NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CL103 – COMPUTER PROGRAMMING LAB

Instructors: Mr. Basit Jasani, Mr. Zain-ul-Hassan, Mr. Mukhtiar
Basit.jasani@nu.edu.pk | zain.hassan@nu.edu.pk

Instructors: Ms. Solat Jabeen, Ms. Neelam Shah
Solat.jabeen@nu.edu.pk | neelam.shah@nu.edu.pk

LAB MANUAL 04

Outline

- ✓ Pointer to function OR Function Pointer
- ✓ Pointer and multidimensional array
- ✓ Dynamic Memory Management
- ✓ What is a function?
- ✓ Void function
- ✓ The Function returns a value
- ✓ What is recursion?
- ✓ Direct Vs Indirect Recursion
- ✓ Exercise

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)
{
        Cout << "DoIt\n";
        return a+b+c;
}
void PassPtr(int (*pt2Func)(float, char, char))
{
        int result = (*pt2Func)(12, 'a', 'b');
        // call using function pointer cout << result;
}
void Pass_A_Function_Pointer()
{
        Cout << “Executing ’Pass_A_Function_Pointer’\n";
        PassPtr(&DoIt);
}
int main()
{
        Pass_A_Function_Pointer();
}
```

MEMORY ALLOCATION

There are two ways through which memory can be allocated to a variable. They are ***static allocation*** of memory and ***dynamic memory allocation***. So far, we have declared variable and array statically. This means at compile time memory is allocated to variable or array.

Example of static memory allocation

int a = 10;

Dynamic Memory Allocation

Dynamic memory allocation is needed if we want a variable amount of memory that can only be determined during runtime.
When we talk about dynamic memory allocation or runtime memory allocation, two keywords are important in ***C++.*** They are ***new*** and ***delete***.

- ***new and new[ ] operator***

new operator is used to dynamically allocate memory. When we allocate memory using new keyword a pointer is needed. This is so because new allocates memory and dynamically and it returns address to allocated memory and this returned memory address is stored in a pointer variable.

dynamically memory allocation for a variable

General Synatx for dynamically memory allocation for a variable

Pointer = new type;

Example

*int *ptr;*
ptr = new int;// dynamically memory allocated for an integer and address of allocated memory is stored in ptr

Accessing value stored at the address in pointer variable (single variable)

Value at address stored in pointer variable will be accessed by dereference '*' operator as it is normally done.

Example

```
int *ptr;
ptr = new int;
*ptr = 10; // dereferencing pointer
```

dynamically memory allocation for an array

General Synatx for dynamically memory allocation for an array

Pointer = new type[size];

Example

*int *ptr;*
ptr = new int[10];// dynamically memory allocated for an integer array of size 10 and address of first element of array is stored in ptr

Accessing value stored at the address in pointer variable (array)

This can be done in two ways.

1. Use ***pointer subscript notation***. i.e ptr[0] = 10, ptr[1] = 20 etc.
   OR
2. Use ***deferencing operator.*** i.e *(ptr+0) = 10, *(ptr+1) = 20;

- ***delete and delete[ ] operator***

To free dynamic memory after it is used, **delete** operator is used so that the memory becomes available again for other requests of dynamic memory.

```
#include<iostream>
using namespace std;
int main()
{
	system("color 70");
	int i,n;
	int *p;
	cout<<"How many numbers would you like to
type? ";
	cin>>i;
	p = new int[i];
	for(n=0;n<i;n++)
	{
		cout<<"Enter Number: ";
		cin>>p[n];
	}
	cout<<endl<<"You have entered: ";
	for(n=0;n<i;n++)
	{
		cout<<p[n]<<",";

	}
	cout<<endl<<endl;

	delete[] p;

	cout<<"After deallocation, You have entered: ";
	for(n=0;n<i;n++)
	{
		cout<<p[n]<<",";
	}
}
```

```
How many numbers would you like to type? 5
Enter Number: 1
Enter Number: 2
Enter Number: 3
Enter Number: 4
Enter Number: 5

You have entered: 1,2,3,4,5,

After deallocation, You have entered: 9198608,0,9175384,0,5,
--------------------------------
Process exited after 6.403 seconds with return value 0
Press any key to continue . . .
```

FUNCTION

A function is a group of statements that together perform a particular task. Every C++ program has atleast one function and that is a main function.
Based on the nature of task, we can divide up our program into several fuctions.

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 repeadtedly 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

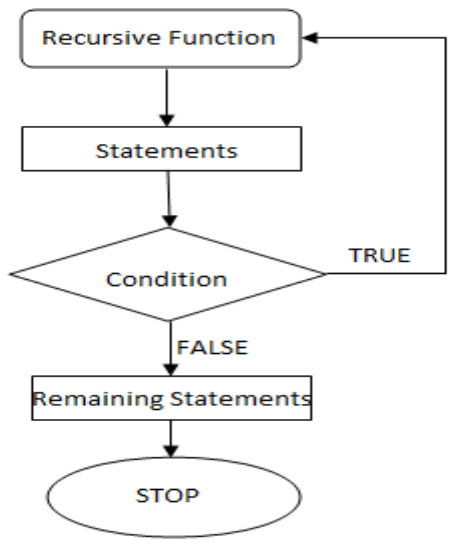

Fig: Flowchart showing recursion

DIRECT Vs INDIRECT RECURSION

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

1. ***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;
}
```

2. *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.