Fundamentals of Programming in C++
CoSc 2031

Chapter 3

Pointer

Mekonnen K.
(MSc)

Email: mekonnen307@gmail.com

Objectives:

- Pointer Data Type and Pointer Variables
- Pointer Declaration
- Pointer Operators
- Initializing Pointer Variables
- Operations, Expressions and Arithmetic on Pointer Variables
- Strings and pointers
- Relationship between pointers & arrays
- Functions and Pointers
- Dynamic memory management

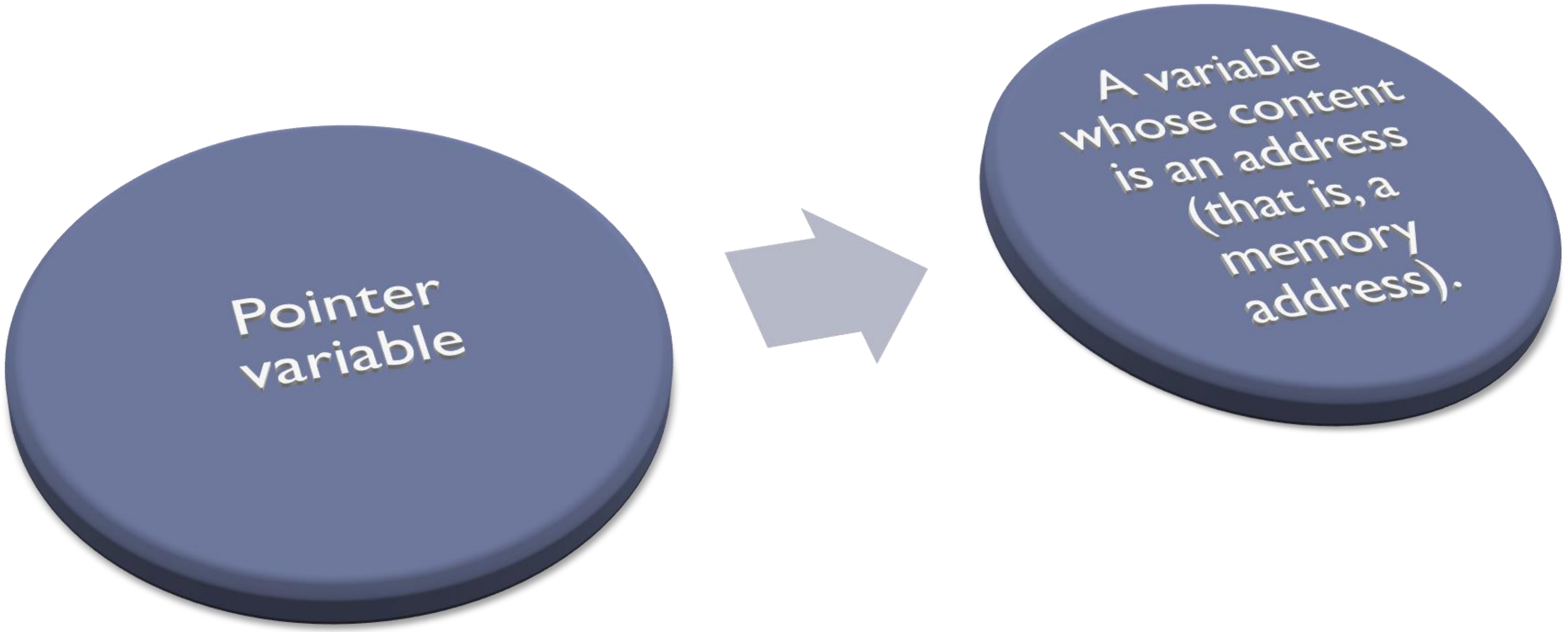
What is Pointer

- A pointer is a variable that stores a memory address. Pointers are used to store the addresses of other variables or memory items. Pointers are very useful for another type of parameter passing, usually referred to as **Pass By Address**. Pointers are essential for dynamic memory allocation.

Pointer Data Type and Pointer Variables

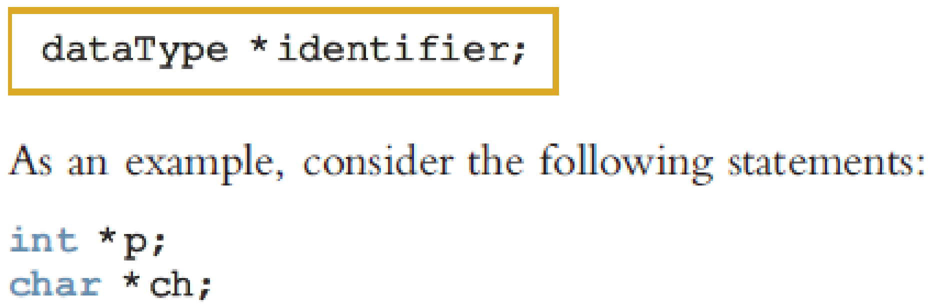
- C++'s data types are classified into three categories:
  - Simple
  - Structured
  - Pointers.

