|  |
| --- |
| **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES**  **CL 103 - COMPUTER PROGRAMMING LAB**  **Instructors:** Mr. Basit Ali, Ms. Mahrukh Khan, Ms. Ammara Yaseen, Ms. Tooba Ali, Ms. Maham Mobin  **Email:** [basit.jasani@nu.edu.pk](mailto:basit.jasani@nu.edu.pk) , [mahrukh.khan@nu.edu.pk](mailto:mahrukh.khan@nu.edu.pk), [ammara.yaseen@nu.edu.pk](mailto:ammara.yaseen@nu.edu.pk), [tooba.ali@nu.edu.pk](mailto:tooba.ali@nu.edu.pk), [maham.mobin@nu.edu.pk](mailto:maham.mobin@nu.edu.pk) |

**Lab # 02**

# Outline

* Pointers
* Pointers to array
* Double Pointers
* Dynamic Memory Management (new and delete operators)
* 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

**POINTERS**

Pointer is a variable whose value is a memory address. Normally, a variable directly contains a specific value. A pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name directly references a value, and a pointer indirectly references a value.

**REFERENCE OPERATOR ( & )**

The address that locates a variable within memory is what we call a reference to that variable.

ted = &andy;

assume that andy is placed during runtime in the memory address 1776.

andy = 25;

fred = andy;

ted = &andy;

The values contained in each variable after the execution of this, are shown in the following diagram:



**DE-REFERENCE OPERATOR ( \* )**

The asterisk (\*) acts as a dereference operator and that can be translated to "value pointed by".

beth = \*ted;

*We could read as: "beth equal to value pointed by ted", beth would take the value 25, since ted is 1776, andthe value pointed by 1776 is 25.*



**POINTER TYPE DECLARATION**

**SYNTAX:** type \* variable ;

**EXAMPLE:** int \*ptr, char \* character, float \* greatvalue;

**EXPLANATION:** The value of the pointer variable ***ptr*** is a memory address. A data item whose address is stored in this variable must be of the specified type.



**POINTERS AND ARRAYS**

The identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to.

**EXAMPLE:** *Assume the following declaration:*

int numbers [20];

int \* p;

The following assignment operation would be valid:

p = numbers;



**POINTER INITIALIZATION**

When declaring pointers we can explicitly specify which variable we want them to point to.

**EXAMPLE:** int number;

int \*tommy = &number;

The above code is equivalent to:

int number;

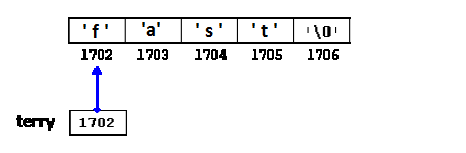
int \*tommy;

tommy = &number;

We can also initialize the content at which the pointer points with constants at the same moment the pointer is declared:

**EXAMPLE:** char \* terry = "fast";

Assuming that "fast" is stored at the memory locations that start at addresses1702, we can represent the previous declaration as:



**POINTER ARITHMETIC**

Only addition and subtraction operations are allowed with pointers. Furthermore, when incrementing a pointer, the size in bytes of the type pointed to is added to the pointer.

**EXAMPLE:** char \*mychar; //points to memory location 1000

short \*myshort; //points to memory location 2000

long \*mylong; //points to memory location 3000

mychar++; //mychar now contains 1001

myshort++; //myshort now contains 2002

mylong++; //mylong now contains 3004

**POINTERS TO POINTERS (DOUBLE POINTER)**

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other

pointers). In order to do that, we only need to add an asterisk (\*) for each level of reference in their declarations:

**EXAMPLE:** char a;

char \* b;

char \*\* c;

a = 'z';

b = &a;

c = &b;

This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could berepresented as:



* c has type char\*\* and a value of 8092
* \*c has type char\* and a value of 7230
* \*\*c has type char and a value of 'z'

**VOID POINTER**

Void pointersare pointers that can point to any data type. One limitation of void pointers is that they cannot be directly de-referenced.

**EXAMPLE:**

****

**NULL POINTER**

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointingto any valid reference or memory address.

**EXAMPLE:** int \* p;

p = 0; // p has a null pointer value

Do not confuse null pointers with void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to some where without a specific type.

**POINTERS TO FUNCTIONS**

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function, since these cannot be passed de-referenced.

**EXAMPLE:**



**DYNAMIC MEMORY ALLOCATION**

**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 examples***

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

1. 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]<<",";**  **}**  **}** |  |

**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();  } |

**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 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);*  *}* |

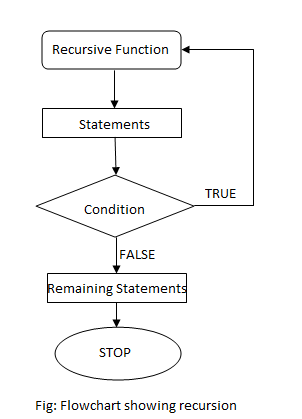
***Characteristics of Recursion***

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.

