|  |
| --- |
| **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES**  **CS 201–DATA STRUCTURES LAB**  **Lab Session 05** |
| **Instructors:** Mr. Faizan Yousuf, Ms. Safia, Ms. Maham Mobin |

Outline

* Functions
* Recursion
* Direct and Indirect Recursion
* Tail Recursion
* Backtracking
* Rat-Maze Example
* Excessive Recursion
* Pointers to a Function
* Task

**FUNCTION**

A function is a group of statements that together perform a particular task. Every C++ program has at least one function and that is a main function.

Based on the nature of task, we can divide up our program into several functions.

***What are local variables?***

*These are the variables that are declared in the function. Their lifetime ends when the execution of the function finishes and are only known in the function in which they are declared.*

## Function returns a value

Value returning functions are used when only one result is returned and that result is used directly in an expression.

***General Format of a function return a value***

***datatype nameOfFunction()***

***{***

***return variable;***

***}***

## Void Function

Void functions are used when function doesn’t return a value.

***General Format of a void function***

***void nameOfFunction()***

***{***

***Statement1; Statement2;***

***…***

***Statement n;***

***}***

**RECURSION**

# When a function repeatedly calls itself, it is called a recursive function and the process is called recursion.

It seems like a never ending loop, or more formally it seems like our function will never finish. In some cases, this might true, but in practice we can check if a certain condition becomes true then return from the function.

***Base Case***

***The case/condition in which we end our recursion is called a base case.***

***Example of finite recursion***

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

***void myFunction( int counter)***

***{***

***if(counter == 0)***

***return;***

***else***

***{***

***cout <<counter<<endl; myFunction(--counter); return;***

***}***

***}***

***int main()***

***{***

***myFunction(10);***

***}***

## Characteristics of Recursion

Every recursion should have the following characteristics.

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

***General Format***

returntype recursive\_func ([argument list])

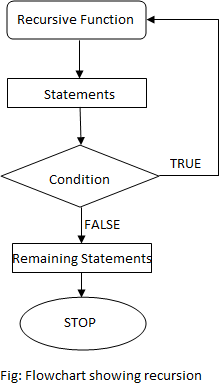
{

statements;

recursive\_func ([actual argument])

}

## Flow chart for Recursion



**DIRECT Vs INDIRECT RECURSION**

There are two types of recursion, direct recursion and indirect recursion.

## Direct Recursion

A function when it calls itself directly is known as Direct Recursion.

***Example of Direct Recursion***

#include<iostream> using namespace std; int factorial (int n)

{

if (n==1 || n==0) return 1;

else

return n\*factorial(n-1);

}

int main()

{

int f = factorial(5); cout << f;

}

## Indirect Recursion

A function is said to be indirect recursive if it calls another function and the new function calls the first calling function again.

***Example of Indirect Recursion***

#include<iostream> using namespace std; int func1(int);

int func2(int); int func1(int n)

{

if (n<=1) return 1;

else

return func2(n);

}

int func2(int n)

{

return func1(n-1);

}

int main()

{

int f = func1(5); cout << f;

}

Here, recursion takes place in 2 steps, unlike direct recursion.

* First, func1 calls func2
* Then, func2 calls back the first calling function func1.

## Disadvantages of Recursion

* Recursive programs are generally slower than non recursive programs. This is because, recursive function needs to store the previous function call addresses for the correct program jump to take place.
* Requires more memory to hold intermediate states. It is because, recursive program requires the allocation of a new stack frame and each state needs to be placed into the stack frame, unlike non-recursive(iterative) programs.

**Tail Recursion**

# What is tail recursion?

A recursive function is tail recursive when recursive call is the last thing executed by the function. For example the following C++ function print() is tail recursive.

// An example of tail recursive function void print(int n)

{

if (n < 0) return; cout << " " << n;

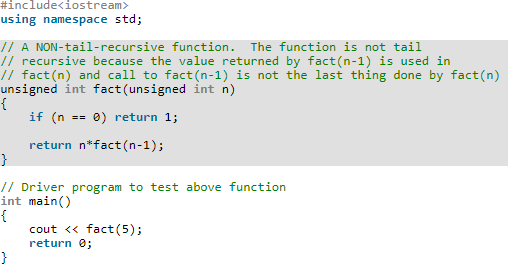
// The last executed statement is recursive call print(n-1);

}

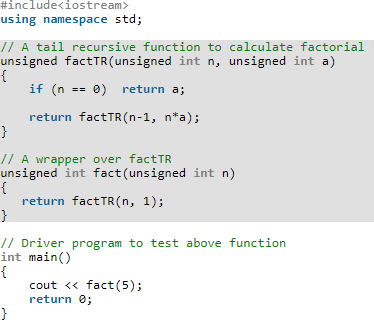
# Why do we care?

The tail recursive functions considered better than non tail recursive functions as tail- recursion can be optimized by compiler. The idea used by compilers to optimize tail- recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function’s stack frame is of no use.

Can a non-tail recursive function be written as tail-recursive to optimize it? Consider the following function to calculate factorial of n. It is a non-tail-recursive function. Although it looks like a tail recursive at first look. If we take a closer look, we can see that the value returned by fact(n-1) is used in fact(n), so the call to fact(n-1) is not the last thing done by fact(n)



The above function can be written as a tail recursive function. The idea is to use one more argument and accumulate the factorial value in second argument. When n reaches 0, return the accumulated value.



