Discipline Courses-I
Semester-I
Paper: Programming Fundamentals
Unit-V
Lesson: Pointers
Lesson Developer: Rakhi Saxena
College/Department: Acharya Narendra Dev College, University of Delhi

Table of Contents

Introduction

In this chapter, the concept of a pointer has been discussed. The pointer is similar to a broker in human life. The broker operates between the seller and buyer and transacts the value between the two. The broker accepts a value from seller and transfer the value to a buyer. The pointer of C language performs the work in the similar fashion. Every variable stored in the system memory is allocated an unique address. The pointer variable stores the unique memory address of another variable.

The chapter focuses on pointer declaration, pointer initialization and dereferencing of pointers. Further, the chapter throws light on arithmetic operations permissible on pointers. The chapter then discusses void pointers, null pointers and dangling pointers. Further, chapter covers passing pointer as argument to functions. Then, the chapter discusses usage of pointers in arrays followed by concepts like array of pointers, pointer of a pointer, pointer to constant objects and constant pointers. Also, the chapter focuses on dynamic memory allocation.

Learning Objective

After reading this chapter you should be able to:

- Define, initialize, dereference a pointer
- Demonstrate the permissible operations on pointers
- Explain the concept of void and wild pointers
- Pass pointer to functions
- Use Pointers with arrays
- Apply Memory allocation techniques for Pointers

2.1 Fundamentals of Pointers

2.1.1 Concept of Pointers

The pointer is similar to a broker in human life. The broker operates between the seller and buyer and transacts the value between the two. The broker accepts a value from seller and transfer the value to a buyer. The pointer of C++ language performs the work in the similar fashion. The pointers accept a value from a variable, store it and give it to another variable.

Within the computers memory, every stored data occupies one or more contiguous memory cells. Suppose v is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The address of v's memory location can be determined by the use of pointers. **Pointer** is defined as a variable used to store memory addresses. Just like, an integer variable can hold only integers. Each pointer variable can hold only address of a specific data type such as int, char, float etc., or any user defined data type.

2.1.2 Need for Pointers

The usage of the pointer is essential in the following situations:

- Pointers are also closely associated with arrays and therefore provide an alternate way to access individual array elements.
- Moreover, pointers provide a convenient way to represent multidimensional array, allowing a single multidimensional array to be replaced by a lower dimensional array of pointers.
- Pointers can be used to pass information back and forth between a function and its reference point.
- In particular, pointers provide a way to return multiple data items from a function via function arguments.
- Allocating memory from the system dynamically.

2.2 Pointers in C++

2.2.1 Pointer Declaration

The interpretation of a pointer declaration differs from the interpretation of other variable declarations. When a pointer variable is declared, the variable name must be preceded by an **asterisk (*).** This identifies the fact that the variable is a pointer. When a pointer variable is defined, the C++ compiler needs to know the type of variable the pointer points to. The syntax for pointer declaration is as follows:

```
DataType * PtrVar;
```

where DataType could be a primitive data type or user defined structure (such as structures and classes). The PtrVar could be any valid C++ variable name. The character star (*) following the DataType informs the compiler that the variable PtrVar is a pointer variable.

For Example,

```
int *pv;  //pv is a pointer to an integer
```

2.2.2 Pointer Initialization

The pointer variable must be bound to a memory location. It can be achieved by assigning the address of a variable to a pointer variable. The address of variable's memory location can be determined by **address operator "&" (ampersand),** a unary operator. The expression &variable name evaluates the address of its operand.

For example,

```
int *pv;  //pv is a pointer to an integer
int v;    // integer variable v
pv =&v; //  address of variable v is assigned to pointer variable pv
```

here, pv points to v, since it "points" to the location where v is stored in memory. Remember, however, that pv represents v's address, not its value. Thus, pv is referred to as a pointer variable. The relationship between pv and v illustrated in figure 2.1.

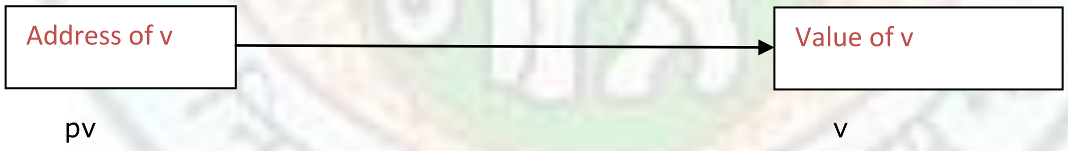


figure 2.1 : Relationship between pv and v

Note that, the above 3 lines of code are equivalent to following 2 lines of code.

```
int v , *pv;
pv=&v;
```

Further, the above 2 lines of code are equivalent to following single line of code.

```
int v , *pv =&v;
```

Value addition: Beware !
Heading text: Operator &
Body text: Sufficient care must be taken to avoid any kind of confusion between the following: • unary address operator & which precedes a variable name and stores it's address. • binary logical operator & which performs a bit-wise AND operation.

Now, let us consider a simple example illustrating the use of address operator "&".

Example 2.1

```
//use of '&' operator to access address
#include <iostream>
using namespace std;
void main ()
{
  int v = 100;  // define and initialize integer variable
  int *pv;     //pv is a pointer to an integer
  pv=&v;    //  address of variable v is assigned to pointer variable pv
  // print the address and content of the above variable
  cout « "Address " « pv « " contains value " « v « endl;
  getchar();
}
```

Output

```
Address 0x8fd3fff4 contains value 100
```

Value addition: Common Misconception
Heading text: Correct Interpretation of Code

Body text: It is important to interpret following line of code correctly.
 	
`int v , *pv =&v;`
 	
It is pv which is being assigned to the address of v, (and not *pv). In other words, think of pv as being declared of type "int *" and initialised to &v.
This statement cannot be written as below
 	
`int *pv=&v, v; //incorrect`
since at the point where pv is set to the address of v, storage has not yet been allocated for v.

2.2.3 Data Type of Pointers

There are different types of pointers, like integer, float and char pointers. These can be declared according to the declaration of type as int, float or char or double. Then the pointer will be able to store the address of declared type. Following program illustrates this fact.

Example 2.2

```
#include <iostream>
using namespace std;
void main( )
{
int *ptr,a=9;
float *ptr1,a1=9.23;
char *ptr2,a2='+';
ptr=&a;
ptr1=&a1;
ptr2=&a2;
cout<<"a="<<a<<" a1="<<a1<<" a2="<<a2;
getchar();
}
```

Output

```
a=9 a1=9.23 a2=+
```

Value addition: Did you Know

Heading text: Storage of Variables in program's Stack Area

Body text:

The addresses printed by the programs, depend on the current configuration of a system. Further, the addresses of the variables are in hexadecimal notation.

Moreover, they are in the decreasing order. From this, it is evident that all variables are created in the program's stack area and the stack always grows from a higher to lower memory address. This is illustrated in following example.

Example 2.3

```
#include <iostream>
using namespace std;
void main ()
{
  // define and initialize integer variables
  int a = 100;
  int b =200;
  int c =300;
 // print the addresses and contents of the above variables
  cout <<"Address " << &a<< " contains value " <<a <<endl;
  cout <<"Address " <<&b<<" contains value " <<b <<endl;
  cout <<"Address " <<&c << " contains value " << c<< endl;
  getchar();
}
```

Output

```
Address 0x8fd3fff4 contains value 100
Address 0x8fd3fff2 contains value 200
Address 0x8fd3fff0 contains value 300
```

Further, each of the addresses differs from others by exactly two bytes, since integer variables are allocated 2 bytes of memory.

2.2.4 Dereferencing of Pointers

Dereferencing is the process of accessing and manipulating data stored in the memory location pointed to by a pointer. The unary operator * (asterisk) is used to dereference pointers. In this case, it is called as the **indirection operator***.* For example, the data item represented by v can be accessed by the expression *pv. Thus, an indirect reference can appear in place of an ordinary variable. i.e. v=*pv because both representation have the same integer value. The following program demonstrates the use of indirection operator " * " with reference to pointers.

Example 2.4

```
#include <iostream>
using namespace std;
void main()
{
          int  u=3, v, *pu, *pv;
          pu=&u;
          v=*pu;
          pv=&v;
          cout<<"\nu = "<<u<<" &u = "<<&u<<" pu = "<<pu<<" *pu = "<<*pu;
          cout<<"\nv = "<<v<<" &v = "<<&v<<" pv = "<<pv<<" *pv = "<<*pv;
          getchar();
     }
```

Output

```
u  = 3 &u = 0x8febfff4 pu = 0x8febfff4 *pu = 3
v  = 3 &v = 0x8febfff2 pv = 0x8febfff2 *pv=3
```

Value addition: Fact
Heading text Precedence & Associatively
Body text: • The unary operators & and * are members of the same precedence group as the other unary operators, i.e. - , ++ , -- , !, sizeof and (type). • The associatively of the unary operators is right to left.

