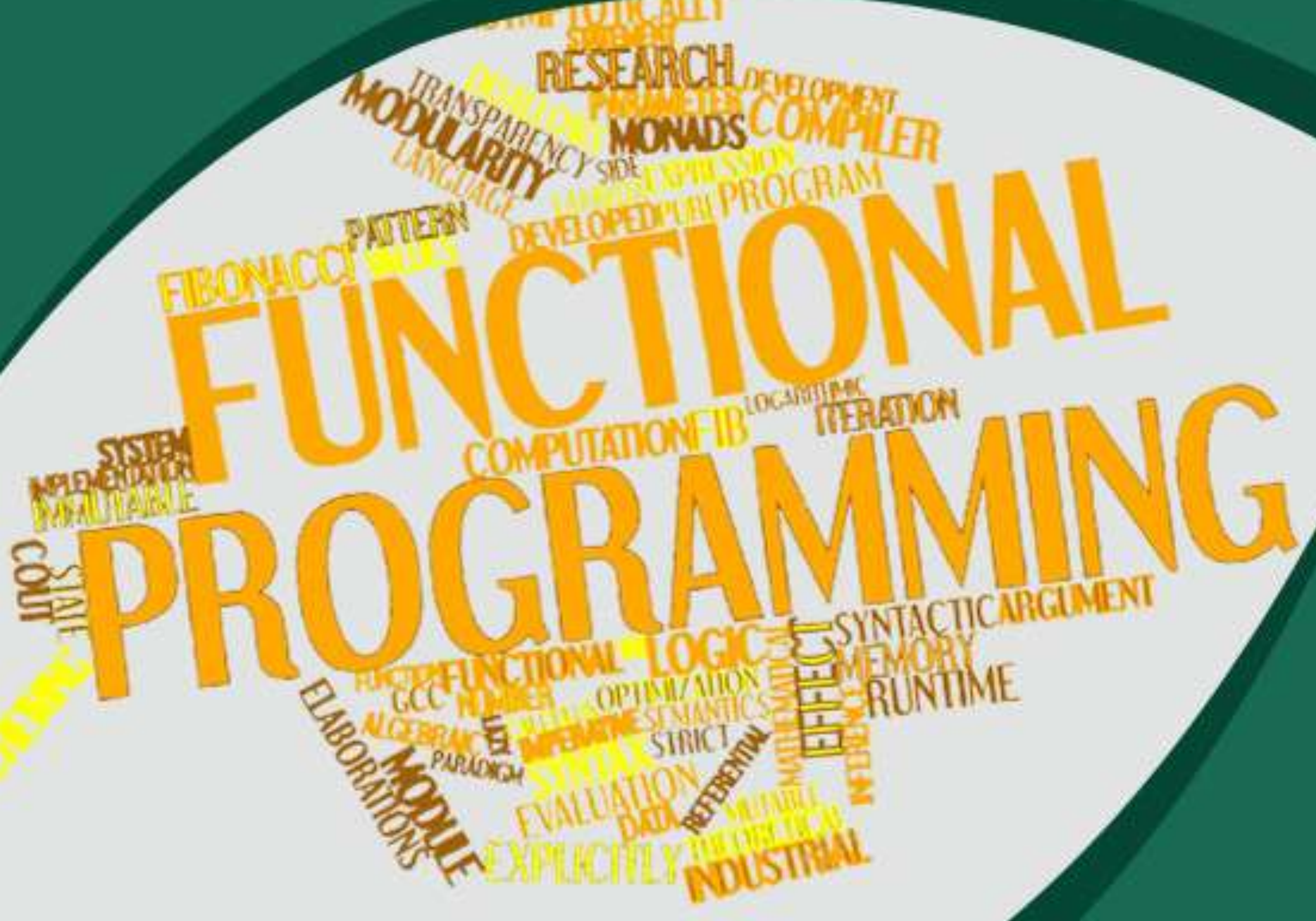


tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

This tutorial provides a brief overview of the most fundamental concepts of functional programming languages in general. In addition, it provides a comparative analysis of object-oriented programming and functional programming language in every example.

Audience

This tutorial will help all those readers who are keen to understand the basic concepts of functional programming. It is a very basic tutorial that has been designed keeping in mind the requirements of beginners.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of Computer Programming terminologies in general and a good exposure to any programming language such as C, C++, or Java.

Copyright & Disclaimer

Table of Contents

About the Tutorial i

Audience i

Prerequisites i

Copyright & Disclaimer i

Table of Contents ii

1. FUNCTIONAL PROGRAMMING – INTRODUCTION 1

Functional Programming – Characteristics 1

Functional Programming – Advantages 1

Functional Programming vs. Object-oriented Programming 2

Efficiency of a Program Code 3

2. FUNCTIONAL PROGRAMMING – FUNCTIONS OVERVIEW 4

Function Prototype 6

Function Signature 6

3. FUNCTIONAL PROGRAMMING – FUNCTION TYPES 7

Predefined Functions 7

User-defined Functions 8

4. FUNCTIONAL PROGRAMMING – CALL BY VALUE 12

5. FUNCTIONAL PROGRAMMING – CALL BY REFERENCE 14

6. FUNCTIONAL PROGRAMMING – FUNCTION OVERLOADING 16

7. FUNCTIONAL PROGRAMMING – FUNCTION OVERRIDING 18

8. FUNCTIONAL PROGRAMMING – RECURSION 20

9. FUNCTIONAL PROGRAMMING – HIGHER ORDER FUNCTIONS 22

10. FUNCTIONAL PROGRAMMING – DATA TYPES 24

Data Types Supported by C++ **24**

Data Types Supported by Java **25**

Data Types Supported by Erlang **25**

11. FUNCTIONAL PROGRAMMING – POLYMORPHISM 29

12. FUNCTIONAL PROGRAMMING – STRINGS 32

13. FUNCTIONAL PROGRAMMING – LISTS 36

14. FUNCTIONAL PROGRAMMING – TUPLE 41

Operations on Tuples **42**

15. FUNCTIONAL PROGRAMMING – RECORDS 44

16. FUNCTIONAL PROGRAMMING – LAMBDA CALCULUS 48

17. FUNCTIONAL PROGRAMMING – LAZY EVALUATION 51

18. FUNCTIONAL PROGRAMMING – FILE I/O OPERATIONS 52

Writing into a File **52**

Reading from a File **53**

Delete an Existing File **54**

Determining the Size of a File **55**

1. Functional Programming – Introduction

Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

Functional programming languages are categorized into two groups, i.e.:

- **Pure Functional Languages:** These types of functional languages support only the functional paradigms. For example: Haskell.

- **Impure Functional Languages:** These types of functional languages support the functional paradigms and imperative style programming. For example: LISP.