What is the Pointer Value


Pointer variable
A variable whose content is an address (that is, a memory address).

Pointer Data Type and Pointer Variables

- There is no name associated with pointer data types
- If no name is associated with a pointer data type, how do you declare pointer variables?



```
dataType *identifier;
```

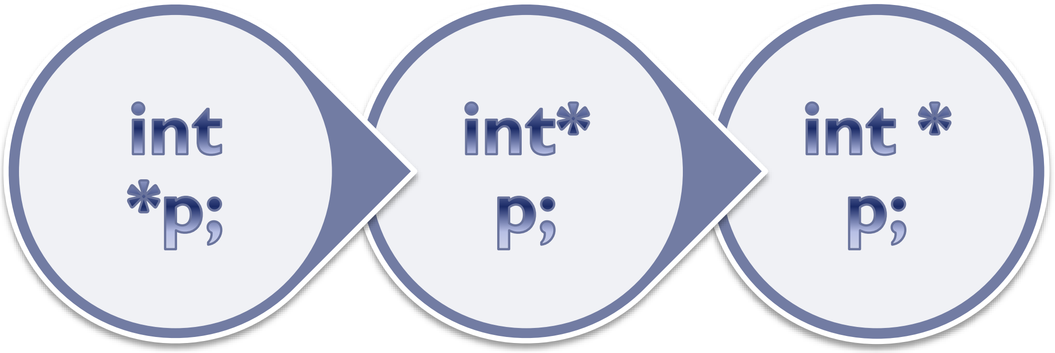
As an example, consider the following statements:

```
int *p;
char *ch;
```

- p is called a pointer variable of type int
- ch is called a pointer variable of type char.

Pointer Declaration

- Thus, the character * can appear anywhere between the data type name and the variable name.



Pointer Declaration

- int* p, q;
- In this statement:
  - only p is the pointer variable, not q.
  - Here, q is an int variable.
- we prefer to attach the character * to the variable name. So the preceding statement is written as:

```
int *p, q;
```

Pointer Operators

- C++ provides two operators operator to work with pointers.
  - (&) the address of operator
  - (*) the dereferencing

Address of Operator (&)

- is a **unary** operator that returns the address of its operand.
- For example, given the statements:

```
int x;
int *p;
```

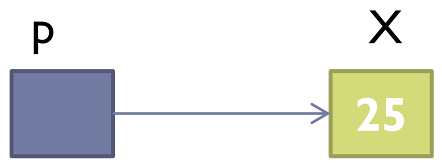
- The statement:

```
p = &x;
```

- assigns the address of x to p. That is, x and the value of p refer to the same memory location

Dereferencing Operator (*)

- referred to as **indirection operator**
- refers to the object to which its operand (that is, the pointer) points.
- For example, given the statements:



```
int x = 25;
int *p;
p = &x;    //store the address of x in p
```

Dereferencing Operator (*)

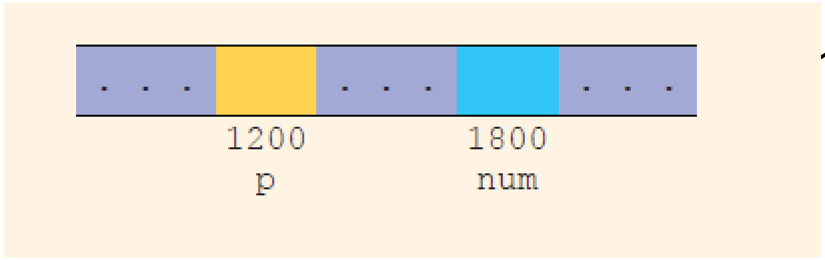
- Let us consider the following statements:

```
int *p;
int num;
```

In these statements:

- **p** is a pointer variable of type **int**
- **num** is a variable of type **int**.

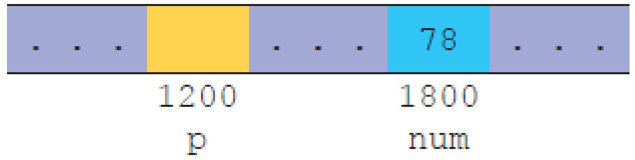
- Let us assume that memory location 1200 is allocated for p, and memory um.