Any operation that is performed on the dereferenced pointer directly affects the value of the variable it points to. The following example illustrates this fact.

Example 2.5

```
#include <iostream>
using namespace std;
void main()
    {
           int u1,u2;
           int v=3;
           int *pv;
           u1=2*(v+5);
```

```
        pv=&v;
        u2=2*(*pv+5);
        cout<<"\n u1="<<u1<<" u2= "<<u2;
        getchar();
    }
```

Output

```
u1=16 u2= 16
```

This program involves the use of two integer expressions. The first, 2*(v+5), is an ordinary arithmetic expression whereas the second, 2*(*pv+5), involves the use of a pointer. The expressions are equivalent, since v and *pv each represent the same integer value.

An indirect reference can also appear on the left side of an assignment statement. This provides another method for assigning a value to a variable. The next example clearly shows the usage of indirection operator * on left hand side of an assignment statement.

Example 2.6

```
#include <iostream>
using namespace std;
void main()
    {
        int v=3;
        int *pv;
        pv=&v;
        cout<<"\n*pv= "<<*pv<<"    v= "<<v;
        *pv=0;
         cout<<"\n*pv= "<<*pv<<"   v= "<<v;
         getchar();
    }
```

Output

```
*pv= 3   v= 3
*pv= 0   v= 0
```

Value addition: Beware !
Heading text & and * operators
Body text: • The address operator (&) must act upon operands that are associated with unique addresses, such as ordinary variables or single array elements. Thus the address operator cannot act upon arithmetic expressions, such as 2* (u+v). • The indirection operator (*) can only act upon operands that are pointers.

Here is a simple example on usage of pointers.
Example 2.7

```
#include <iostream>
using namespace std;
void main( )
{
int a,*ptr,b;
a =9;
ptr = &a; b = *ptr; ++a;
cout<<"a="<<a<<"  *ptr="<< *ptr<<"  b="<<b;
getchar();
}
```

Output

```
a=10  *ptr=10 b=9
```

Another simple example of pointer usage

Example 2.8

```
// pointer (address variables) usage demonstration
#include <iostream>
using namespace std;
{
      int *iptr;              //  pointer to integer, figure 2.2a
     int var1, var2;        //  two integer variables, figure 2.2b
      var1 = 10;            //  figure 2.2c
      var2 = 20;           //  figure 2.2d
      iptr = &var1;       // figure 2.2e
      cout « "Address and contents of var1 is " « iptr « " and " « *iptr;
      iptr = &var2;      //  figure 2.2f
      cout«"\nAddress and contents of var2 is " « iptr « " and" « *iptr;
      *iptr = 125;       //  figure 2.2g
       var1 = *iptr + 1; //  figure 2.2h
       getchar();
}
```

Output

```
Address and contents of var1 is 0x8fdcfff4 and 10
Address and contents of var2 is 0x8fdcfff2 and20
```

Here, in main ( ) , the first statement
int *iptr;
specifies that iptr is a pointer to an integer.
It could also be written as
int* iptr;
It makes no difference as far as the compiler is concerned. But there are certain advantages in following the former convention (i.e., placing the * closer to the variable name). The compiler always associates the * with the pointer variable name rather than the data type.
The above program had made it evident that indirection allows the contents of a variable to

be accessed and manipulated without using the name of the variable. Thus, all variables that can be accessed directly (by their names) can also be accessed indirectly by means of pointers.

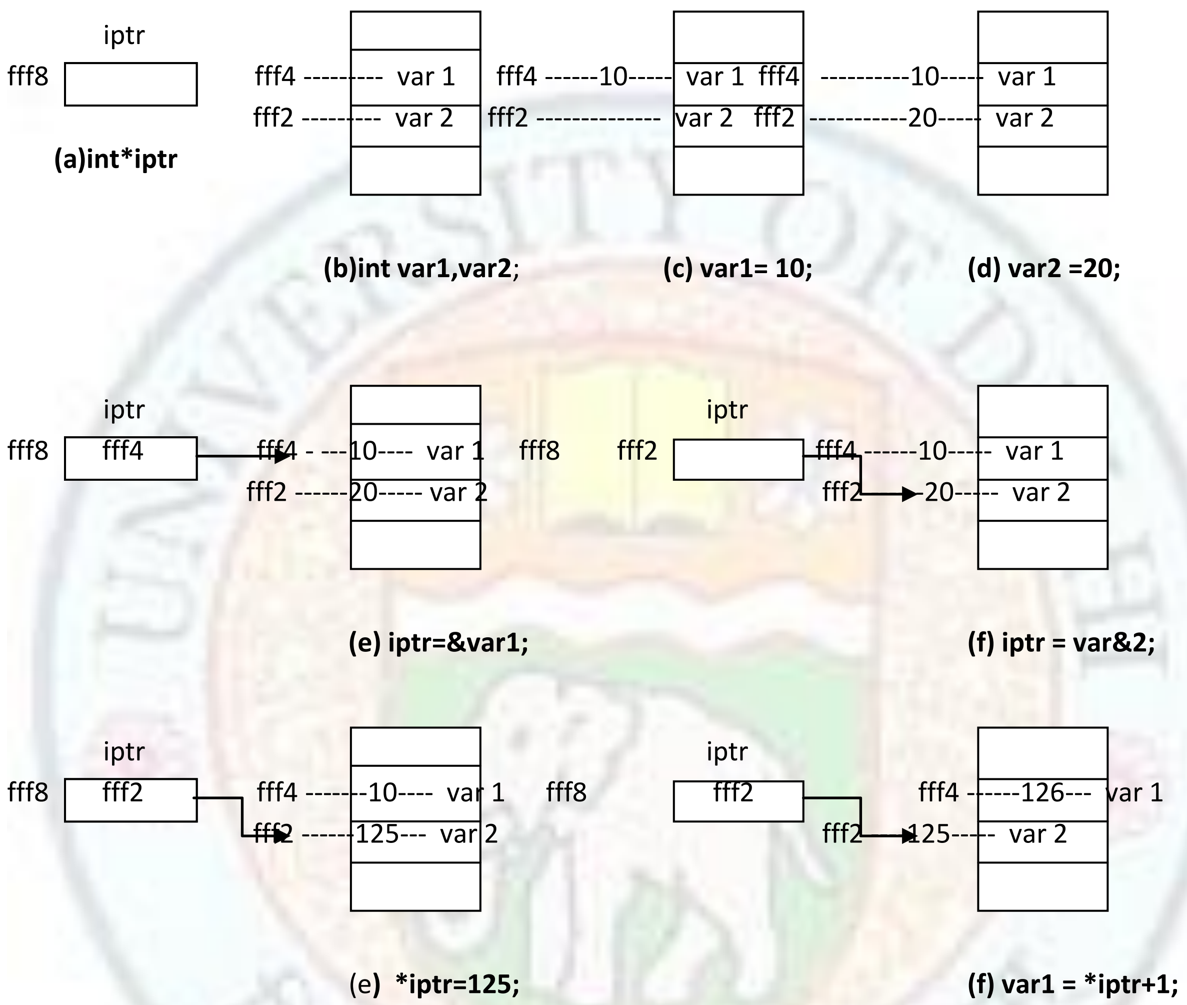


Figure 2.2 Dereferencing of Pointers

2.3 Operations on Pointers

2.3.1 Exchange of values between two pointers

The values can be exchanged between two pointers by assigning their addresses or their values. For example, If ptr1 and ptr2 are two pointers, then their values can be exchanged by equalizing their addresses or their values.

```
ptr2=ptr1; //equalizing their values

*ptr2=*ptr1 //equalizing their addresses
```

This can be seen in the following program

Example 2.9

```
/* Exchange of value between two pointers */
#include <iostream>
using namespace std;
void main()
{
int a,b;
int *ptr1, *ptr2;
a = 5;
ptr1 = &a;
b = (*ptr1) * 2;
ptr2 = ptr1;
cout<<"a="<<a <<"  b="<<b<<"  *ptr1="<< *ptr1<<"   *ptr2="<<*ptr2;
getchar();
}
```

Output

```
a=5  b=10  *ptr1=5  *ptr2=5
```

In this program 'ptr1' pointer receives address of 'a' and 'b' is obtained by multiplying 'ptr1' with 2. Then the address of 'ptr2' pointer is given to ptr1, which automatically transfers 'ptr1' value to 'ptr2' pointer.