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;*  *}* |

1. ***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.

**EXERCISE**

//draws a rectangle using functions

#include <stdio.h>

#include <iostream>

using namespace std;

void draw\_solid\_line(int size);

void draw\_hollow\_line(int size);

void draw\_rectangle(int len, int wide);

int main(void) {

int length, width;

cout<<"Enter length and width of rectangle >";

cin>>length;

cin>>length>>width;

draw\_rectangle(length, width);

system("pause");

return 0;

}

void draw\_solid\_line(int size) {

//Write your Code Here

}

void draw\_hollow\_line(int size) {

//Write your Code Here

}

void draw\_rectangle(int len, int wide) {

int i;

draw\_solid\_line(wide);

if (len > 2) {

for (i=1; i<=len - 2; i++)

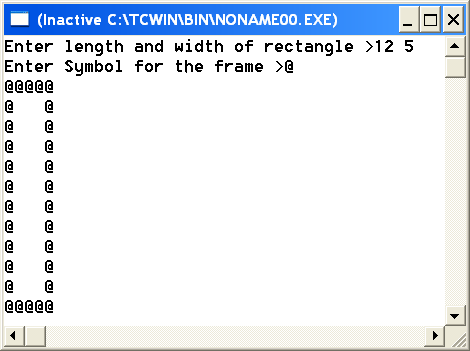
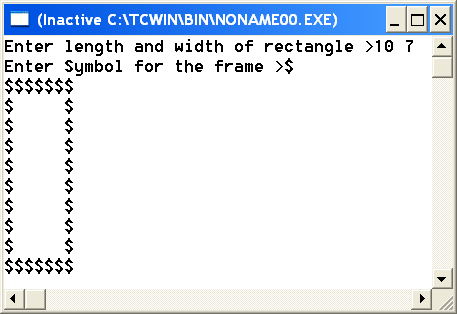
draw\_hollow\_line(wide);

}

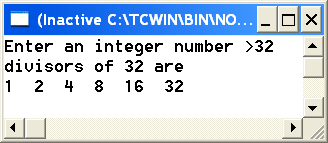
draw\_solid\_line(wide);

}

1. Modify above code to get following output. (Optional Question)



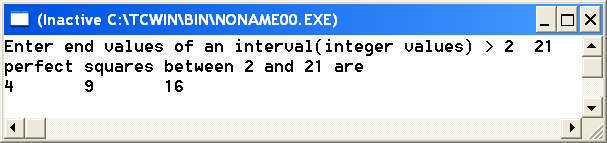
2. Write function divisors that receive an integer number and return its divisors using pointer array on main including 1 and itself.



3. Write a logical function perfect\_Square that receives a positive integer number and checks if it is a perfect square or not. (Optional Question)

Note: perfect square numbers are 4, 9,16,25,36 etc….

Write a main function that makes use of the perfect\_Square function to find and print all perfect squares between n1 and n2. n1 and n2 are end values of a range introduced by the user.



4. Write a logical function, is\_prime, that takes an integer number and determines if the number is prime or not.

Note: A prime number is one that does not have proper factors.

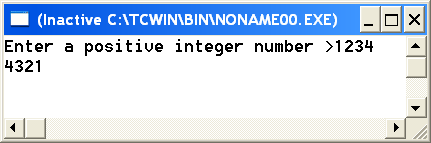
Write a main function that makes use of the is\_prime function to find and print all the prime numbers from 2 to 100.

5. Write a program that computes the area and perimeter of a rectangle using 2 functions. One function is used to read the width and length, and the other to compute the area and perimeter. Write a main function to test your functions. (Optional Question)

6. Write a function that receives a time in seconds and returns the equivalent time in hours, minutes, and seconds. Write a main function to test your function. For example if the received time is 4000 seconds, the function returns 1 hour, 6 minutes, and 40 seconds.

Use integer division and remainder.

7. Write a recursive function that receives a positive integer number and prints it on the screen in reverse order as shown below. Write a main function for testing. (Optional Question)



8. Write a recursive function countdigits that receives an integer number and returns how many digits it contains. Write a main function to read an integer number and tests the written function countdigits.

Input: 12345

Output: 5 Digits

9. Write a function that reverses the elements of an array. In other words, the last element must become the first; the second from last must become the second and so on. The function must accept only one pointer value and return void. (Optional Question)

10. Write a program that will read 10 integers from the keyboard and place them in an array.

The program then will sort the array into ascending and descending order and print the sorted list. The program must not change the original array or create any other integer arrays.

Hint: It requires two pointer arrays.