Functional Programming – Characteristics

The most prominent characteristics of functional programming are as follows:

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.

- Functional programming supports **higher-order functions** and **lazy evaluation** features.

- Functional programming languages don’t support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.

- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

Functional Programming – Advantages

Functional programming offers the following advantages:

- **Bugs-Free Code:** Functional programming does not support **state**, so there are no side-effect results and we can write error-free codes.

- **Efficient Parallel Programming:** Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.

- **Efficiency:** Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.

- **Supports Nested Functions:** Functional programming supports Nested Functions.

- **Lazy Evaluation:** Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

As a downside, functional programming requires a large memory space. As it does not have state, you need to create new objects every time to perform actions.

Functional Programming is used in situations where we have to perform lots of different operations on the same set of data.

- Lisp is used for artificial intelligence applications like Machine learning, language processing, Modeling of speech and vision, etc.

- Embedded Lisp interpreters add programmability to some systems like Emacs.

Functional Programming vs. Object-oriented Programming

The following table highlights the major differences between functional programming and object-oriented programming:

Functional Programming	OOP
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “What you are doing”	Focus is on “How you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side-effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.
It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.
Supports both "Abstraction over Data" and "Abstraction over Behavior".	Supports only "Abstraction over Data".

Efficiency of a Program Code

The efficiency of a programming code is directly proportional to the algorithmic efficiency and the execution speed. Good efficiency ensures higher performance.

The factors that affect the efficiency of a program includes:

- The speed of the machine
- Compiler speed
- Operating system
- Choosing right Programming language
- The way of data in a program is organized
- Algorithm used to solve the problem

The efficiency of a programming language can be improved by performing the following tasks:

- By removing unnecessary code or the code that goes to redundant processing.
- By making use of optimal memory and nonvolatile storage
- By making the use of reusable components wherever applicable.
- By making the use of error & exception handling at all layers of program.
- By creating programming code that ensures data integrity and consistency.
- By developing the program code that's compliant with the design logic and flow.

An efficient programming code can reduce resource consumption and completion time as much as possible with minimum risk to the operating environment.

2. Functional Programming – Functions Overview

In programming terms, a **function** is a block of statements that performs a specific task. Functions accept data, process it, and return a result. Functions are written primarily to support the concept of reusability. Once a function is written, it can be called easily, without having to write the same code again and again.

Different functional languages use different syntax to write a function.

Prerequisites to Writing a Function

Before writing a function, a programmer must know the following points:

- Purpose of function should be known to the programmer.
- Algorithm of the function should be known to the programmer.
- Functions data variables & their goal should be known to the programmer.
- Function's data should be known to the programmer that is called by the user.

Flow Control of a Function

When a function is "called", the program "transfers" the control to execute the function and its "flow of control" is as below:

- The program reaches to the statement containing a "function call".
- The first line inside the function is executed.
- All the statements inside the function are executed from top to bottom.
- When the function is executed successfully, the control goes back to the statement where it started from.
- Any data computed and returned by the function is used in place of the function in the original line of code.

Syntax of a Function

The general syntax of a function looks as follows:

```
returnType functionName(type1 argument1, type2 argument2, . . . )
{
     // function body
}
```

Defining a Function in C++

Let’s take an example to understand how a function can be defined in C++ which is an object-oriented programming language. The following code has a function that adds two numbers and provides its result as the output.

```
#include <stdio.h>
int addNum(int a, int b);              // function prototype

int main()
{
    int sum;
    sum = addNum(5,6);                 // function call
    printf("sum = %d",sum);
    return 0;
}

int addNum (int a,int b)               // function definition
{
    int result;
    result = a+b;
    return result;                     // return statement
}
```

It will produce the following output:

```
Sum=11
```

Defining a Function in Erlang

Let’s see how the same function can be defined in Erlang, which is a functional programming language.

```
-module(helloworld).
-export([add/2,start/0]).

add(A,B) ->
   C = A+B,
   io:fwrite("~w~n",[C]).
start() ->
   add(5,6).
```

It will produce the following output:

```
11
```

Function Prototype

A function prototype is a declaration of the function that includes return-type, function-name & arguments-list. It is similar to function definition without function-body.

For Example: Some programming languages supports function prototyping & some are not.

In C++, we can make function prototype of function 'sum' like this:

```
int sum(int a, int b)
```

Note: Programming languages like Python, Erlang, etc doesn't supports function prototyping, we need to declare the complete function.

What is the use of function prototype?

The function prototype is used by the compiler when the function is called. Compiler uses it to ensure correct return-type, proper arguments list are passed-in, & their return-type is correct.

Function Signature

A function signature is similar to function prototype in which number of parameters, data-type of parameters & order of appearance should be in similar order. For Example:

```
void Sum(int a, int b, int c);              // function 1


void Sum(float a, float b, float c);         // function 2


void Sum(float a, float b, float c);         // function 3
```

Function1 and Function2 have different signatures. Function2 and Function3 have same signatures.

Note: Function overloading and Function overriding which we will discuss in the subsequent chapters are based on the concept of function signatures.

- Function overloading is possible when a class has multiple functions with the same name but different signatures.
- Function overriding is possible when a derived class function has the same name and signature as its base class.

3. Functional Programming – Function Types

Functions are of two types:

- Predefined functions
- User-defined functions

In this chapter, we will discuss in detail about functions.

Predefined Functions

These are the functions that are built into Language to perform operations & are stored in the Standard Function Library.

For Example: 'Strcat' in C++ & 'concat' in Haskell are used to append the two strings, 'strlen' in C++ & 'len' in Python are used to calculate the string length.

Program to print string length in C++

The following program shows how you can print the length of a string using C++:

```
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;

int main()
{
	char str[20]="Hello World";
	int len;
	len=strlen(str);
	cout<<"String length is: "<<len;
	return 0;
}
```

It will produce the following output:

```
String length is: 11
```

Program to print string length in Python

The following program shows how to print the length of a string using Python, which is a functional programming language:

```
str = "Hello World";

print("String length is: ", len(str))
```

It will produce the following output:

```
String length is: 11
```

User-defined Functions

User-defined functions are defined by the user to perform specific tasks. There are four different patterns to define a function:

- Functions with no argument and no return value
- Functions with no argument but a return value
- Functions with argument but no return value
- Functions with argument and a return value

Functions with no argument and no return value

The following program shows how to define a function with no argument and no return value **in C++**:

```
#include <iostream>

using namespace std;

void function1()

{

    cout <<"Hello World";

}


int main()

{

     function1();

     return 0;

}
```

It will produce the following output:

```
Hello World
```

The following program shows how you can define a similar function (no argument and no return value) **in Python**:

```
def function1():
   print ("Hello World")


function1()
```