2.3.2 Some Permissible operations on Pointers

The number of bytes accessed by using a pointer depends on its type, but the size of the pointer variable remains the same irrespective of the data type to which it is pointing.

For example

- a pointer to an integer accesses 2 bytes of memory
- a pointer to a char accesses 1 byte of memory
- a pointer to a float accesses 4 bytes of memory
- a pointer to a double accesses 8 bytes of memory

The C++ language allows arithmetic operations to be performed on pointer variables. The arithmetic operators available for use with pointers can be classified as

- Unary operators: ++ (increment) and -- (decrement),
- Binary operators: + (addition) and - (subtraction).

The C++ compiler takes into account the size of the data type being pointed, while performing arithmetic operations on a pointer. For example, if a pointer to an integer is incremented using the ++ operator (preceding or succeeding the pointer), then the initial address contained in the pointer is incremented by two and not one, assuming that an integer occupies two bytes in memory. Similarly, incrementing a pointer to float causes the initial address contained in the float pointer to be actually incremented by 4 and not 1. In general, a pointer to some type, d_type (where d_type can be primitive or user defined data type), when incremented by an integral value i, has the following effect:

(current address in pointer) + i * sizeof(d_type)

For example,
px is a pointer variable that represents the address of some variable x, then we can write expressions such as

++px,
--px,
(px+3),
(px+i), and
(px-i),

where i is an integer variable. Each expression will represents an address that is located some distance from the original address represented by px. The exact distance will be the product of the integer quantity and the number of bytes or words associated with the data item to which px points. For example, if px points to an integer quantity then the expression (px+3) will result in an address that is 6 bytes beyond the integer to which px points. This is because each integer quantity requires two bytes of memory.

<u>Example 2.10</u>

```
#include <iostream>
using namespace std;
void main()
{
int *px;
float *py;
cout<<px<<\n;
cout<<py<<\n;
cout<<px+3<<\n;
cout<<py+3<<\n;
getchar();
}
```

<u>Output</u>

```
0x8f97zc52
0x8f970000
0x8f97zc58
0x8f97000c
```

One pointer variable can be subtracted from another provided both variable points to element of same array. The resulting value indicates the number of words or bytes separating the corresponding array elements. The following example shows that one pointer variable can be subtracted from another pointer variable.

<u>Example 2.11</u>

Program in which two different pointer variables point to the first and last elements of integer array.

```
#include <iostream>
using namespace std;
void main()
{
        int *px,*py;
        static int a[6]={1,2,3,4,5,6};
        px=&a[0];
        py=&a[5];
        cout<<"px="<<px<<"py="<<py;
        cout<<"\n\n py-px="<<py-px;
        getchar();
}
```

<u>OUTPUT</u>

```
px=52   py=5c
py-px=5
```

In particular the pointer variable px points to a[0], and py points to a[5]. The difference between, py-px should be 5, since a[5] is the fifth element beyond a[0].

Pointers variables can be compared provided both variables are of the same data type. Such comparisons can be useful when both pointer variables point to element of same array.

Example:
Several logical expressions involving these two variables (int *px, *pv) are shown below. All of the expressions are syntactically correct.

```
        (px<py)
        (px>=py)
        (px==py)
        (px!=py)
        (px==null)
```

Expressions such as (px<py) indicate weather or not the element associated with px is ranked ahead of element associated with py.

These permissible operations are summarized below.

1. A pointer variable can be assigned the address of ordinary variable.
   ```
   int v, *pv;
   pv=&v;
   ```

2. A pointer variable can be assigned the value of another pointer variable.
   ```
   int *pv, *pu;
   pv=pu
   ```

3. A pointer variable can be assigned a null.

int *pv=null;

4. An integer quantity can be added to or subtracted from a pointer variable.
   int *pv;
   pv=pv+5;

5. One pointer variable can be subtracted from another provided both pointers point to elements of same array.
   int *pv, *pu;
   pv=pv-pu;

6. Two pointer variable can be compared provided both pointers point to object of same data type.
   int *pv, *pu;
   (pv>=pu)

Other arithmetic operations on pointers are not allowed.

Value addition: Did you know
Heading text L-value of an expression
Body text: • The left-hand side of any assignment of a numerical value to a quantity can be any expression that denotes an object in memory. Such an expression is called an L-value (or left- value). • For example, 38, *-a* and *&b* are not L-values (since they cannot occur on the left-hand sides of assignments), but *a, *p* and **&a* are L-values. • Any expression beginning with & is not an L-value, whereas any expression beginning with * is an L-value. • An expressions such as &x cannot appear on left side of assignment statement.

2.4 Various Types of Pointers

2.4.1 Void Pointers

Pointers defined to be of a specific data type cannot hold the address of some other type of variable i.e it is synactically incorrect in C++ to assign the address of (say) an integer variable to a pointer of type float.
The assignment of incompatible variable address to a pointer variable results in compilation error. Such type-compatibility problems can be overcome by using a general purpose pointer type called **void pointer**. The format for declaring a *void pointer* is as follows:

```
void *v_ptr; //  define a pointer to void
```

It uses the reserved word void for specifying the type of the pointer. Pointers defined in this manner do not have any type associated with them and can hold the address of any type of variable.

The following are some valid C++ statements:

```
void *vd_ptr;
int *it_ptr;
int invar;.
char chvar;
float flvar;
vd_ptr = &invar; //  valid
vd_ptr = &chvar; //  valid
vd_ptr = &flvar; //  valid
it_ptr = &invar; //  valid
```

The following are some invalid statements:

```
it_ptr = &chvar; //  invalid
it_ptr = &flvar; //  invalid
```

Pointers to void cannot be directly dereferenced like other pointer variables using the **indirection operator**. Prior to dereferencing a pointer to void, it must be suitably typecasted to the required data type. The next program illustrates the typecasting of void pointers while accessing memory locations pointed to by them.

Example 2.12

```
// the use void pointers to hold pointer of any type
#include <iostream>
using namespace std;
void main ()
{
float f1 = 200.5;//define and initialize float f1 to 200.5
void *vptr; //define pointer to void
vptr=&f1; //pointer assignment
cout<<"f1 contains "<<*((float *) vptr);
getchar();
}
```

Output

```
f1 contains 200.5
```

In this program, the expression *((float *) vptr) in the cout statement displays the contents of the variable f1 using a void pointer variable with typecasting.
Note that pointer arithmetic cannot be performed on void pointers without typecasting, since they have no type associated with them.

A pointer is a variable whose value either is 0 or is the address of some other variable or object. If its value is 0, we say the pointer is **null**. Otherwise, we say that the pointer points to the variable or object whose address it stores. An uninitialized pointer is called a **dangling pointer**. For example:

```
int n=44;
string s="Hello";
int *p;          //a dangling pointer
int *q=0;        // a null pointer
int *pn=&n;   //point to n
string*ps=&s;//point to s
```

2.4.2 Wild Pointers

A pointer becomes a **Wild Pointer** when it is pointing to an unallocated memory or when it is pointing to a data item whose memory is already released. Side effects of such pointers are creation of **garbage memory and dangling reference**. The memory becomes garbage memory when a pointer pointing to a memory object (data item) is lost; i.e., it indicates that the memory item continues to exist, but the pointer to it is lost; it happens when memory is not released explicitly. A memory access using a pointer is known as dangling reference when a pointer to the memory item continues to exist, but memory allocated to that item is released; i.e., accessing memory object, for which no memory is allocated.
Pointers become wild pointers under the following situations:

- When a pointer is uninitialized
- Pointer modification
- Pointer referencing to a data which is destroyed

(1) *When pointer is uninitialized:* It contains an illegal address and it is difficult to predict the outcome of a program. For instance, in the definition int *p; it is impossible to predict which integer value the pointer p is pointing to.

For example,

```
int *p; //pointer is uninitialized
for(int i=0; i<10;i++)
        p[i]=p[i]+5; //accessing uninitialized pointer
```

the above lines of code might modify some sensitive data pertaining to a system leading to corruption of the whole system or the program may behave erratically.