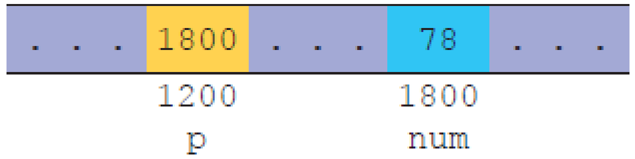


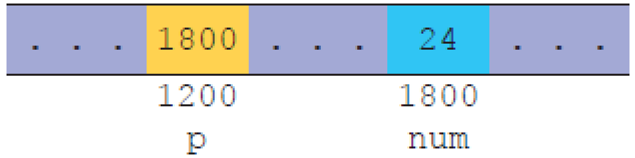
Dereferencing Operator (*)

- Consider the following statements.

1. num = 78;
2. p = &num;
3. *p = 24;

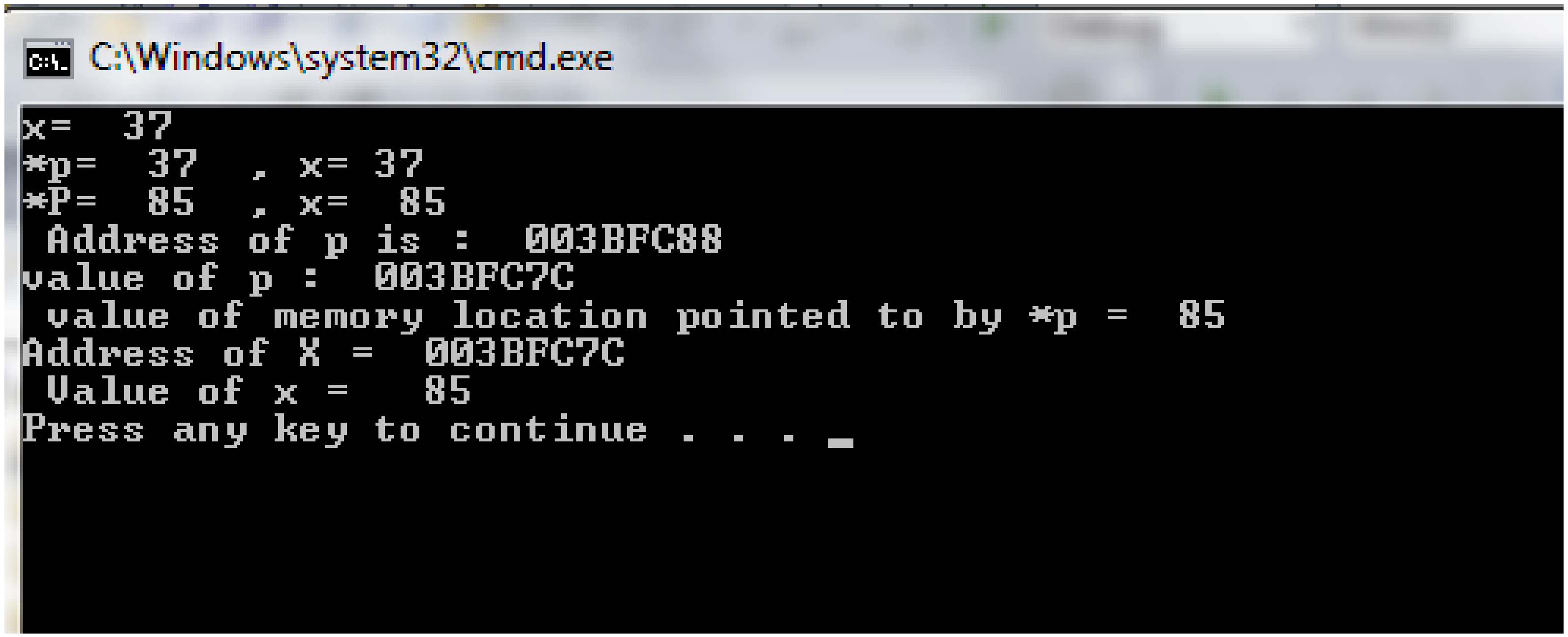






Example

```
#include <iostream>
#include <string>
using namespace std;
//================================
int main() {
      int *p;
      int x = 37;
      cout <<"x=  " << x<<endl;
      p=&x;
      cout <<"*p=  "<< *p <<"  , x= "<<x<<endl;
      *p=85;
      cout <<"*P=  " <<*p <<"  , x=  "<<x<<endl;
      cout <<" Address of p is :  "<< &p <<endl;
      cout<<"value of p :  "<< p<<endl;
      cout<< " value of memory location pointed to by *p =  "<<*p<<endl;
      cout<<"Address of X =  " << &x<<endl;
      cout <<" Value of x =   " << x<<endl;
      return 0;
}
```



```
x=  37
*p=  37  , x= 37
*P=  85  , x=  85
 Address of p is :  003BFC88
value of p :  003BFC7C
 value of memory location pointed to by *p =  85
Address of X =  003BFC7C
 Value of x =   85
Press any key to continue . . . _
```