It will produce the following output:

```
Hello World
```

Functions with no argument but a return value

The following program shows how to define a function with no argument but a return value **in C++**:

```
#include <iostream>
using namespace std;
string function1()
{
    return("Hello World");
}


int main()
{
     cout<<function1();
     return 0;
}
```

It will produce the following output:

```
Hello World
```

The following program shows how you can define a similar function (with no argument but a return value) **in Python**:

```
def function1():
    return "Hello World"
res = function1()
print(res)
```

It will produce the following output:

```
Hello World
```

Functions with argument but no return value

The following program shows how to define a function with argument but no return value **in C++**:

```
#include <iostream>
using namespace std;
void function1(int x, int y)
{
     int c;
     c=x+y;
     cout<<"Sum is: "<<c;
}

int main()
{
     function1(4,5);
     return 0;
}
```

It will produce the following output:

```
Sum is: 9
```

The following program shows how you can define a similar function **in Python**:

```
def function1(x,y):
    c = x + y
    print("Sum is:",c)
function1(4,5)
```

It will produce the following output:

```
Sum is: 9
```

Functions with argument and a return value

The following program shows how to define a function **in C++** with no argument but a return value:

```
#include <iostream>
using namespace std;
int function1(int x, int y)
{
     int c;
     c=x+y;
     return c;
}
int main()
{
     int res;
     res=function1(4,5);
     cout<<"Sum is: "<<res;
     return 0;
}
```

It will produce the following output:

```
Sum is: 9
```

The following program shows how to define a similar function (with argument and a return value) **in Python**:

```
def function1(x,y):
    c = x + y
    return c

res = function1(4,5)
print("Sum is ",res)
```

It will produce the following output:

```
Sum is 9
```

4. Functional Programming – Call by Value

After defining a function, we need pass arguments into it to get desired output. Most programming languages support **call by value** and **call by reference** methods for passing arguments into functions.

In this chapter, we will learn "call by value" works in an object-oriented programming language like C++ and a functional programming language like Python.

In Call by Value method, the **original value cannot be changed**. When we pass an argument to a function, it is stored locally by the function parameter in stack memory. Hence, the values are changed inside the function only and it will not have an effect outside the function.

Call by Value in C++

The following program shows how Call by Value works in C++:

```
#include <iostream>
using namespace std;
void swap(int a, int b)
{
int temp;
     temp=a;
     a=b;
     b=temp;
     cout<<"\n"<<"value of a inside the function: "<<a;
     cout<<"\n"<<"value of b inside the function: "<<b;
}

int main()
{
int a=50, b=70;
     cout<<"value of a before sending to function: "<<a;
     cout<<"\n"<<"value of b before sending to function: "<<b;
     swap(a, b);  // passing value to function
     cout<<"\n"<<"value of a after sending to function: "<<a;
     cout<<"\n"<<"value of b after sending to function: "<<b;
     return 0;
}
```

It will produce the following output:

```
value of a before sending to function: 50
value of b before sending to function: 75
value of a inside the function: 75
value of b inside the function: 50
value of a after sending to function: 50
value of b after sending to function: 75
```

Call by Value in Python

The following program shows how Call by Value works in Python:

```
def swap(a,b):
   t=a;
   a=b;
   b=t;
   print "value of a inside the function: :",a
   print "value of b inside the function: ",b

# Now we can call the swap function
a=50
b=75
print "value of a before sending to function: ",a
print "value of b before sending to function: ",b
swap(a,b)
print "value of a after sending to function: ", a
print "value of b after sending to function: ",b
```

It will produce the following output:

```
value of a before sending to function:  50
value of b before sending to function:  75
value of a inside the function: : 75
value of b inside the function:  50
value of a after sending to function:  50
value of b after sending to function:  75
```

5. Functional Programming – Call by Reference

In Call by Reference, the **original value is changed** because we pass reference address of arguments. The actual and formal arguments share the same address space, so any change of value inside the function is reflected inside as well as outside the function.

Call by Reference in C++

The following program shows how Call by Reference works in C++:

```
#include <iostream>
using namespace std;
void swap(int *a, int *b)
{
     int temp;
     temp=*a;
     *a=*b;
     *b=temp;
     cout<<"\n"<<"value of a inside the function: "<<*a;
     cout<<"\n"<<"value of b inside the function: "<<*b;
}

int main()
{
     int a=50, b=75;
     cout<<"\n"<<"value of a before sending to function: "<<a;
     cout<<"\n"<<"value of b before sending to function: "<<b;
     swap(&a, &b);  // passing value to function
     cout<<"\n"<<"value of a after sending to function: "<<a;
     cout<<"\n"<<"value of b after sending to function: "<<b;
     return 0;
}
```

It will produce the following output:

```
value of a before sending to function: 50
value of b before sending to function: 75
value of a inside the function: 75
value of b inside the function: 50
value of a after sending to function: 75
value of b after sending to function: 50
```

Call by Reference in Python

The following program shows how Call by Reference works in Python:

```
def swap(a,b):
   t=a;
   a=b;
   b=t;
   print "value of a inside the function: :",a
   print "value of b inside the function: ",b
   return(a,b)

# Now we can call swap function
a=50
b=75
print "value of a before sending to function: ",a
print "value of b before sending to function: ",b
x=swap(a,b)
print "value of a after sending to function: ", x[0]
print "value of b after sending to function: ",x[1]
```

It will produce the following output:

```
value of a before sending to function:  50
value of b before sending to function:  75
value of a inside the function: : 75
value of b inside the function:  50
value of a after sending to function:  75
value of b after sending to function:  50
```

6. Functional Programming – Function Overloading

When we have multiple functions with the same name but different parameters, then they are said to be overloaded. This technique is used to enhance the readability of the program.

There are two ways to overload a function, i.e.:

- Having different number of arguments
- Having different argument types

Function overloading is normally done when we have to perform one single operation with different number or types of arguments.