An attempt to dereference a dangling pointer or a null pointer, causes a run-time error. In C++, this throws an "exception" which then crashes the program 'because there was no code to "handle" the exception. For example, the following program causes a run-time error due to an attempt to dereference a dangling pointer.

```
 int main ()
        {
          string* ps;
          cout « "*ps = " « *ps « "\n"; // error: ps is not initialized.
        }
```

The exception is thrown when the operating system attempts to evaluate the expression *ps. The pointer ps is not initialized so it is a dangling pointer. i.e. it points to nothing. So the dereference operator * cannot be applied to ps.
To avoid the serious error illustrated in above example, always initialize pointer variables. For example,

```
string name ("Erika") ;
string* p=&name;
```

(2) Pointer modification: The inadvertent storage of a new address in a pointer variable is referred to as pointer modification. This situation will occur when some other wild pointer modifies the address of a valid pointer. It transforms a valid pointer to a wild pointer.

(3) Pointer referencing to a data which is destroyed. In this case, the pointer tries to access memory object or item which no longer exists.

2.5 Pointers and Functions

2.5.1 Passing Pointers to Functions

Pointers can also be passed to a function as argument. When an argument is passed using pointers the address of a data item is passed to a function. The contents of that address can be accessed freely, either within the function or within the calling routine. Moreover, any change that is made to the data item (i.e. pointed by pointer) will be recognized in both the function and calling routine. So, the arguments using pointers are passed by reference. Thus, the use of a pointer as a function argument permits the corresponding data item to be altered globally from within the function.
Example:
Program for difference between ordinary arguments, which are passed by value, and pointer arguments, which are passed by reference,

Example 2.13

```
#include <iostream>
using namespace std;
void funct1(int u,int v);
void funct2(int *pu,int *pv);
void main()
{
int u=1;
int v=3;
cout<<"\n before calling funct1:    u="<<u<<"v="<<v;
funct1(u,v);
cout<<"\n after calling funct1:     u="<<u<<"v="<<v;
cout<<"\n before calling funct2:    u="<<u<<"v="<<v;
funct2(&u,&v);
cout<<"\n after calling funct2:     u="<<u<<"v="<<v;
getchar();
}
void funct1(int u,int v)
{
        u=0;
        u=0;
        cout<<"\n within funct1:     u="<<u<<"v="<<v;
        return;
}
void funct2(int *pu, int *pv)
{
        *pu=0;
        *pv=0;
        cout<<"\n within funct2:     *pu="<<*pu<<"*pv="<<*pv;
}
```

Output

```
 before calling funct1: u=1v=3
 within funct1: u=0v=3
 after calling funct1:  u=1v=3
 before calling funct2: u=1v=3
 within funct2: *pu=0*pv=0
 after calling funct2:  u=0v=0
```

This program contains two function called funct1 and funct2. The first function funct1 receives two integer variables as arguments. These variables are originally assigned the values 1 and 3 respectively. The values are then changed, to 0,0 within funct1. The new values are not recognize in main, however, because the arguments were passed by value, and any changes to the arguments are local to the function in which the changes occur. Now, consider the second function, funct2. This function receives two pointers to integer variables as it arguments. The arguments are identified as pointers by the in direction operators (i.e. the asterisks) that appear in the argument declaration. In addition, argument declaration indicates that the pointers contain the addresses of the integer quantities. Within funct2, the contents of the pointer addresses are reassign the values 0,0. Since the addresses are recognizing in both funct2 and main, the reassigned values will be recognize within main after the call to funct2. Therefore, the integer variables u and v will have their values changed from 1,3 to 0,0.

The output illustrates

- the local nature of the alterations within funct1, and global nature of alterations within funct2.
- The pointer variables, pu and pv, have not been declared else where in main. This is permitted in the function prototype, however, because pu and pv are dummy arguments rather than actual arguments.
  void funct2(int *,int*);

Let us consider a program to swap two numbers using pointers.

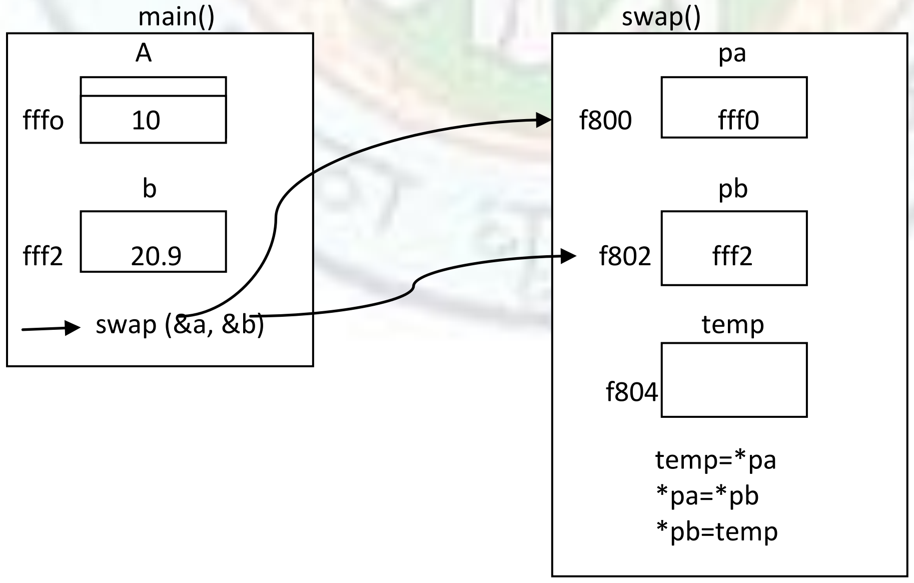
Example 2.14

```
#include <iostream>
using namespace std;
void swap (float *, float *);  //function prototype
void main ( )
{
    float a, b;
    cout « "Enter real number <a>: "; cin » a;
    cout « "Enter real number <b>: "; cin » b;
    // Pass address of the variables whose values are to be swapped
    swap (&a, &b); // figure 9.3a
   cout « "After swapping .......... \n";
   cout « "a contains " « a « endl;
   cout «"b contains " « b;
   getchar();
}
void swap ( float *pa, float *pb ) // function to swap two  numbers
{
    float temp;
    temp = *pa;
     *pa = *pb;
     *pb = temp;
 }
```

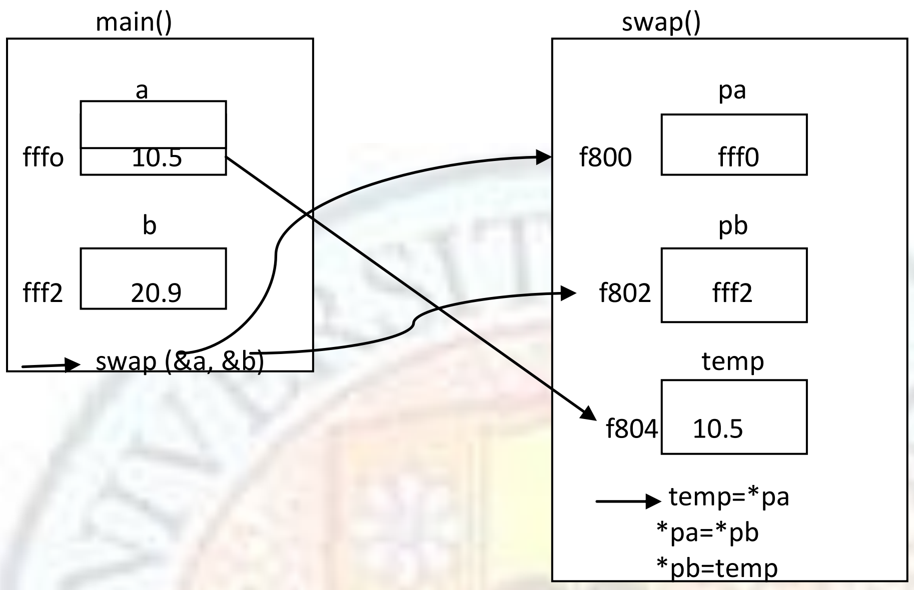
OUTPUT

```
Enter real number <a>: 10.5
Enter real number <b>: 20.9
After swapping
a contains 20.9
b contains 10.5
```