Summary:

```
int    v;          //defines    variable    v of type int
int*   p;          //defines    p as a pointer to int
p = &v;            //assigns    address of variable v to pointer p
v = 3;             //assigns    3 to v
*p = 3;            //also    assigns    3 to v
```

Initializing Pointer Variables

- C++ does not automatically initialize variables
- pointer variables must be initialized if you do not want them to point to anything.
- Pointer variables are initialized using the following two statements :
  - p = NULL;
  - p = 0 ;
- The number 0 is the only number that can be directly assigned to a pointer variable.

Operations on Pointer Variables

- There are four arithmetic operators that can be used on pointers:

Increment	++
Decrement	--
Addition	+
Subtraction	-

- The following operation can be performed on pointers.
  - We can add integer value to a pointer.
  - We can subtract an integer value from a pointer.
  - We can compare two pointers, if they point the elements of the same array.
  - We can subtract one pointer from another pointer if both point to the same array.
  - We can assign one pointer to another pointer provided both are of same type.

Operations on Pointer Variables

- The following operations cannot be performed on pointers
  - Addition of two pointers.
  - Subtraction of one pointer from another pointer when they do not point to the same array.
  - Multiplication of two pointers.
  - Division of two pointers.

Operations on Pointer Variables

For example, suppose that we have the following statements:

```
int *p, *q;
```

The statement:

```
p = q;
```

- copies the value of q into p. After this statement executes, both p and q point to the same memory location.
- Any changes made to *p automatically change the value of *q, and vice versa.

Comparison

The expression:

- p == q

evaluates to true if p and q have the same value—that is, if they point to the same memory location.

Similarly, the expression:

- p != q

evaluates to true if p and q point to different memory locations.

Decrement and increment

- ++ increments the value of a pointer variable by the size of the memory to which it is pointing.
- Similarly, **--** the value of a pointer variable by the size of the memory to which it is pointing.
- to explain the increment and decrement operations on pointer variables:

```
int *p;
double *q;
char *chPtr;
studentType *stdPtr;   //studentType is as defined before
```

Recall that the size of the memory allocated for an

- int variable is 4 bytes
- a double variable is 8 bytes
- a char variable is 1 byte.
- studentType is 40 bytes.

Decrement and increment

The statement:

- p++; or p = p + 1;

  increments the value of p by 4 bytes because p is a pointer of type int.

Similarly, the statements:

- q++;
- chPtr++;

  increment the value of q by 8 bytes and the value of chPtr by 1 byte, respectively.

The statement:

- stdPtr++;

  increments the value of stdPtr by 40 bytes.

Cont.

- Moreover, the statement:
- p = p + 2;
- increments the value of p by 8 bytes.
- Thus, when an integer is added to a pointer variable, the value of the pointer variable is incremented by the integer times the size of the memory that the pointer is pointing to.
- Similarly, when an integer is subtracted from a pointer variable, the value of the pointer variable is decremented by the integer times the size of the memory to which the pointer is pointing.

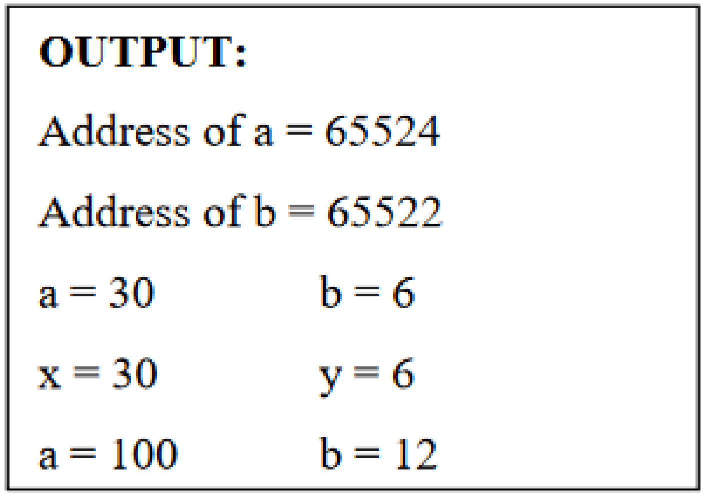
Notes

- Pointer arithmetic can be very dangerous.
- The program can accidentally access the memory locations of other variables and change their content without warning. leaving the programmer trying to find out what went wrong.