Function Overloading in C++

The following example shows how function overloading is done in C++, which is an object-oriented programming language:

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

void addnum(int,int);
void addnum(int,int,int);
int main()
{
     addnum (5,5);
     addnum (5,2,8);
     return 0;
}
void addnum (int x, int y)
{
     cout<<"Integer number: "<<x+y<<endl;
}
void addnum (int x, int y, int z)
{
     cout<<"Float number: "<<x+y+z<<endl;
}
```

It will produce the following output:

```
Integer number: 10
Float number: 15
```

Function Overloading in Erlang

The following example shows how to perform function overloading in Erlang, which is a functional programming language:

```
-module(helloworld).
-export([addnum/2,addnum/3,start/0]).

addnum(X,Y) ->
   Z = X+Y,
   io:fwrite("~w~n",[Z]).

addnum(X,Y,Z) ->
   A = X+Y+Z,
   io:fwrite("~w~n",[A]).

start() ->
   addnum(5,5),
   addnum(5,2,8).
```

It will produce the following output:

```
10
15
```

7. Functional Programming – Function Overriding

When the base class and derived class have member functions with exactly the same name, same return-type, and same arguments list, then it is said to be function overriding.

Function Overriding using C++

The following example shows how function overriding is done in C++, which is an object-oriented programming language:

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

class A
{  public:
   void display()
   {
     cout<<"Base class";
   }
};
class B:public A
{
  public:
  void display()
  {
     cout<<"Derived Class";
  }
};
int main()
    {
      B obj;
      obj.display();
      return 0;
    }
```

It will produce the following output:

```
Derived Class
```

Function Overriding using Python

The following example shows how to perform function overriding in Python, which is a functional programming language:

```
class A(object):
    def disp(self):
        print "Base Class"

class B(A):
    def disp(self):
        print "Derived Class"

x = A()
y = B()

x.disp()
y.disp()
```

Output

```
Base Class
Derived Class
```

8. Functional Programming – Recursion

A function that calls itself is known as a recursive function and this technique is known as recursion. A recursion instruction continues until another instruction prevents it.

Recursion in C++

The following example shows how recursion works in C++, which is an object-oriented programming language:

```
#include <stdio.h>
long int fact(int n);

int main()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    printf("Factorial of %d = %ld", n, fact(n));
    return 0;
}
long int fact(int n)
{
    if (n >= 1)
        return n*fact(n-1);
    else
        return 1;
}
```

It will produce the following output:

```
Enter a positive integer: 5
Factorial of 5 = 120
```

Recursion in Python

The following example shows how recursion works in Python, which is a functional programming language:

```
def fact(n):
     if n == 1:
       return n
     else:
       return n* fact (n-1)

# accepts input from user
num = int(input("Enter a number: "))

# check whether number is positive or not
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
else:
   print("The factorial of " + str(num) +  " is " + str(fact(num)))
```

It will produce the following output:

```
Enter a number: 6
The factorial of 6 is 720
```

9. Functional Programming – Higher Order Functions

A higher order function (HOF) is a function that follows at least one of the following conditions:

- Takes on or more functions as argument
- Returns a function as its result

HOF in PHP

The following example shows how to write a higher order function in PHP, which is an object-oriented programming language:

```
<?php


$twice = function($f, $v)
{
    return $f($f($v));
};


$f = function($v)
{
    return $v + 3;
};
echo($twice($f, 7));
```

It will produce the following output:

```
13
```

HOF in Python

The following example shows how to write a higher order function in Python, which is an object-oriented programming language:

```
def twice(function):
     return lambda x: function(function(x))


def f(x):
     return x + 3
```

```
g = twice(f)


print g(7)
```

It will produce the following output:

```
13
```

10 . Functional Programming – Data Types

A data-type defines the type of value an object can have and what operations can be performed on it. A data type should be declared first before being used. Different programming languages support different data-types. For example,

- C supports char, int, float, long, etc.
- Python supports String, List, Tuple, etc.

In a broad sense, there are three types of data types:

- **Fundamental data types:** These are the predefined data types which are used by the programmer directly to store only one value as per requirement, i.e., integer type, character type, or floating type. For example: int, char, float, etc.

- **Derived data types:** These data types are derived using built-in data type which are designed by the programmer to store multiple values of same type as per their requirement. For example**:** Array, Pointer, function, list, etc.

- **User-defined data types:** These data types are derived using built-in data types which are wrapped into a single a data type to store multiple values of either same type or different type or both as per the requirement. For example: Class, Structure, etc.

Data Types Supported by C++

The following table lists the data types supported by C++:

Data Type	Size	Range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	4 bytes	-2147483648 to 2147483647
unsigned int	4 bytes	0 to 4294967295
signed int	4 bytes	-2147483648 to 2147483647
short int	2 bytes	-32768 to 32767
unsigned short int	2 bytes	0 to 65,535
signed short int	2 bytes	-32768 to 32767
long int	4 bytes	-2,147,483,648 to 2,147,483,647
signed long int	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long int	4 bytes	0 to 4,294,967,295

float	4 bytes	+/- 3.4e +/- 38 (~7 digits)
double	8 bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8 bytes	+/- 1.7e +/- 308 (~15 digits)

Data Types Supported by Java

The following data types are supported by Java:

Data-Type	Size	Range
byte	1 byte	-128 to 127
char	2 bytes	0 to 65,536
short	2 bytes	-32,7688 to 32,767
int	4 bytes	-2,147,483,648 to 2,147,483,647
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	-2147483648 to 2147483647
double	8 bytes	+9.223*1018
Boolean	1 bit	True or False

Data Types Supported by Erlang

In this section, we will discuss the data types supported by Erlang, which is a functional programming language.

Number

Erlang supports two types of numeric literals, i.e. **integer** and **float**. Take a look at the following example that shows how to add two integer values:

```
-module(helloworld).
-export([start/0]).
start() ->
   io:fwrite("~w",[5+4]).
```

It will produce following output:

```
9
```

Atom

An **atom** is a string whose value can't be changed. It must begin with a lowercase letter and can contain any alphanumeric characters and special characters. When an atom contains special characters, then it should be enclosed inside single quotes ('). Take a look at the following example to understand better.

```
-module(helloworld).
-export([start/0]).
start()->
   io:fwrite(monday).
```

It will produce the following output:

```
monday
```

Note: Try changing the atom to "Monday" with capital "M". The program will produce an error.

Boolean

This data type is used to display the result as either **true** or **false**. Take a look at the following example. It shows how to compare two integers.

```
-module(helloworld).
-export([start/0]).
  start() ->
  io:fwrite(5 =< 9).
```

It will produce the following output:

```
true
```

Bit String

A bit string is used to store an area of un-typed memory. Take a look at the following example. It shows how to convert 2 bits of a bit string to a list.

```
-module(helloworld).
-export([start/0]).
start() ->
   Bin2 = <<15,25>>,
   P = binary_to_list(Bin2),
   io:fwrite("~w",[P]).
```

It will produce the following output:

```
[15,25]
```

Tuple

A tuple is a compound data type having fixed number of terms. Each term of a tuple is known as an **element**. The number of elements is the size of the tuple. The following example shows how to define a tuple of 5 terms & prints its size.

```
-module(helloworld).
-export([start/0]).
start() ->
   K = {abc,50,pqr,60,{xyz,75}} ,
   io:fwrite("~w",[tuple_size(K)]).
```

It will produce the following output:

```
5
```

Map

A map is a compound data type with a variable number of key-value associations. Each key-value association in the map is known as an **association-pair**. The **key** and **value** parts of the pair are called **elements**. The number of association-pairs is said to be the size of the map. The following example shows how to define a map of 3 mappings and print its size.

```
-module(helloworld).
-export([start/0]).
 start() ->
   Map1 = #{name=>’abc’,age=>40, gender=>’M’},
   io:fwrite("~w",[map_size(Map1)]).
```

It will produce the following output:

```
3
```

List

A list is a compound data type having variable number of terms. Each term in the list is called an element. The number of elements is said to be the length of the list. The following example shows how to define a list of 5 items and print its size.

```
-module(helloworld).
-export([start/0]).
start() ->
   List1 = [10,15,20,25,30] ,
   io:fwrite("~w",[length(List1)]).
```

It will produce the following output:

```
5
```

Note: 'String' data-type is not defined in Erlang.