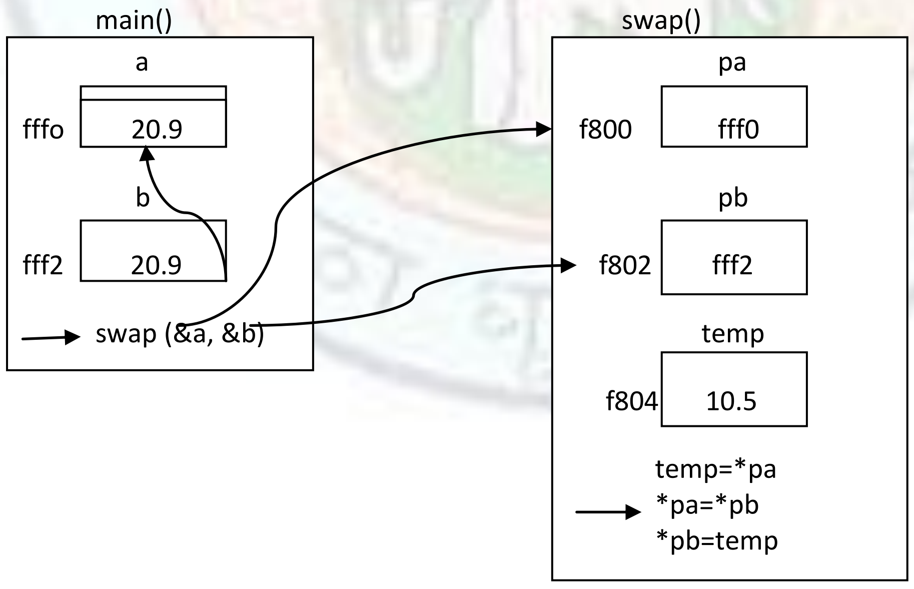
here, in main ( ) , the parameters are accessed directly with their names whereas in swap ( ) , they are accessed using the *indirection operator.* In swap( ), accessing contents of the memory location pointed to by the variable pa, actually accesses the contents of the variable a. Similarly, accessing the contents of the memory location pointed to by the variable pb actually access the contents of the variable b. Hence, swapping the contents of memory using pointer variables pa and pb along with the indirection operator will in fact exchange the contents of the actual parameters a and b (passed by caller) as shown in Figure 2.3



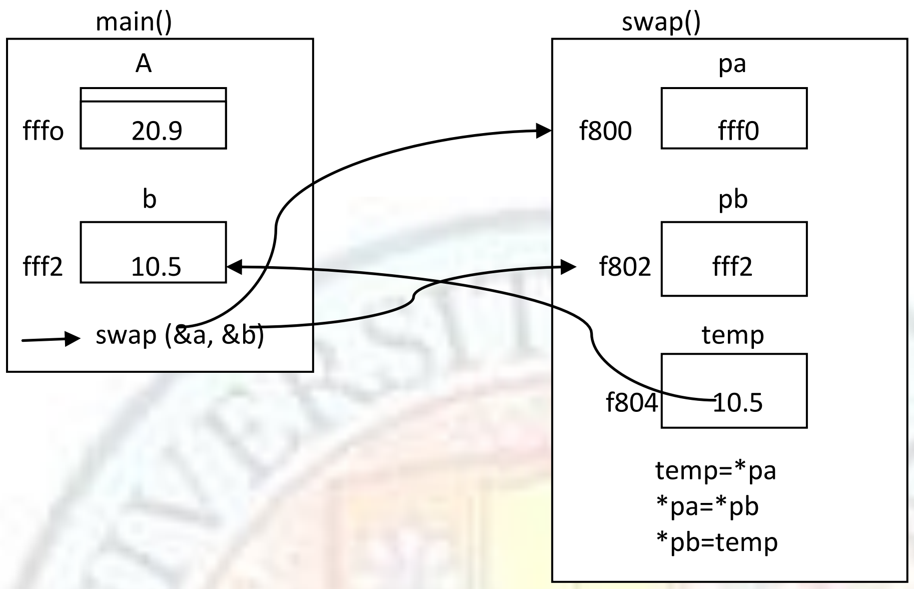
(a) swap (&a, &b)



(b)temp =*pa;



(c)*pa = *pb ;



(d) *pb = temp;

Figure 2.3 Swapping of two numbers

- When an argument is passed to a function parameter by value, the parameter becomes an independent local variable for the function, initialized with the value of that argument, so changes to the parameter have no effect upon the argument.

- When an argument is passed to a function parameter by reference, the parameter becomes a synonym for that argument, so any change to the parameter is a change to the argument.

<u>Example 2.15</u>

```
//Passing by value and by reference
#include <iostream>
using namespace std;
void f(int,int&);
void f(int x, int &y)
{x+=1000;
y*=1000;
}
void main()
{
int m=22;
int n=44;
cout<<"m="<<m<<"\n";
cout<<"n="<<n<<"\n";
f(m,n);
cout<<"m="<<m<<"\n";
cout<<" n="<<n<<"\n";
getchar();
}
```

Output

```
m=22
n=44
m=22
 n=-21536
```

The function **f ()** changes the values of both of its parameters. But only the change on **y** affects its argument because it is passed by reference. Passing by value is safer than passing by reference because it prevents the unintentional changing of the argument. But it has the disadvantage of having to duplicate the argument. When an object is passed to a function that should not change it, it should be **passed *by* constant reference**. This has the advantage of preventing unintentional changes to the argument without the disadvantage of duplicating it. i.e.

```
void f(int x, const int &y);
```

2.5.2 Pointers to Functions

A pointer-to-function can be defined to hold the starting address of a function, and the same can be used to invoke a function. The syntax of defining a pointer to a function is

ReturnType (*PtrToFn) (arguments_if_any) ;

The definition of a pointer to a function requires the function's return type and the function's argument list to be specified along with the pointer variable. It should be remembered that the function prototype or definition should be known before it's address is assigned to a pointer. Once a pointer to a function is defined, it can be used to point to any function which matches with the return type and the argument-list stated in the definition of the pointer to a function. Consider a statement such as

```
int (*any_func) (int, int)
```

It defines the variable any_func as a pointer to a function. The variable any_func can point to any function that takes two integer arguments and returns a single integer value. For instance, it can point to the following functions:

```
int min( int a, int b );
int max( int a, int b );
int add{ int x, int y );
```

Address of a Function

The address of a function can be obtained by just specifying the name of the function without the trailing parentheses. The following statements assign address of the functions to pointer to the function variable any_func since prototype of all of them is same:

```
    any_func = min;
    any_func = max;
    any_func = add;
```

Passing Function Address

The address of a function can be passed as an argument to functions, either by a function name or a pointer holding the address of a function. The following program illustrates these concept. It takes two integer parameters and returns the largest and smallest among them.

Invoking a Function using Pointers

The syntax for invoking a function using a pointer to a function is as follows:

*(*PtrToFn)(arguments_ifany);*
or
PtrToFn(arguments_ifany);

Consider the following pointer to functions

```
int (*pfunc1) ( int );
float (*pfunc2) ( float, float );
```

If these hold addresses of an appropriate function. the statements

```
(*pfunc1)( 2);
(*pfunc2) ( 2.5, a );
pfunc1 ( i );
```

invoke functions pointed to by them. The parameters can be constants or variables. In the definition of pointers to functions, the pointer variable along with the symbol * plays the role of the function name. Hence, while invoking functions using pointers the function name is replaced by the pointer variable.

<u>Example 2.16</u>

```
//passing pointer to function type parameters
#include <iostream>
using namespace std;
int small (int a, int b)
{
  return a<b?a:b;
}
int large (int a, int b)
{
  return a>b?a:b;
}
```

```
int select(int (*fn)(int,int),int x, int y)
{
  int value=fn(x,y);
  return value;
}
void main( void )
  {
        int m, n;
        int (*ptrf) (int, int); //definition of pointer to function
        cout <<"Enter two integers: ";
        cin>>m>>n;
        int high = select( large, m, n ); //function as parameter
        ptrf = small;
        int low = select( ptrf, m, n ); // pointer to function as parameter
        cout <<"Large = " << high << endl;
        cout << "Small = "  << low;
        getchar();
}
```

Output

```
Enter two integers:
12
22
Large = 22
Small = 12
```

In the above program, the function declarator

```
int select( int (*fn) (int, int), int x, int y )
```

indicates that it takes the pointer to a function as the first parameter and the remaining two integer parameters.

In main ( ), the statement

```
int high = select( large, m, n );
```

passes the address of the function large () and two integer variables as actual parameters. The pointer to the function parameter large operates on the last two parameters m and n and returns an integer result. Similarly, the statement

```
int low = select( ptrf, m, n );
```

passes a pointer to a function variable ptrf (note that, ptrf is initialized to the address of small ( ). Such a mechanism is useful in selecting the type of operation to be performed at runtime.