Example.

- A program to illustrate the pointer expression and pointer arithmetic.

```
#include<iostream.h>
void main( )
{
        int a, b, x, y;
        int *ptr1, *ptr2;
        a = 30;
        b = 6;
        ptr1 = &a
        ptr2 = &b;
        x = *ptr1 + *ptr2 – 6;
        y = 6 - *ptr1 / *ptr2 + 30;
        cout<<”Address of a = “<<ptr1<<endl;
        cout<<”Address of b = “<<ptr2<<endl;
        cout<<”a = “<<a<<”b = “<<b<<endl;
        cout<<”x = “<<x<<”y = “<<y<<endl;
        *ptr1 = *ptr1 + 70;
        *ptr2 = *ptr2 * 2
        cout<<”a = “<<a<<”b = “<<b<<endl;
}
```



```
OUTPUT:
Address of a = 65524
Address of b = 65522
a = 30          b = 6
x = 30          y = 6
a = 100         b = 12
```

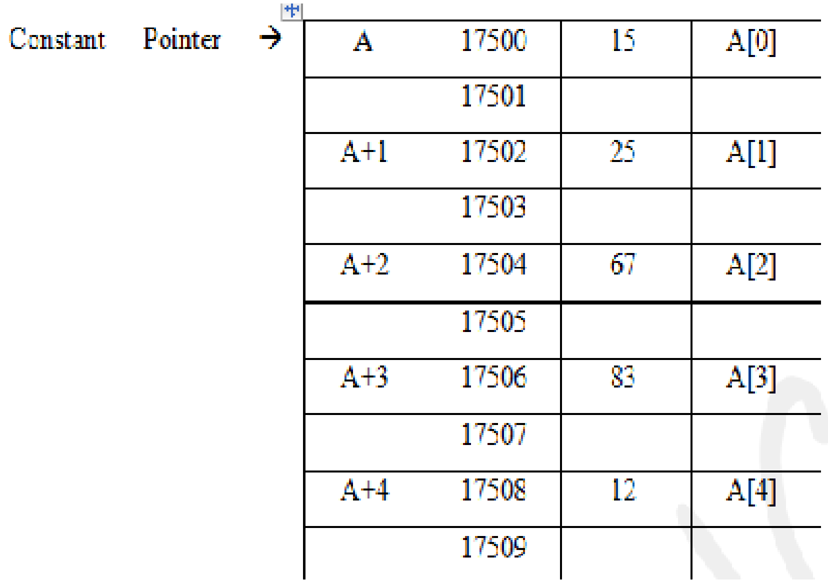
Pointers and arrays

- There is a close relationship between array and pointers in C++.
- Consider the following program which prints the elements of an array A.
-
```
#include<iostream.h>
void main( )
{
        int A[5] = { 15, 25, 67, 83, 12};
        for (int i = 0; i<5; i++)
        cout<<A[i]<<"\t";
}
```
- Output of the above program is: **15 25 67 83 12**

Pointers and arrays

- When we declare an array, its name is treated as a constant pointer to the first element of the array.
- This is also known as the **base address of the array**.
- In other words base address is the address of the first element in the array of the address of a[0].
- If we use constant pointer to print array elements.

-
  ```
  #include<iostream.h>
  void main( )
  {
          int A[5] = { 15, 25, 67, 83, 12};
          cout<< *(A) <<"\t";
          cout<< *(A+1) <<"\t";
          cout<< *(A+2) <<"\t";
          cout<< *(A+3) <<"\t";
          cout<< *(A+4) <<"\t";
  }
  ```



Constant Pointer →

A	17500	15	A[0]
	17501		
A+1	17502	25	A[1]
	17503		
A+2	17504	67	A[2]
	17505		
A+3	17506	83	A[3]
	17507		
A+4	17508	12	A[4]
	17509		

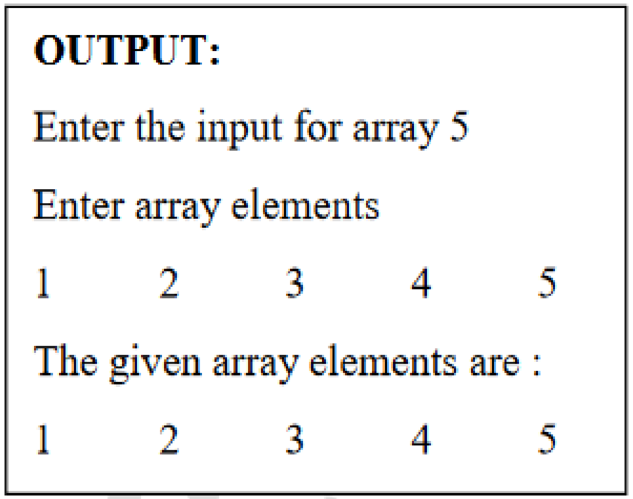
- Output of the above program is: **15 25 67 83 12**

