero conditions are implicit due to previous conditions being false.

The function for this implementation is as follows. Please note that unsigned types have been used to avoid the possibility of illegal negative inputs. Also note that we are counting the number of times this function is called using the variable "calls". This is just to emphasize that a large number of calls are needed with some implementations….

```
// Program for Ackermann function using direct formula
#include <iostream>
 using namespace std;
 static unsigned int calls;


// Ackermann function directly from formula

unsigned int direct_ackermann(unsigned int m, unsigned int n)

 {
        calls++;
        if (m == 0)
                return n + 1;
        else    if (n == 0)
                return direct_ackermann(m - 1, 1);
                else
                        return direct_ackermann(m - 1, direct_ackermann(m, n - 1));
}

// main() function that calls direct_ackermann() function

int main()
 {
    unsigned int m, n, result;
    cout << endl << " Enter m: ";
    cin >> m;
    cout << endl << "Enter n: ";
    cin >> n;
```

```
    calls = 0;
    result = direct_ackermann(m, n);
    cout << endl << "Ackerman direct: " << result << endl;
    cout<< "Calls: "<< calls <<endl;
    fflush(stdin);
    getchar();
    return 0;
}
```

Output of the program:

```
Enter m: 2
Enter n: 1
Ackerman direct: 5
Calls:14
```

In the above function, we have used nested recursion. This implementation makes a lot of calls to the function and therefore takes a lot of time. We also get the number of calls to the function printed in the output just for comparison sake….

Value addition: Beware!!!
Heading text: Growth of Ackermann function
Note that this function grows very quickly -- even Ackerman(4, 3) cannot be feasibly computed on ordinary computers. Therefore do not try with bigger values.
Source:

1.6.4.2 Implementation using partially iterative method

If you have tried the above implementation on computer, you must have noticed the large amount of time it takes for computation of even small values. If we can turn two of the recursive calls above into iterative constructs, this will greatly reduces the number of recursive calls, although the number still remains prohibitive even for small inputs.

```
// Program for Ackermann function using partially iterative formula
#include <iostream>
 using namespace std;
 static unsigned int calls;

// Function using partially iterative method

unsigned int iterative_ackermann(unsigned int m, unsigned int n)
```

```
{
   calls++;
   while (m != 0)
          {
                 if (n == 0)
                 {
                        n = 1;
                 }
                 else
                 {
                        n = iterative_ackermann(m, n - 1);
                 }
                 m--;
          }
   return n + 1;
}
// main() function that calls iterative_ackermann() function

int main()
 {
    unsigned int m, n, result;
    cout << endl << " Enter m: ";
    cin >> m;
    cout << endl << "Enter n: ";
    cin >> n;
    calls = 0;
    result = iterative_ackermann(m, n);
    cout << endl << "Ackerman iterative: " << result << endl;
    cout<< "Calls: "<< calls <<endl;
    fflush(stdin);
    getchar();
    return 0;
}
```

Output of the program:

Enter m: 2
Enter n: 1
Ackerman iterative: 5
Calls: 6

Did you notice the number of calls??? Although fewer than direct, they still are large for greater values.

1.6.4.3 Implementation using the pre-computed values in formula

We see from the previous section that even small values require a large number of calls to the function. If you make small changes in the above function i.e. pre-compute for small m values, things can be improved a lot.

It is possible to prove the following useful formulas for small values of the input m:

$A(0,n) = n + 1$
$A(1,n) = n + 2$
$A(2,n) = 2n + 3$
$A(3,n) = 2^{n+3} - 3$

Using these, we can drastically reduce runtime not only for these inputs, but for larger inputs which make recursive calls with these inputs. We perform the exponentiation using shifting. The code for Ackermann function that uses pre-computed values in formula is as follows:

```
// Program for Ackermann function using pre-computed values

#include <iostream>
using namespace std;
static unsigned int calls;

// Ackermann function using pre-computed values
unsigned int precomputed_ackermann(unsigned int m, unsigned int n)
{
   calls++;
   while(1)
    {
          switch(m)
          {
          case 0:  return n + 1;
          case 1:  return n + 2;
          case 2:  return (n << 1) + 3;
          case 3:  return (1 << (n+3)) - 3;
          default:
                    if (n == 0)
                    {
                              n = 1;
```

```
                }
                else
                {
                        n = precomputed _ackermann(m, n - 1);
                }
        m--;
        break;
      }
   }
}

// main() function that calls pre-computed values ackermann() function

int main()
 {
    unsigned int m, n, result;
    cout << endl << " Enter m: ";
    cin >> m;
    cout << endl << "Enter n: ";
    cin >> n;
    calls = 0;
    result = precomputed_ackermann(m, n);
    cout << endl << "Ackerman precomputed: " << result << endl;
    cout<< "Calls: "<< calls <<endl;
    fflush(stdin);
    getchar();
    return 0;
}
```

Output of the program:

```
Enter m: 2
Enter n: 1
Ackerman precomputed: 5
Calls:1
```

Note: This doesn't quite work correctly for *n* + 3 equal to the word size, for which 1 << (n+3) is undefined due to overflow, but we ignore this here.

So did you notice the results?

A comparison of all three:

If we run for ackermann(2,1), results for the three functions are:

Direct	5	calls	14
Iterative	5	calls	6
Precomputed	5	calls	1

If we run for ackermann(3,2), results are:

Direct	29	calls	541
Iterative	29	calls	258
Precomputed	29	calls	1

If we run for ackermann(4,1), results are:

Direct	65533	calls	2862984010
Iterative	65533	calls	1431459240
Precomputed	65533	calls	2

Looking at the results, especially the number of calls, I hope you appreciate the benefit of precomputed Ackermann function implementation.

Value addition: Thing to know
Heading text: Values of Ackermann function..
This section explains some methods of computing the Ackermann function. Of course, since most values of the Ackermann function exceed the size of a machine word (even a 64-bit word), these implementations are more useful for illustration than practical computation of Ackermann function values.
Source:

Summary

- Function is a named group of program statements performing a particular task which can be invoked multiple times from different places in a program.

- When a function is called, execution control goes to first statement inside the function body. At the end of function, control comes back to the next line in the calling program.
- Function consists of three things:
    - Function call: The caller program use function call to invoke function.
    - Function Prototype: Before the call for function is encountered, function prototype tells the compiler about the return type, and number and datatype of the arguments being received by function.
    - Function Definition: The actual body of function is written here.
- User-defined functions help users in dividing big problems into smaller, more manageable modules.
- Only one value can be returned from function while any number of parameters can be passed from calling program to the function.
- main() is a special function serving as the starting point for execution in C++ and having int as return type.
- Inline function is a function whose code is replaced at the point of function call by the compiler.
- It depends upon the compiler to expand the code of an inline function or not to expand it.
- Default arguments are the arguments in the parameter list of a function, having most appropriate value specified as default in the function. In case no value is supplied by the caller program, function uses this default value.
- Const arguments are the arguments specified as constant in the function so that they cannot be modified even by mistake.
- C++ provides a library of number of built-in, prewritten functions stored in separate header files which can be used by programmers in their programs.
- The calling program can call the function by passing arguments in two ways: Call by value and Call by reference.
- Call by value is the method of passing arguments where another copy of the variables is created for the use of functions.
- Any changes made to the arguments inside the functions are **not** reflected in the calling program.
- Call by reference is the method of passing arguments where the address of the variable is passed to the function instead of the value of the variable.
- Any changes made to the arguments inside the functions are reflected in the calling program also.
- Default way of passing the arguments is call by value.
- References can be used to return values from the function by returning the address of the argument to be returned instead of the value itself.
- Scope of variable is the area in the program where that variable is visible.
- Depending upon where a variable can be used, the scope of the variables can broadly be classified as Local Variables and Global Variables.
- Local variables are the variables that are declared inside a function and hence are visible only in that function.
- Global variables are the variables that are declared outside any function and hence are visible to all functions.
- Function overloading is the process of giving two or more functions same name i.e. they share the same name.
- To resolve the confusion of which function is to be invoked at a function call, each function should have a different signature.
- Different signature is distinguished by different number of parameters or different datatypes of parameters.
- Recursion is the process of a function calling itself.