**Backtracking**

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

According to the wiki definition,

***Backtracking*** can be defined as a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem.

**Pseudo Code for Backtracking** : Recursive backtracking solution.

void findSolutions(n, other params) : if (found a solution) :

solutionsFound = solutionsFound + 1; displaySolution();

if (solutionsFound >= solutionTarget) : System.exit(0);

return

for (val = first to last) : if (isValid(val, n)) :

applyValue(val, n); findSolutions(n+1, other params); removeValue(val, n);

Finding whether a solution exists or not

boolean findSolutions(n, other params) : if (found a solution) :

displaySolution(); return true;

for (val = first to last) : if (isValid(val, n)) :

applyValue(val, n);

if (findSolutions(n+1, other params)) return true;

removeValue(val, n);

return false;

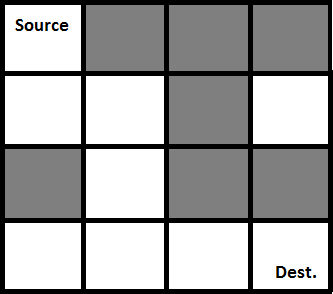
**Rat in a Maze Problem**

A Maze is given as N\*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N- 1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Following is an example maze.

Gray blocks are dead ends (value = 0).



Following is binary matrix representation of the above maze.

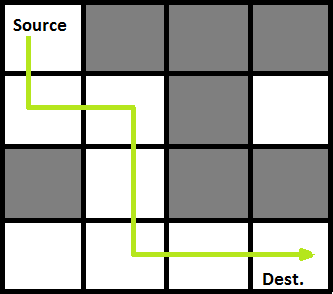
{1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

Following is a maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrx.

{1, 0, 0, 0}

{1, 1, 0, 0}

{0, 1, 0, 0}

{0, 1, 1, 1}

All enteries in solution path are marked as 1.

**Naive Algorithm**

The Naive Algorithm is to generate all paths from source to destination and one by one check if the generated path satisfies the constraints.

while there are untried paths

{

generate the next path

if this path has all blocks as 1

{

print this path;

}

}

# Backtracking Algorithm

If destination is reached print the solution matrix

Else

1. Mark current cell in solution matrix as 1.
2. Move forward in the horizontal direction and recursively check if this move leads to a solution.
3. If the move chosen in the above step doesn't lead to a solution then move down and check if this move leads to a solution.
4. If none of the above solutions works then unmark this cell as 0 (BACKTRACK) and return false.

**Excessive Recursion**

Some recursive methods repeat the computations for some parameters, which results in long computation time even for simple cases

//This can be implemented as follows: int fib(int n){

if (n<2){

return n;

}

else{

}

}

return fib(n-2)+fib(n-1);

**POINTER TO FUNCTION OR FUNCTION POINTER**

The function pointer is actually a variable which points to the address of a function.

Function Pointers provide an extremely interesting, efficient and elegant programming technique. You can use them to replace switch/if-statements, and to realize late-binding.

## Late binding refers to deciding the proper function during runtime instead of compile time.

Unfortunately, function pointers have complicated syntax and therefore are not widely used. If at all, they are addressed quite briefly and superficially in textbooks. They are less error prone than normal pointers because you will never allocate or deallocate memory with them.

## Define a function pointer

Since a function pointer is nothing else than a variable, it must be defined as usual.

***General Syntax***

Return\_type (\*name\_of\_pointer\_variable) (data type of arguments separated by comma) = NULL;

In the following example, we define a function pointer named function\_ptr. It points to a function, which takes two integersand returns a float value.

***Example – Declare function pointer***

float (function\_pointer) (int, int) = NULL;

## Assign an address to function pointer

It’s quite easy to assign the address of a function to a function pointer. You simply take the name of a suitable and known function or member function. Although it’s optional for most compilers you should use the address operator & infront of the function’s name in order to write portable code.

***Example***

float division(int a, int b)

{

return a/b;

}

function\_ptr = division; //short form

function\_ptr = &division; //assignment using address operator

}

int main()

{

float (\*func\_ptr)(float,float)=NULL;

func\_ptr = &division;

float result = (\*func\_ptr)(9, 5); cout<<result <<endl;

func\_ptr = division;

result = (\*func\_ptr)(9, 5); cout<<result <<endl;

return 0;

}

## Pass function pointer as an argument

You can pass a function pointer as a function’s calling argument. You need this for example if you want to pass a pointer to a callback function. The following code shows how to pass a pointer to a function which returns an int and takes a float and two char.

***Example***

#include<iostream> using namespace std;

int DoIt (float a, char b, char c)

{

printf("DoIt\n"); return a+b+c;

}

void PassPtr(int (\*pt2Func)(float, char, char))

{

int result = (\*pt2Func)(12, 'a', 'b');

// call using function pointer printf("%d", result);

}

void Pass\_A\_Function\_Pointer()

{

printf("Executing ’Pass\_A\_Function\_Pointer’\n"); PassPtr(&DoIt);

}

int main()

{

Pass\_A\_Function\_Pointer();

}