Pointers and arrays

- Here the expression *(A+3) has exactly same effect as A[3] in the program.
- The difference between constant pointer and the pointer variable is that the constant pointer cannot be incremented or changed while the pointer to an array which carries the address of the first element of the array may be incremented.

Pointers and arrays

- The following example shows the relationship between pointer and one dimensional array.
-
```
#include<iostream.h>
void main( )
{
        int a[10], i, n;
        cout<<”Enter the input for array”;
        cin>>n;
        cout<<”Enter array elements:”;
        for(i=0; i<n; i++)
        cin>>*(a+i);
        cout<<The given array elements are :”;
        for(i=0; i<n; i++)
        cout<<”\t”<<*(a+i);
}
```



OUTPUT:

```
Enter the input for array 5
Enter array elements
1       2       3       4       5
The given array elements are :
1       2       3       4       5
```

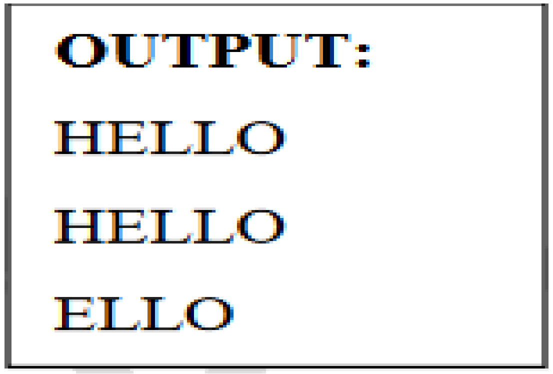
Pointers and string

- String is sequence of characters ends with null ("\0") character.
- C++ provides two methods of declaring and initializing a string.
- **Method 1:**
  - char str1[ ] = "HELLO";
- When a string is declared using an array, the compiler reserves one element longer than the number of characters in the string to accommodate NULL character.
- The string str1[ ] is 6 bytes long to initialize its first 5 characters HELLO\0.
- **Method 2:**
  - char *str2 = "HELLO";

Pointers and string

- C++ treats string constants like other array and interrupts a string constant as a pointer to the first character of the string.
- This means that we can assign a string constant to a pointer that point to a char.
- Example: A program to illustrate the difference between strings as arrays and pointers.

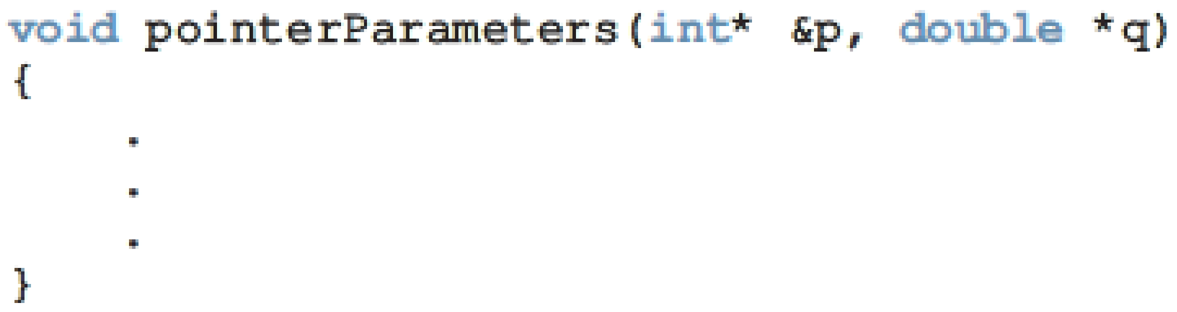
-
  ```
  #include<iostream.h>
  void main( )
  {
          char str1[ ] = “HELLO”;
          char *str2 = “HELLO”;
          cout<<str1<<endl;
          cout<<str2<<endl;
          str2++;
          cout<<str2<<endl;
  }
  ```



```
OUTPUT:
HELLO
HELLO
ELLO
```

Functions and Pointers

- A pointer variable can be passed as a parameter to a function either by value or by reference.