11 . Functional Programming – Polymorphism

Polymorphism, in terms of programming, means reusing a single code multiple times. More specifically, it is the ability of a program to process objects differently depending on their data type or class.

Polymorphism is of two types:

- **Compile-time Polymorphism:** This type of polymorphism can be achieved using method overloading.

- **Run-time Polymorphism:** This type of polymorphism can be achieved using method overriding and virtual functions.

Advantages of Polymorphism

Polymorphism offers the following advantages:

- It helps the programmer to reuse the codes, i.e., classes once written, tested and implemented can be reused as required. Saves a lot of time.

- Single variable can be used to store multiple data types.

- Easy to debug the codes.

Polymorphic Data Types

Polymorphic data-types can be implemented using generic pointers that store a byte address only, without the type of data stored at that memory address. For example,

```
function1(void *p, void *q)
```

where **p** and **q** are generic pointers which can hold **int**, **float** (or any other) value as an argument.

Polymorphic Function in C++

The following program shows how to use polymorphic functions in C++, which is an object-oriented programming language.

```
#include <iostream>
Using namespace std:
class A
{   public:
    void show()
    {
cout<<"A class method is called/n";
```

```
   }
};
class B:public A
{ public:
  void show()
  {
cout<<"B class method is called/n";
  }
};

int main()
{
     A x;         // Base class object
     B y;         // Derived class object
     x.show();    // A class method is called
     y.show();    // B class method is called
     return 0;
}
```

It will produce the following output:

```
A class method is called
B class method is called
```

Polymorphic Function in Python

The following program shows how to use polymorphic functions in Python, which is a functional programming language.

```
class A(object):
    def show(self):
        print "A class method is called"

class B(A):
    def show(self):
        print "B class method is called"

def checkmethod(clasmethod):
    clasmethod.show()

AObj = A()
BObj = B()

checkmethod(AObj)
checkmethod(BObj)
```

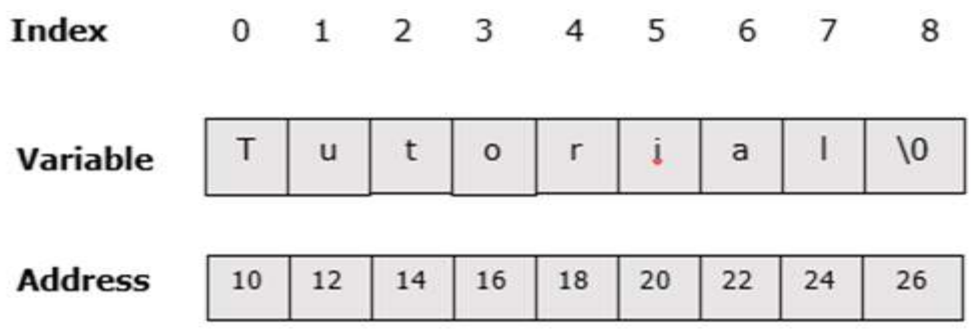
It will produce the following output:

```
A class method is called
B class method is called
```

12 . Functional Programming – Strings

A **string** is a group of characters including spaces. We can say it is a one-dimensional array of characters which is terminated by a NULL character ('\0'). A string can also be regarded as a predefined class which is supported by most of the programming languages such as C, C++, Java, PHP, Erlang, Haskell, Lisp, etc.

The following image shows how the string "Tutorial" will look in the memory.



Create a String in C++

The following program is an example that shows how to create a string in C++, which is an object-oriented programming language.

```
#include <iostream>
using namespace std;
int main ()
{
     char greeting[20] = {'H', 'o', 'l', 'i', 'd', 'a', 'y', '\0'};
     cout << "Today is: ";
     cout << greeting << endl;
     return 0;
}
```

It will produce the following output.

```
Today is: Holiday
```

String in Erlang

The following program is an example that shows how to create a string in Erlang, which is a functional programming language.

```
-module(helloworld).
-export([start/0]).

start() ->
   Str = "Today is: Holiday",
   io:fwrite("~p~n",[Str]).
```

It will produce the following output.

```
Today is: Holiday
```

String Operations in C++

Different programming languages support different methods on strings. The following table shows a few predefined string methods supported by C++.

Method Name	Description
Strcpy(s1,s2)	It copies the string s2 into string s1
Strcat(s1,s2)	It adds the string s2 at the end of s1
Strlen(s1)	It provides the length of the string s1
Strcmp(s1,s2)	It returns 0 when string s1 & s2 are same
Strchr(s1,ch)	It returns a pointer to the first occurrence of character ch in string s1
Strstr(s1,s2)	It returns a pointer to the first occurrence of string s2 in string s1

The following program shows how the above methods can be used in C++:

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
{
     char str1[20] = "Today is ";
     char str2[20] = "Monday";
```

```
      char str3[20];
      int  len ;

      strcpy( str3, str1);  // copy str1 into str3
      cout << "strcpy( str3, str1) : " << str3 << endl;

      strcat( str1, str2);  // concatenates str1 and str2
      cout << "strcat( str1, str2): " << str1 << endl;

      len = strlen(str1); // String length after concatenation
      cout << "strlen(str1) : " << len << endl;
      return 0;
}
```

It will produce the following output:

```
strcpy(str3, str1) : Today is
strcat(str1, str2): Today is Monday
strlen(str1) : 15
```

String Operations in Erlang

The following table shows a list of predefined string methods supported by Erlang.

Method Name	Description
len(s1)	It provides the string s2 into string s1
equal(s1,s2)	It returns true when string s1 & s2 are equal else return false
concat(s1,s2)	It adds string s2 at the end of string s1
str(s1,ch)	It returns index position of character ch in string s1
str (s1,s2)	It returns index position of s2 in string s1
substr(s1,s2,num)	This method returns the string s2 from the string s1 based on the starting position & number of characters from the starting position.
to_lower(s1)	This method returns string in lower case.

The following program shows how the above methods can be used in Erlang:

```
-module(helloworld).
-import(string,[concat/2]).
-export([start/0]).
   start() ->
   S1 = "Today is ",
   S2 = "Monday",
   S3 = concat(S1,S2),
   io:fwrite("~p~n",[S3]).
```

It will produce the following output:

```
"Today is Monday"
```

13 . Functional Programming – Lists

List is the most versatile data type available in functional programming languages used to store a collection of similar data items. The concept is similar to arrays in object-oriented programming. List items can be written in a square bracket separated by commas. The way to writing data into a list varies from language to language.

Program to Create a List of Numbers in Java

List is not a data type in Java/C/C++, but we have alternative ways to create a list in Java, i.e., by using **ArrayList** and **LinkedList**.

The following example shows how to create a list in Java. Here we are using a Linked List method to create a list of numbers.