Value addition: Fact
Heading text Recursive call to main()
Body text: When an attempt is made to invoke main () within a program, generally compilers

generate an error message such as: cannot call main from within the program. Because in C++, main () cannot be invoked recursively. This restriction can be violated by using a pointer to functions. The next program invokes main () recursively using a pointer to functions.

Example 2.17

```
// recursive call to main () using a pointer to functions
#include <iostream>
using namespace std;
void main()
{
void (*p) ();
cout« "Hello ... ";
p = main;
(*p) ();
getchar();
}
```

Output

```
Hello ... Hello ... Hello ... Hello ... Hello ... Hello ... Hello ... Hello ... Hello ... Hello ...
```

The above program generates Hello ... message indefinite number of times. It stops when stack overflow occurs. In main ( ), the last two statements assign the address of main to the pointer p and transfer control to main () using pointer to a function respectively.

2.6 Pointers and Arrays

2.6.1 Pointers and One Dimensional Arrays

Since an array name is actually a pointer to the first element in the array. Therefore, if x is one dimensional array, then the address of the first array element can be expressed as either &x[0] or simply as x. Moreover, the address of the second array element can be written as either &x[1] or as (x+1), and so on. In general, the address of array element (i+1) can be expressed as either &x[i] or as (x+i). Thus we have two different ways to write the address of any array element:

- we can write actual array element, preceded by an ampersand; or
- we can write an expression in which subscript is added to the array name.

In the expression (x+i), for example, x represent an address, whereas i represent as an integer quantity.
Moreover, x is the name of an array whose element may be character, integer, floating-point quantities, etc (though all of the array element must be of same data type). Thus, we are not simply adding numerical values. Rather, we are specifying an address that is a certain number of memory cells beyond the address of the first array element. Or, in simpler terms, we are specifying a location that is i array element beyond the first. Hence, the expression (x+i) is a symbolic representation for an address specification rather than an arithmetic expression. Recall that the number of memory cell associated with an array element will depend upon the data type of the array. For example, an integer quantity occupies two bytes (two memory cells), a floating-point quantity requires four bytes, and are double precision quantity required eight byte of quantity.

Since &x[i] and (x+i) both represent the address of the ith element of x, it would seem reasonable that x[i] and *(x+i) both represent content of that address, i.e, the value of the ith element of x. For example, consider a one dimensional array as shown below:

0	1	2	3	4	5	6	7	8	9
3	7	-9	3	6	-1	7	9	1	

Here,

array[0] = *array = 3

Thus, the array becomes a pointer of its own in a function program and acquires base address of array. Also,
array[1] =* (array +1) = 7
array[2] =* (array +2) =-9
array[3] =* (array +3) = 3
Thus, by incrementing the base address of array any element of array can be acquired by use of pointers. Following program illustrates the relationship between array elements and their addresses.

Example 2.18

```
#include <iostream>
using namespace std;
void main()
{
        static int x[10]={10,11,12,13,14,15,16,17,18,19};
        int i;
        for(i=0;i<=9;i++)
{
        // display an array element
        cout<<"\n\t i="<<i<<"\tx[i]="<<x[i]<<"\t*(x+i)="<<*(x+i);
        // display an corresponding array element
        cout<<"\n\t x[i]="<<x[i]<<"\t(x+i)="<<(x+i);
}
}
```

OUTPUT

```
        i=0     x[i]=10         *(x+i)=10     &x[i]=72       x+i=72
        i=1     x[i]=11         *(x+i)=11     &x[i]=74       x+i=74
        i=2     x[i]=12         *(x+i)=12     &x[i]=76       x+i=76
        i=3     x[i]=13         *(x+i)=13     &x[i]=78       x+i=78
        i=4     x[i]=14         *(x+i)=14     &x[i]=7A       x+i=7A
        i=5     x[i]=15         *(x+i)=15     &x[i]=7C       x+i=7C
        i=6     x[i]=16         *(x+i)=16     &x[i]=7E       x+i=7E
        i=7     x[i]=17         *(x+i)=17     &x[i]=80       x+i=80
        i=8     x[i]=18         *(x+i)=18     &x[i]=82       x+i=82
        i=9     x[i]=19         *(x+i)=19     &x[i]=84       x+i=84
```

When assigning a value to an array element such as x[i], the left side of the assignment statement may be written as either x[i] or *(x+i). Thus, a value may be assigned directly to an array element, or it may be assigned to the memory area whose address is that of the array element.

For example,

```
int line[80];
/* assign values */
```

```
line[2]=line[1];
line[2]=*(line+1);
*(line+2)=line[1];
*(line+2)=*(line+1);
```

Each of the four assignment statements assigns the value of the second array elements (i.e., line[]) to the third array element (line[2]). Thus, the four statements are all equivalent. An experienced programmer would probably choose either the first or the fourth, however, in order that the notation be consistent.

Value addition: Common Misconceptions

Heading text Illegal Assignment Statement

Body text:

- Expressions such as x, (x+i) and &x[i] cannot appear on the left side of an assignment statement.
- The address of one array element cannot be assign to some other array element. Thus we cannot write a statement such as
  &line[2]=&line[1];

The next program illustrates the use of pointer holding the address of arrays and pointer arithmetic in manipulating large amount of data stored in sequence.

Example 2.19

```
// smallest in an array of 'n' elements using pointers
#include <iostream>
using namespace std;
void main ()
{
int i, n, small, *ptr, a[50];
cout <<"Size of the array? ";
cin >>n;
    cout <<"Array elements ?\n";
    for (i = 0; i < n; i++)
     cin >>a[i];
    //assign address of a[0] to pointer 'ptr'.
    // This can be done in two ways:
    // 1. ptr = &a[0];
    // 2. ptr = a;
    ptr = a;
  // contents of a[0] assigned to small
  small = *ptr;
```

```
   // pointer points to next element in the array i.e., a[1]
   ptr++;
   // loop n-l times to search for smallest element in the array
   for (i = 1; i < n; i++)
  {
  if(small > *ptr)
      small = *ptr;
   ptr++; // pointer is incremented to point to a[i+1]
}
cout <<"Smallest element is " << small;
getchar();
}
```

Output

```
Size of the array?
5
Array elements ?
2
3
22
4
0
Smallest element is 0
```

Value addition: Did you Know

Heading text Precedence of * and [] operators

Body text:

In C++, the notations *p [3] and (*p)[3] are different since * operator has a lower precedence than [] operator. The following examples illustrate the difference between these two notations:

1. int *data[10];

It defines an array of 10 pointers. The increment operation such as
    data++; or ++data;
is invalid; the array variable data is a constant pointer.

2. int (*data) [10];

It defines a pointer to an array of 10 elements. The increment operation such as
    data++; or ++data;
is invalid; the variable data will point beyond 10 integers, i.e. 10 * sizeof (int) will be added to the variable data.

The name of an array holds the starting address of the array. Hence if arr [3] is an array of any data type, then the name of the array arr is the address of (and does not point to) the 0th element of the array and arr+ 1 is the address of the 1st element of the array. If arr is a pointer, then arr+i cannot be replaced by an expression arr++ executing i times. Using the increment operator with it (the name of the array) is incorrect as the starting address of the array has been placed in the code directly by the compiler, thus making the array name a constant. The array name does not have any storage location allocated unlike a pointer variable which itself has a storage location. Hence, performing an increment operation on the address of the array (which is a constant) is like performing the increment operation; 5++ which is meaningless. The following program illustrates these concepts.

Example 2.20

```
// Pointers can be incremented but not an array
#include <iostream>
using namespace std;
void main()
{

int ia[3] = { 2,5,9};
int *ptr=ia;
for( int i = 0; i < 3; i++)
{ //cout<<*(ia++);  error, array address of ia cannot be changed
cout <<" " <<*ptr++;   // pointer update
}
getchar();
}
```

Output

```
2 5 9
```

In the above program, the elements of the array are accessed using the pointer ptr which is assigned the starting address of the array ia. The pointer variable ptr is incremented every time to point to the next element. The expression ia++ is incorrect.