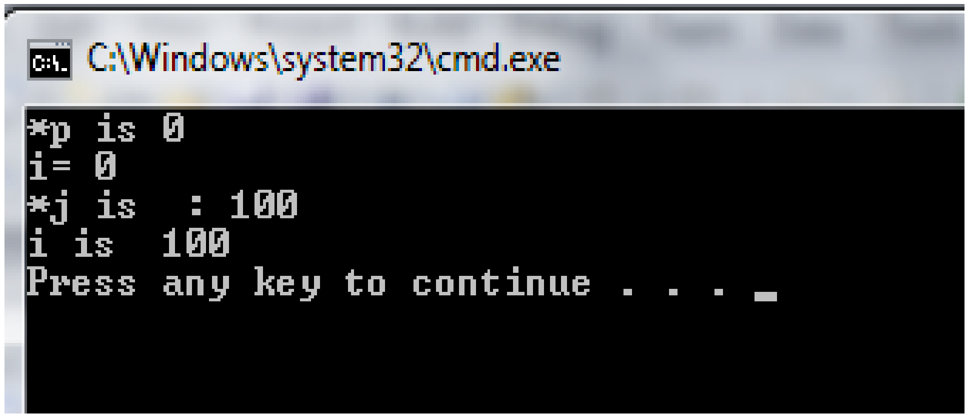
```
void pointerParameters(int* &p, double *q)
{
        .
        .
        .
}
```

- In the function pointerParameters, both p and q are pointers. The parameter p is a reference parameter; the parameter q is a value parameter.

- Furthermore, the function pointerParameters can change the value of *q, but not the value of q. However, the function pointerParameters can change the value of both p and *p.

Example

```
#include <iostream>
using namespace std;
void f(int *j);
int main()
{
int i =0 ;
int *p;
p = &i; // p now points to i
cout<<"*p is " << *p<<endl
<< "i= " << i <<endl;
f(p);
cout << "i is  " << i << endl;
return 0;
}
void f(int *j)
{
*j = 100; // var pointed to by j is assigned 100
cout <<"*j is  : "<< *j<<endl;
}
```



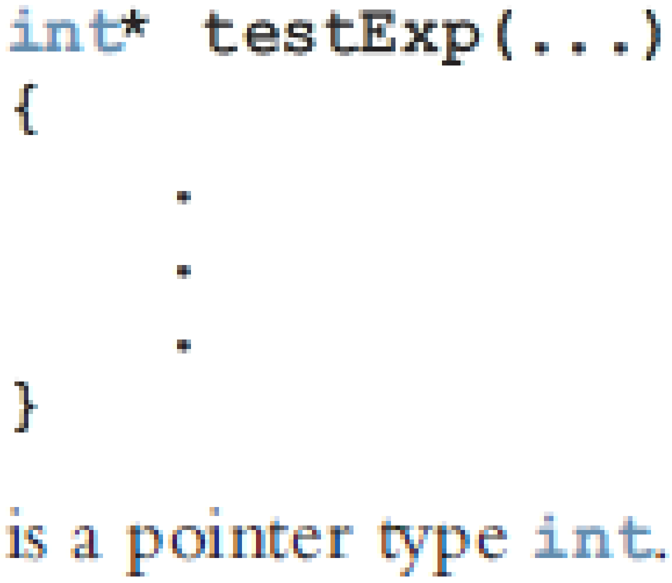
```
*p is 0
i= 0
*j is  : 100
i is  100
Press any key to continue . . . _
```

Functions and Pointers

- Passing a pointer as an argument to a function is in some ways similar to passing a reference.
- They both permit the variable in the calling program to be modified by the function.
- However, the mechanism is different. A reference is an alias for the original variable, while a pointer is the address of the variable.

Pointers and Function Return Values

- In C++, the return type of a function can be a pointer. For example, the return type of the function:

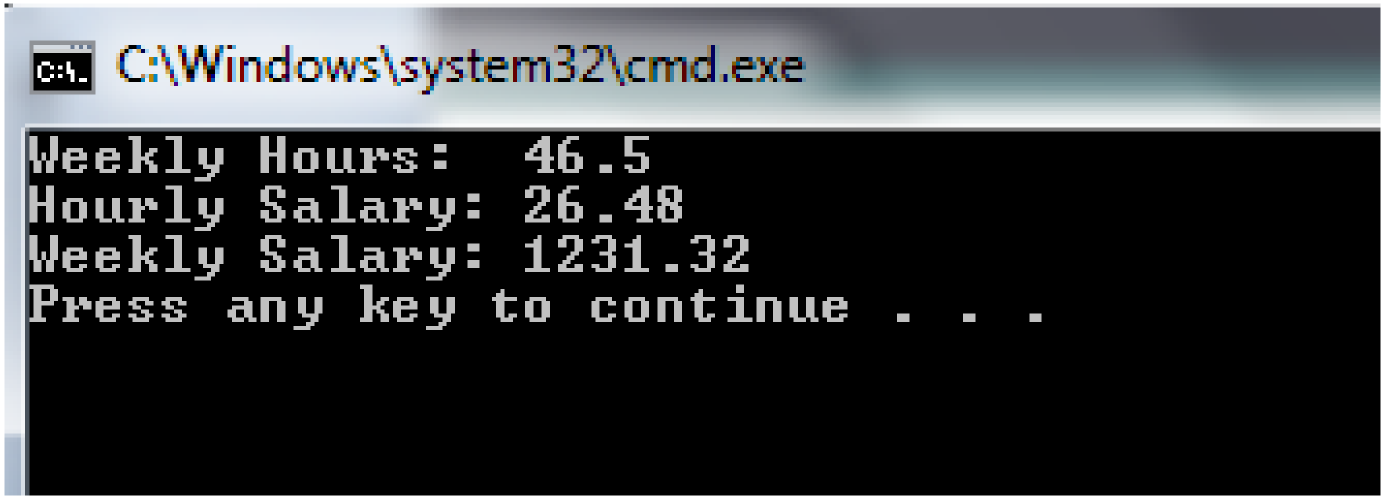


```
int* testExp(...)
{
        .
        .
        .
}
```

is a pointer type int.

Example

```
#include <iostream>
using namespace std;
double * GetSalary()
{
    double salary = 26.48;
    double *HourlySalary = &salary;
    return HourlySalary;
}
int main()
{
    double hours  = 46.50;
    double salary = *GetSalary();
    cout << "Weekly Hours:  " << hours << endl;
    cout << "Hourly Salary: " << salary << endl;
    double WeeklySalary = hours * salary;
    cout << "Weekly Salary: " << WeeklySalary << endl;
    return 0;
}
```



```
Weekly Hours:  46.5
Hourly Salary: 26.48
Weekly Salary: 1231.32
Press any key to continue . . .
```

Memory Allocation of Pointers

- Memory Allocation is done in two ways:

  o Static Allocation of memory

  o Dynamic allocation of memory.

- **Static Allocation of Memory:**
  - Static memory allocation refers to the process of allocating memory during the compilation of the program i.e. before the program is executed.
  - Example:
  - int a; // Allocates 2 bytes of memory space during the compilation.

Memory Allocation of Pointers

- **Dynamic Allocation of Memory:**
  - **Dynamic memory allocation** refers to the process of allocating memory during the execution of the program or at run time.
  - Memory space allocated with this method is not fixed.
  - C++ supports dynamic allocation and de-allocation of objects using **new** and **delete** operators.
  - These operators allocate memory for objects from a pool called the **free store**.
  - The new operator calls the special function operator **new** and **delete** operators call the special function operator delete.

New Operator

- We can allocate storage for a variable while program is running by using **new operator**.
- It is used to allocate memory without having to define variables and then make pointers point to them.
- The following code demonstrates how to allocate memory for different variables.
- To allocate memory type integer
  - int * pnumber;
  - pnumber = new int;

New Operator

- The first line declares the pointer, pnumber. The second line then allocates memory for an integer and then makes pnumber point to this new memory.
- To allocate memory for array,
  - **double *dptr = new double[25];**
- To allocates dynamic structure variables or objects,
  - **student sp = new student**

Delete Operator

- The **delete operator** is used to destroy the variables space which has been created by using the new operator dynamically.
- Use delete operator to free dynamic memory as :
  - **delete iptr;**
- To free dynamic array memory:
  - **delete [] dptr;**
- To free dynamic structure,
  - **delete structure;**

Difference between Static and Dynamic Memory Allocation

Sl no	Static Memory Allocation	Dynamic Memory Allocation
1.	Memory space is allocated before the execution of program.	Memory space is allocated during the execution of program.
2.	Memory space allocated is fixed	Memory space allocated is not fixed
3.	More memory space is required	Less memory space is required.
4.	Memory allocation from stack area	Memory space form heap area.

Free store (Heap memory)

- Free store is a pool of memory available to **allocated and de-allocated storage** for the objects during the **execution of the memory.**
- **Memory Leak:**
- If the objects, that are allocated memory dynamically, are not deleted using delete, the memory block remains occupied even at the end of the program.
- Such memory blocks are known as orphaned memory blocks.
- These orphaned memory blocks when increases in number, bring adverse effect on the system. This situation is called **memory leak**.

THE END OF CHAPTER THREE!!