```
import java.util.*;

import java.lang.*;

import java.io.*;


/* Name of the class has to be "Main" only if the class is public. */

class HelloWorld

{

     public static void main (String[] args) throws java.lang.Exception

     {

           List<String> listStrings = new LinkedList<String>();

           listStrings.add("1");

           listStrings.add("2");

           listStrings.add("3");

           listStrings.add("4");

           listStrings.add("5");


           System.out.println(listStrings);

     }

}
```

It will produce the following output:

```
[1, 2, 3, 4, 5]
```

Program to Create a List of Numbers in Erlang

```
-module(helloworld).
-export([start/0]).

start() ->
   Lst = [1,2,3,4,5],
   io:fwrite("~w~n",[Lst]).
```

It will produce the following output:

```
 [1 2 3 4 5]
```

List Operations in Java

In this section, we will discuss some operations that can be done over lists in Java.

- **Adding Elements into a List** – The methods add(Object), add(index, Object), addAll() are used to add elements into a list. For example,

  ```
  ListStrings.add(3, “three”)
  ```

- **Removing Elements from a List** – The methods remove(index) or removeobject() are used to remove elements from a list. For example:

  ```
  ListStrings.remove(3,”three”)
  ```

  Note: To remove all elements from the list clear() method is used.

- **Retrieving Elements from a List** – The get() method is used to retrieve elements from a list at a specified location. The getfirst() & getlast() methods can be used in LinkedList class. For example,

  ```
  String str= ListStrings.get(2)
  ```

- **Updating Elements in a List** – The set(index,element) method is used to update an element at a specified index with a specified element. For Example,

  ```
  listStrings.set(2,”to”)
  ```

- **Sorting Elements in a List** – The methods collection.sort() and collection.reverse() are used to sort a list in ascending or descending order. For example,

  ```
  Collection.sort(listStrings)
  ```

- **Searching Elements in a List** – The following three methods are used as per the requirement:
    - **Boolean contains(Object)** method returns **true** if the list contains the specified element, else it returns **false**.
    - **int indexOf(Object)** method returns the index of the first occurrence of a specified element in a list, else it returns -1 when the element is not found.
    - **int lastIndexOf(Object)** returns the index of the last occurrence of a specified element in a list, else it returns -1 when the element is not found.

List Operations in Erlang

In this section, we will discuss some operations that can be done over lists in Erlang.

- **Adding two lists:** The append(listfirst,listsecond) method is used to create a new list by adding two lists. For example,

```
append(list1,list2)
```

- **Deleting an element:** The delete(element,listname) method is used to delete the specified element from the list & it returns the new list. For example,

```
delete(5,list1)
```

- **Deleting last element from the list:** The droplast(listname) method is used to delete the last element from a list and return a new list. For example,

```
droplast(list1)
```

- **Searching an element:** The member(element, listname) method is used to search the element into the list, if found it returns true else it returns false. **For Example:**

```
member(5,list1)
```

- **Getting maximum and minimum value:** The max(listname) and min(listname) methods are used to find the maximum and minimum values in a list. For example,

```
max(list1)
```

- **Sorting list elements:** The methods sort(listname) and reverse(listname) are used to sort a list in ascending or descending order. For example,

```
sort(list1)
```

- **Adding list elements:** The sum(listname) method is used to add all the elements of a list and return their sum. For example,

  ```
  sum(list1)
  ```

Sort a list in ascending and descending order using Java

The following program shows how to sort a list in ascending and descending order using Java:

```
import java.util.*;

import java.lang.*;

import java.io.*;


class SortList

{

     public static void main (String[] args) throws java.lang.Exception

     {

        List<String> list1 = new ArrayList<String>();

        list1.add("5");

        list1.add("3");

        list1.add("1");

        list1.add("4");

        list1.add("2");


        System.out.println("list before sorting: " + list1);


        Collections.sort(list1);


        System.out.println("list in ascending order: " + list1);

        Collections.reverse(list1);


        System.out.println("list in dsending order: " + list1);

     }

}
```

It will produce the following output:

```
list before sorting: [5, 3, 1, 4, 2]
list in ascending order: [1, 2, 3, 4, 5]
list in dsending order: [5, 4, 3, 2, 1]
```

Sort a list in ascending order using Erlang

The following program shows how to sort a list in ascending and descending order using Erlang, which is a functional programming language:

```
-module(helloworld).
-import(lists,[sort/1]).
-export([start/0]).

start() ->
   List1=[5,3,4,2,1],
   io:fwrite("~p~n",[sort(List1)]),
```

It will produce the following output:

```
[1,2,3,4,5]
```

14 . Functional Programming – Tuple

A tuple is a compound data type having a fixed number of terms. Each term in a tuple is known as an **element**. The number of elements is the size of the tuple.

Program to define a tuple in C#

The following program shows how to define a tuple of four terms and print them using C#, which is an object-oriented programming language.

```
using System;
public class Test
{
      public static void Main()
      {
            var t1 = Tuple.Create(1, 2, 3, new Tuple<int, int>(4, 5));
            Console.WriteLine("Tuple:" + t1);
      }
}
```

It will produce the following output:

```
Tuple :(1, 2, 3, (4, 5))
```

Program to define a tuple in Erlang

The following program shows how to define a tuple of four terms and print them using Erlang, which is a functional programming language.