Value addition: Did you Know
Heading text Passing Array as an argument to a function
Body text: • We have already mentioned the fact that an array name is actually a pointer to the array; i.e. the array name represents the address of the first element in the array. Therefore, an array name is treated as a pointer when it is passed to the function. However, it is not necessary to precede the array name with an ampersand within the function call. • It is possible to pass a portion of an array, rather than an entire array, to a function. To do so, the address of the first array element to be passed must be specified as an argument. The remainder of the array, starting with the specified array element, will then be passed to the function. For example, **Example 2.21** #include <iostream> using namespace std; void process(float z[]) main() {

```
        float z[100];
        ………………..
        /* enter elements for elements of z */

        ………………..
        Process(&z[50]);
        …………………
}
void process(float f[])
{
        …………………
        /* process elements of f */
        /* ranging from f[50] to f[99] */
       …………………...
        return;
}
```

Within main, z is declared to be a 100-element, floating-point array. After the elements of z are enter into the compute, the address of z[50] (i.e. &z[50]) is passed to the function process. Hence, the last 50 elements of z (i.e. the elements z[50] to z[99]) will be available to function "process()".

The outline of function "process()" can also be written as

```
        void process(float *f)
        {
                …………
                /* process elements of f */
                /* ranging from f[50] to f[99] */
               …………..
                return;
        }
```

2.6.2 Array of Pointers

As a pointer variable always contains an address, an **array of pointers** is a collection of addresses. The elements of an array of pointers are stored in the memory just like the elements of any other kind of array. T*he* syntax for defining an array of pointers is the same as array definition, except that the array name is preceded by the star symbol during definition as follows:

*DataType *ArrayName[ARRSIZE];*

An array of pointers is useful for holding a pointer to a list of strings. They can be utilized in implementing algorithms involving excessive data movements. It is a traditional style to sort data, by data movement. This method of sorting incurs much overhead in terms of both the time and space complexity, as it requires temporary space for exchanging the data between the records and has excessive data movement. This is especially true if the size of the data being sorted is large. Pointers can be utilized to perform the same with much flexibility and less overhead. In this method, instead of data exchange, pointers are exchanged to accomplish the same task.

2.6.3 Pointers and Multi Dimensional Arrays

The pointers can be used in two dimensional arrays as similar to single dimensional array. It is possible to access the two dimensiol1al array elements using pointers in the same way as the one-dimensional array. Each row of the two dimensional array is treated as one dimensional array. The name of the array indicates the starting address of the array. The expressions arrayname [i] and (arrayname+i) point to the ith row of the array. Therefore, *(arrayname+ i) +j points to the jth element in the ith row of the array. The subscript j actually acts as an offset to the base address of the ith row. The two dimensional array elements can also be accessed by using the notation arrayname[i][j].

Three-dimensional Array

A three dimensional array can be thought of as an array of two dimensional arrays. Each element of a three dimensional array is accessed using three subscripts, one for each dimension. As usual, the array name points to the base address of the three dimensional array. The array name with a single subscript i contains the base address of the ith two-dimensional array. Hence arrayname [i] or (arrayname+ i) is the address of the ith two dimensional array. The expression arrayname [i] [j] or * (arrayname+i) +j represents the base address of the jth row in the ith two dimensional array. Similarly, the expression * (* ((arrayname+ i)+j) + k) points to the kth element in the jth row in the ith two dimensional array.

We can define a 2-dimensional array as a pointer to a group of contiguous 1-dimensional array.

```
Data-type (*ptvar)[expression2];
        Rather than
Data-type array [expression1][expression2];
```

This concept can be genralised to higher dimensional arrays; i.e.

```
Data-type array[expression1][ expression2]………[ expression n];
```

In this declarations data-type refers to the data type of the array, ptvar is the name of pointer variable, array is the corresponding array name, and expression1, expression2, ….. expression n are positive valued integers expressions that indicate the maximum no. of array elements associated with the sub-script.

Example:
Suppose x is 2-d integer array having 10 rows and 20 columns. We can declare x as

```
        int (*x)[20];
Rather than
        int x[10][20];
```

In the first declaration, x is defined to be a pointer to a group of contiguous, 1- dimensional, 20 element integer arrays. Thus, x points to the first 20 elements array, which is actually the first row of the original 2- dimensional array. Similarly (x+1) points to second 20 element array, which is second row of original 2- dimensional array and so on.

Now consider a three-dimensional floating point array t. This array can be defined as

```
        float (*t)[20][30];
Rather than
        float t[10][20][30];
```

In the first declaration, t is defined as a pointer to a group of contiguous 2- dimensional, 20x30 floating point arrays. Hence t points to the first 20x30 array, (t+1) points to second 20x30 array, and so on.

Example 2.22

```
//matrix of unknown number of rows and known number of columns
#include <iostream>
using namespace std;
void show( int a[] [3], int m )
{
int (* c) [3] ; // pointer to an array of 3 elements
c= a;
for( int i = 0; i < m; i++ )
{
for ( int j = 0; j < 3; j++)
cout << c[i] [j] << " ";
cout << endl;
}
}
void main ()
{
        int c[2] [3]={{1,2,3}, {4,5,6}};
        show (c, 2);
}
```

OUTPUT

```
1 2 3
4 5 6
```

An individual array element within a multi-dimensional array can be accessed by repeated use of indirection operator. When an n-dimensional array is expressed in this manner an individual array element within the n-dimensional array can be accessed by a single use of indirection operator. The following expression illustrates how this is done.

For Example:

An individual array element, such as x[2][5], can be accessed by writing

```
*(x[2]+5)
```

In this expression, x[2] is a pointer to the first element in row 2, so that (x[2]+5) points to element 5 within row 2. The object of this pointer, *(x[2]+5), therefore refers to x[2][5].

Now consider a three-dimensional floating point array t. suppose the dimensionality of t is 10x20x30. This array can be expressed as a 2-d array of pointers by writing

```
float *t[10][20];
```

Therefore we have 200 pointers, each pointing to 1-dimensional array.

An individual array element, such as t[2][3][5], can be accessed by writing

```
*(t[2][3]+5)
```

In this expression, t[2][3] is a pointer to the first element in the 1-d array represented by t[2][3]. Hence, (t[2][3]+5) points to element 5 within this array. The object of this pointer, *(t[2][3]+5), therefore represents t[2][3][5]. This situation is directly analogous to the 2-d case.

Example 2.23

```
//  pointer to 3-dimensional arrays
#include <iostream>
using namespace std;
void main()
{
int arr [2] [3] [2] = { {{ 2,1}, {3 , 6}, {5, 3}}, {{0, 9}, {2 , 3}, {5, 8} }};
cout << arr << endl;
cout << *arr << endl;
cout << **arr << endl;
cout << ***arr << endl;
cout << arr+1 << endl;
cout << *arr+1 << endl;
cout << **arr+1 << endl;
cout << ***arr+1 << endl;
for( int i=0; i < 2; i++ )
{
for( int j=0; j < 3; j++)
{
for( int k=0; k < 2; k++)
{
cout << "arr[" << i << "] [" << j << "] [" << k<< "] = ";
cout << *(*(*(arr+i)+j)+k) << endl;
}
}
}
}
```

OUTPUT

```
0x8fe7ffdc
0x8fe7ffdc
0x8fe7ffdc
```

```
2
0x8fe7ffe8
0x8fe7ffe0
0x8fe7ffde
3
arr[0] [0] [0] = 2
arr[0] [0] [1] = 1
arr[0] [1] [0] = 3
arr[0] [1] [1] = 6
arr[0] [2] [0] = 5
arr[0] [2] [1] = 3
arr[1] [0] [0] = 0
arr[1] [0] [1] = 9
arr[1] [1] [0] = 2
arr[1] [1] [1] = 3
arr[1] [2] [0] = 5
arr[1] [2] [1] = 8
```

2.7 More on Pointers

2.7.1 Pointer of Pointer

In C++ language a double pointer can be declared as a pointer of a pointer. This pointer can be declared by using ** operator. The syntax for defining a pointer to pointer is:

```
DataType **PtrToPtr;
```

which uses two * symbols (placed one beside the other). It implies that PtrToPtr is a pointer-to-a-pointer addressing a data object of type DataType.
For example,

```
   int **ptr;
```

This means that the ptr pointer is a double pointer indicated by **. This double pointer acts as a pointer of a pointer. For example,

```
int a = 100,, *b, **c;
b=&a; // b contains address of integer a. Hence, b is an integer pointer
c=&b; // c contains address of an integer pointer b. Hence, c is a pointer to an integer
        // pointer.
cout<<a <<*b<<**c;
```

The output will be same for all three i.e.

```
100 100 100
```

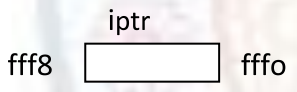
The following program illustrates the format for defining and using a pointer to another pointer.