- In a recursive function, one should be careful to include the termination condition known as the base case.
- Recursion is used to reduce a problem into smaller problems that can be solved by further breaking down problem into similar problem.
- Factorial of a number and Fibonacci series can be found using recursion with computer.
- GCD of two numbers and Ackermann equation are two other problems that can be solved more comfortably using recursion.

Exercises

1.1 What is the return type of the function with prototype: "int func(char a, float b, double c);"
A. char
B. int
C. float
D. double

1.2 Which of the following is a valid function call (assuming the function exists)?
A. abc;
B. abc a, b;
C. abc();
D. int abc();

1.3 Which of the following is a complete function?
A. int qaz();
B. int qaz(int x) {return x=x+1;}
C. void qaz(int) {cout<<"Hello"}
D. void qaz(x) {cout<<"Hello"}

1.4 Write a C++ program which uses a function to find sum of numbers between two given numbers. These two numbers are passed to the function and the sum is returned from it.

1.5 Write a C++ program that invokes a function eql() to find whether four numbers a, b, c, d passed to eql() satisfy the equation a3 + b3 +c3 = d3 or not. eql() should return 0 if the above equation is satisfied with the passed numbers otherwise it returns -1.

1.6 In what conditions will you use inline functions?

1.7 How is an inline function different from a preprocessor macro?

1.8 Given the following code fragment:

```
int main()
{
        float abc(float, float);
```

```
                    …
                    …
        }
        void qaz(void)
        {
                float x, y, s;
                cin >>x;
                cin >> y;
                ….
                s = abc(x,y);
        }
```

Will the above function work? Why?

1.9 When are the default arguments required in a function?

1.10 Can the effect of default arguments be achieved by overloading functions? How?

1.11 Write a declaration for a function called funct() that takes two arguments and returns a char value. The first argument is int and cannot be modified. The second argument is float with a default value of 3.14.

1.12 Create two identical functions, **f1( )** and **f2( )**. Inline **f1( )** and leave **f2( )** as an non-inline function. Use the Standard C++ Library function to mark the starting point and ending points and compare the two functions to see which one is faster.

1.13 Identify the problem with the following code:

```
        void large(int &i, int &j);
        int main()
        {
                large (1,2);
        }

        void large(int &a, int &b)
        {
                if (a > b)
                a = -1;
                else
                b= -1;
        }
```

How can we correct it?

1.14 Write a function that computes x^y by using recursion. Write a C++ main() function to test it. (Hint: $5^4 = 5 * 5^3$).

1.15 What is overloading of a function? When can it be useful?

1.16 GCD function can be implemented in nonrecursive way using iterations. Write the function which calculates GCD in iterative way.

Glossary

Base case: A case or a termination condition where we end our recursion is called a base case.

Built-in library functions: These are the set of pre-written functions stored in header files of C++ that can be used by programmers in their programs.

Call by reference: A method of passing parameters to the functions where the addresses of the parameters are passed to the function.

Call by value: A method of passing parameters to the functions where the parameters are copied for the use of functions.

const arguments: It is a function argument that cannot be changed inside the function body, i.e. it remains constant.

Default arguments: A default argument is a value specified in the function definition that is most appropriate argument value for a function parameter in majority of the cases.

Function: It is a named group of program statements performing a particular task which can be invoked multiple times from different places in a program.

Function Prototype: Declaration of a function telling the compiler about the number and type of the arguments passed to it and the return type of function.

Global variables: Variables declared outside any function and therefore recognized in all functions are global functions.

Inline Functions: They are the functions that are expanded i.e. function call is replaced by the function definition at the place where they are called.

Local variables: Variables declared inside a function and recognized only in that function are local variables of that function.

Overloaded functions: Two or more functions having the same name but different signatures are called overloaded functions.

Parameters: Values passed to and from the functions, also called arguments.

Recursion: The method of a function calling itself is called recursion.

Recursive function: A function that calls itself is referred to as a recursive function.

Scope of variables: The area pf the program in which a variable is recognized and can be used is the scope of variable.

References

1. Lippman, Lajoie, "C++ Primer", Third Edition, Addison Wesley.
2. Balagurusamy, "Object Oriented Programming with C++", Fourth Edition, Tata McGraw Hill.