```
-module(helloworld).
-export([start/0]).

start() ->
   P = {1,2,3,{4,5}} ,
   io:fwrite("~w",[P]).
```

It will produce the following output:

```
{1,2,3,{4,5}}
```

Advantages of Tuple

Tuples offer the following advantages:

- Tuples are fined size in nature i.e. we can't add/delete elements to/from a tuple.
- We can search any element in a tuple.
- Tuples are faster than lists, because they have a constant set of values.
- Tuples can be used as dictionary keys, because they contain immutable values like strings, numbers, etc.

Tuples vs Lists

Tuple	List
Tuples are **immutable**, i.e., we can update its data.	List are **mutable**, i.e., we can update its data.
Elements in a tuple can be different type.	All elements in a list is of same type.
Tuples are denoted by round parenthesis around the elements.	Lists are denoted by square brackets around the elements.

Operations on Tuples

In this section, we will discuss a few operations that can be performed on a tuple.

Check whether an inserted value is a Tuple or not

The method **is_tuple(tuplevalues)** is used to determine whether an inserted value is a tuple or not. It returns **true** when an inserted value is a tuple, else it returns **false**. For example,

```
-module(helloworld).
-export([start/0]).
 start() ->
 K = {abc,50,pqr,60,{xyz,75}} ,
     io:fwrite("~w",[is_tuple(K)]).
```

It will produce the following output:

```
True
```

Converting a List to a Tuple

The method **list_to_tuple(listvalues)** converts a list to a tuple. For example,

```
-module(helloworld).
-export([start/0]).
  start() ->
  io:fwrite("~w",[list_to_tuple([1,2,3,4,5])]).
```

It will produce the following output:

```
{1,2,3,4,5}
```

Converting a Tuple to a List

The method **tuple_to_list(tuplevalues)** converts a specified tuple to list format. For example,

```
-module(helloworld).
-export([start/0]).
 start() ->
   io:fwrite("~w",[tuple_to_list({1,2,3,4,5})]).
```

It will produce the following output:

```
[1,2,3,4,5]
```

Check tuple size

The method **tuple_size(tuplename)** returns the size of a tuple. For example,

```
-module(helloworld).
-export([start/0]).
start() ->
   K = {abc,50,pqr,60,{xyz,75}} ,
   io:fwrite("~w",[tuple_size(K)]).
```

It will produce the following output:

```
5
```

15 . Functional Programming – Records

A record is a data structure for storing a fixed number of elements. It is similar to a structure in C language. At the time of compilation, its expressions are translated to tuple expressions.

How to create a record?

The keyword 'record' is used to create records specified with record name and its fields. Its syntax is as follows:

```
record(recodname, {field1, field2, . . fieldn})
```

The syntax to insert values into the record is:

```
#recordname {fieldName1 = value1, fieldName2 = value2 .. fieldNamen = valuen}
```

Program to create records using Erlang

In the following example, we have created a record of name **student** having two fields, i.e., **sname** and **sid**.

```
-module(helloworld).
-export([start/0]).
-record(student, {sname = "", sid}).

start() ->
   S = #student{sname="Sachin",sid=5}.
```

Program to create records using C++

The following example shows how to create records using C++, which is an object-oriented programming language:

```
#include<iostream>
#include<string>
using namespace std;
class student
{
public:
            string sname;
            int sid;
};
int main()
{    student S;
      S.sname = "Sachin";
      S.sid = 5;
     return 0;
}
```

Program to access record values using Erlang

The following program shows how access record values using Erlang, which is a functional programming language:

```
-module(helloworld).
-export([start/0]).
-record(student, {sname = "", sid}).

start() ->
   S = #student{sname = "Sachin",sid = 5},
   io:fwrite("~p~n",[S#student.sid]),
   io:fwrite("~p~n",[S#student.sname]).
```

It will produce the following output:

```
5
Sachin
```

Program to access record values using C++

The following program shows how to access record values using C++:

```
#include<iostream>
#include<string>
using namespace std;
class student
{
public:
            string sname;
            int sid;
};
```

```
int main()
{
student S;
      S.sname = "Sachin";
      S.sid = 5;
      cout<<S.sid<<"\n"<<S.sname;
    return 0;
}
```

It will produce the following output:

```
5
Sachin
```

The record values can be updated by changing the value to a particular field and then assigning that record to a new variable name. Take a look at the following two examples to understand how it is done using object-oriented and functional programming languages.

Program to update record values using Erlang

The following program shows how to update record values using Erlang:

```
-module(helloworld).
-export([start/0]).
-record(student, {sname = "", sid}).

start() ->
   S = #student{sname = "Sachin",sid = 5},
   S1 = S#student{sname = "Jonny"},
   io:fwrite("~p~n",[S1#student.sid]),
   io:fwrite("~p~n",[S1#student.sname]).
```

It will produce the following output:

```
5
Jonny
```

Program to update record values using C++

The following program shows how to update record values using C++:

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

class student
{
public:
      string sname;
      int sid;
};

int main()
{
      student S;
      S.sname = "Jonny";
      S.sid = 5;
      cout<<S.sname<<"\n"<<S.sid;
      cout<<"\n"<< "value after updating"<<"\n";
      S.sid=10;
      cout<<S.sname<<"\n"<<S.sid;
      return 0;
}
```

It will produce the following output:

```
Jonny
5
value after updating
Jonny
10
```

16 . Functional Programming – Lambda Calculus

Lambda calculus is a framework developed by Alonzo Church in 1930s to study computations with functions.

- **Function creation:** Church introduced the notation **λx.E** to denote a function in which 'x' is a formal argument and 'E' is the functional body. These functions can be of without names and single arguments.

- **Function application:** Church used the notation **E_1.E_2** to denote the application of function **E_1** to actual argument **E_2**. And all the functions are on single argument.

Syntax of Lambda Calculus

Lamdba calculus includes three different types of expressions, i.e.,

E ::= x	(variables)
| E_1 E_2	(function application)
| λx.E	(function creation)

Where **λx.E** is called Lambda abstraction and E is known as λ-expressions.

Evaluating Lambda Calculus

Pure lambda calculus has no built-in functions. Let us evaluate the following expression:

```
(+ (* 5 6) (* 8 3))
```

Here, we can't start with '+' because it only operates on numbers. There are two reducible expressions: (* 5 6) and (* 8 3).

We can reduce either one first. For example:

```
(+ (* 5 6) (* 8 3))
(+ 30 (* 8 3))
(+ 30 24)
= 54
```

β-reduction Rule

We need a reduction rule to handle λs:

```
(λx . * 2 x) 4
(* 2 4)
= 8
```

This is called β-reduction.

The formal parameter may be used several times:

```
(λx . + x x) 4
(+ 4 4)
= 8
```

When there are multiple terms, we can handle them as follows:

```
(λx . (λx . + (− x 1)) x 3) 9
```

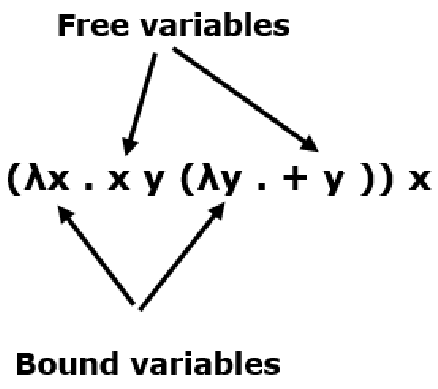
The inner **x** belongs to the inner **λ** and the outer **x** belongs to the outer one.

```
(λx . + (− x 1)) 9 3
+ (− 9 1) 3
+ 8 3
=11
```

Free and Bound Variables

In an expression, each appearance of a variable is either "free" (to λ) or "bound" (to a λ).

β-reduction of **(λx . E) y** replaces every **x** that occurs free in **E** with **y**. For Example:



Alpha Reduction

Alpha reduction is very simple and it can be done without changing the meaning of a lambda expression.

```
λx . (λx . x) (+ 1 x) ↔ α λx . (λy . y) (+ 1 x)
```

For example:

```
(λx . (λx . + (- x 1)) x 3) 9
(λx . (λy . + (- y 1)) x 3) 9
(λy . + (- y 1)) 9 3
+ (- 9 1) 3
+ 8 3
11
```

Church-Rosser Theorem

The Church-Rosser Theorem states the following:

- If E1 ↔ E2, then there exists an E such that E1 → E and E2 → E. "Reduction in any way can eventually produce the same result."
- If E1 → E2, and E2 is normal form, then there is a normal-order reduction of E1 to E2. "Normal-order reduction will always produce a normal form, if one exists."

17 . Functional Programming – Lazy Evaluation

Lazy evaluation is an evaluation strategy which holds the evaluation of an expression until its value is needed. It avoids repeated evaluation. **Haskell** is a good example of such a functional programming language whose fundamentals are based on Lazy Evaluation.

Lazy evaluation is used in Unix map functions to improve their performance by loading only required pages from the disk. No memory will be allocated for the remaining pages.

Lazy Evaluation − Advantages

- It allows the language runtime to discard sub-expressions that are not directly linked to the final result of the expression.
- It reduces the time complexity of an algorithm by discarding the temporary computations and conditionals.
- It allows the programmer to access components of data structures out-of-order after initializing them, as long as they are free from any circular dependencies.
- It is best suited for loading data which will be infrequently accessed.

Lazy Evaluation − Drawbacks

- It forces the language runtime to hold the evaluation of sub-expressions until it is required in the final result by creating **thunks** (delayed objects).
- Sometimes it increases space complexity of an algorithm.
- It is very difficult to find its performance because it contains thunks of expressions before their execution.

Lazy Evaluation using Python

The **range** method in Python follows the concept of Lazy Evaluation. It saves the execution time for larger ranges and we never require all the values at a time, so it saves memory consumption as well. Take a look at the following example.

```
r = range(10)

print(r)

range(0, 10)

print(r[3])
```

It will produce the following output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

3
```

18. Functional Programming – File I/O Operations

We need files to store the output of a program when the program terminates. Using files, we can access related information using various commands in different languages.

Here is a list of some operations that can be carried out on a file:

- Creating a new file
- Opening an existing file
- Reading file contents
- Searching data on a file
- Writing into a new file
- Updating contents to an existing file
- Deleting a file
- Closing a file

Writing into a File

To write contents into a file, we will first need to open the required file. If the specified file does not exist, then a new file will be created.

Let's see how to write contents into a file using C++.

Example

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

int main ()
{
	ofstream myfile;
	myfile.open ("Tempfile.txt", ios::out);
	myfile << "Writing Contents to file.\n";
	cout<<"Data inserted into file";
	myfile.close();
	return 0;
}
```

Note:

- **fstream** is the stream class used to control file read/write operations.
- **ofstream** is the stream class used to write contents into file.

Let's see how to write contents into a file using Erlang, which is a functional programming language.

```
-module(helloworld).
-export([start/0]).

start() ->
     {ok, File1} = file:open("Tempfile.txt", [write]),
     file:write(File1,"Writting contents to file"),
     io:fwrite("Data inserted into file\n").
```

Note:

- To open a file we have to use, **open(filename,mode)**.
- Syntax to write contents to file: **write(filemode,file_content)**

Output: When we run this code "Writing contents to file" will be written into the file **Tempfile.txt**. If the file has any existing content, then it will be overwritten.

Reading from a File

To read from a file, first we have to open the specified file in **reading mode**. If the file doesn't exist, then its respective method returns NULL.

The following program shows how to read the contents of a file **in C++**:

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

int main ()
{
string readfile;
     ifstream myfile ("Tempfile.txt",ios::in);
     if (myfile.is_open())
     {
     while ( getline (myfile,readfile) )
```

```
      {
        cout << readfile << '\n';
      }
      myfile.close();
   }
   else
      cout << "file doesn't exist";
      return 0;
}
```

It will produce the following output:

```
Writing contents to file
```

Note: In this program, we opened a text file in read mode using "ios::in" and then print its contents on the screen. We have used **while** loop to read the file contents line by line by using "getline" method.

The following program shows how to perform the same operation **using Erlang**. Here, we will use the **read_file(filename)** method to read all the contents from the specified file.

```
-module(helloworld).
-export([start/0]).

start() ->
   rdfile = file:read_file("Tempfile.txt"),
   io:fwrite("~p~n",[rdfile]).
```

It will produce the following output:

```
ok, Writing contents to file
```

Delete an Existing File

We can delete an existing file using file operations. The following program shows how to delete an existing file **using C++**:

```
#include <stdio.h>
int main ()
{
if(remove( "Tempfile.txt" ) != 0 )
    perror( "File doesn’t exist, can’t delete" );
```

```
   else
     puts( "file deleted successfully " );
   return 0;
}
```

It will produce the following output:

```
file deleted successfully
```

The following program shows how you can perform the same operation **in Erlang**. Here, we will use the method **delete(filename)** to delete an existing file.

```
-module(helloworld).
-export([start/0]).

start() ->
   file:delete("Tempfile.txt").
```

Output: If the file "Tempfile.txt" exists, then it will be deleted.

Determining the Size of a File

The following program shows how you can determine the size of a file using C++. Here, the function **fseek** sets the position indicator associated with the stream to a new position, whereas **ftell** returns the current position in the stream.

```
#include <stdio.h>
int main ()
{
     FILE * checkfile;
     long size;
     checkfile = fopen ("Tempfile.txt","rb");
     if (checkfile==NULL)
            perror ("file can’t open");
     else
     {
            fseek (checkfile, 0, SEEK_END);        // non-portable
            size=ftell (checkfile);
            fclose (checkfile);
            printf ("Size of Tempfile.txt: %ld bytes.\n",size);
```

```
      }
return 0;
}
```

Output: If the file "Tempfile.txt" exists, then it will show its size in bytes.

The following program shows how you can perform the same operation in Erlang. Here, we will use the method **file_size(filename)** to determine the size of the file.

```
-module(helloworld).
-export([start/0]).


start() ->
   io:fwrite("~w~n",[filelib:file_size("Tempfile.txt")]).
```

Output: If the file "Tempfile.txt" exists, then it will show its size in bytes. Else, it will display "0".