Example 2.24

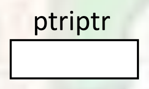
```
//definition and use of pointers to pointers
#include <iostream>
using namespace std;
void main (void)
 {
 int *iptr;   // iptr as a pointer to an integer, figure 2.4a
int **ptriptr; // Defines pointer to int pointer, figure 2.4b
int data;      // Some integer location, figure 2.4c
iptr = &data;  // iptr now points to data, figure 2.4d
ptriptr = &iptr; // ptriptr points to iptr. figure 2.4e
*iptr = 100;    // Same as data = 100, figure 2.4f
cout <<"The variable 'data' contains" <<data <<endl;
**ptriptr = 200; //Same as data = 200, figure 2.4g
cout <<"The variable 'data' contains" <<data <<endl;
data = 300; //figure 2.4h
cout << "ptriptr is pointing to " << **ptriptr <<endl;
getchar();
}
```

OUTPUT

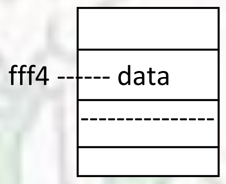
```
The variable 'data' contains100
The variable 'data' contains200
ptriptr is pointing to 300
```



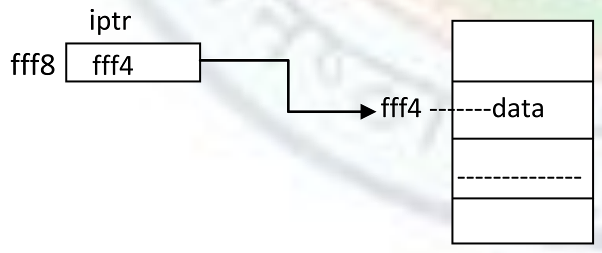
(a) int *iptr



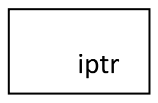
(b) int **ptriptr

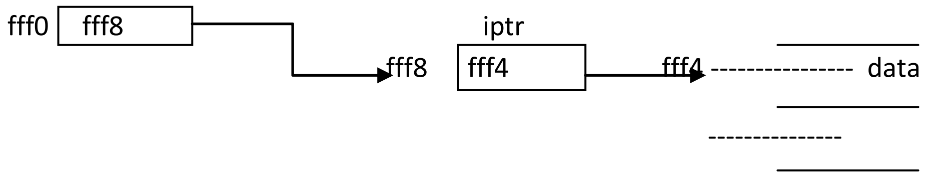


(c) int data;

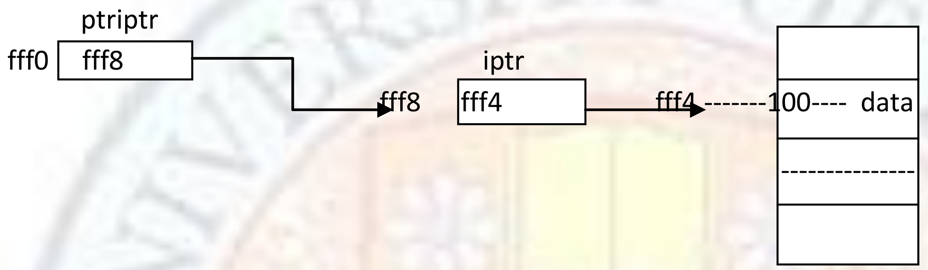


(d) iptr= & data;

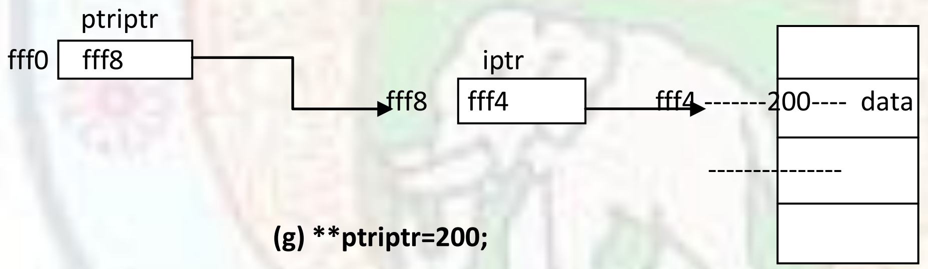




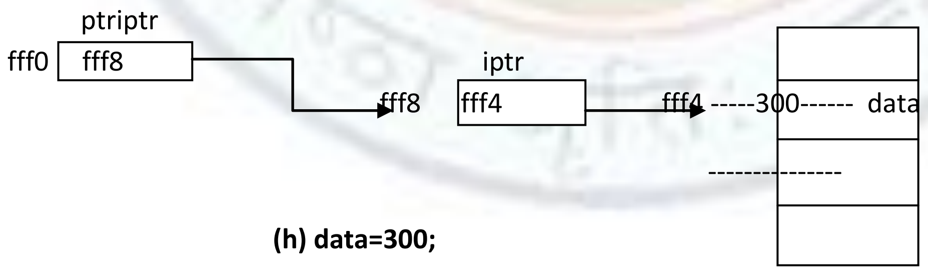
(e)ptriptr=&iptr;



(f) *iptr=100 ;



(g) **ptriptr=200;



(h) data=300;

Figure 2.4 Usage of Pointers to pointers

When a pointer variable is defined, a memory location for the pointer is allocated, but it will not be initialized. Before using the pointer variable, it should be initialized. If the pointer variable has to be initialized in a function other than where it is defined, then the pointer's address has to be passed to the function. The contents of the *pointer to a pointer* variable can be used to access or modify the pointer type formal variable. The next program illustrates passing the address of a pointer. So that the pointer can be made to point to a desired variable (in this program, it is the largest of two integers).

Example 2.25

```
//program to find the largest number using pointers
#include <iostream>
using namespace std;
void FindBig(int *pa, int *pb, int **pbig)
{
//compare the contents of *pa and *pb and assign their address to pbig
if ( *pa > *pb )
*pbig = pa;
else
*pbig = pb;
}
void main ()
{
int a, b, *big;
cout <<"Enter two integers: ";
cin >>a >> b;
FindBig( &a, &b, &big );
cout <<"The value as obtained from the pointer: " << *big;
getchar();
 }
```

Output

```
Enter two integers:
22
33
The value as obtained from the pointer: 33
```

2.7.2 Pointers and Dynamic Memory Allocation

Although in many situations programming can be done without the use of pointers, their usage enhances the capability of the language to manipulate data. **Dynamic memory allocation** is a programming concept wherein the use of pointers becomes indispensable. The allocation of memory space for data structure (storage) during the course of program execution is called dynamic memory allocation. Dynamic variables so created can only be accessed with pointers. For instance, to read the marks of a set of students and store them for processing, an array can be defined as follows:

```
float marks[100];
```

But this method limits the maximum number of students (to 100), which must be decided during the development of the program. On the other hand, by using dynamic allocation, the program can be designed so that the limit for the maximum number of students is restricted only by the amount of memory available in the system. For example, it is possible to define marks array as a pointer variable rather than an array. Thus, we can write

```
float *marks; // float is defined as a pointer

Rather than

float marks[100]; // float is defined as array
```

However, marks is not automatically assigned a memory block when it is defined as a pointer variable, though a block of memory large enough to store 100 float quantities will be reserved in advance when marks is defined as an array.

To assign sufficient memory for marks when defined as pointer, we can make use of **new operator.** The syntax for new operator is as follows:

```
DataType * new Data Type[size in integer];
```

where,
"Datatype *" is return ype, pointer to datatype
"new" is new operator
"Data Type" is datatype
"size in integer" is number of items to be allotted.

The operator new allocates a specified amount of memory during runtime and returns a pointer to that memory location. It computes the size of the memory to be allocated by

```
sizeof( DataType ) * (size in integer)
```

where DataType can be a standard data type or a user defined data type and size in integer can be an integer expression, which specifies the number of elements in the array: The new operator returns NULL, if memory allocation is unsuccessful.

Thus, following statement allocates memory to marks array:

```
marks = new float * [size]; //where size can be specified at runtime.
```

The new operator's counterpart, **delete**, ensures the safe and efficient use of memory. This operator is used to return the memory allocated by the new operator back to the memory pool. Memory thus released, will be reused by other parts of the program. Although, the memory allocated is returned automatically to the system, when the program terminates, it is safer to use this operator explicitly within the pointer. This is absolutely necessary in situations where local variables pointing to the memory get destroyed when the function terminates, leaving memory inaccessible to the rest of the program. The syntax of the delete operator is:

```
delete Ptrvar;
```

where,
delete is delete operator
Ptrvar is pointer returned through new operator

For example, to deallocate the memory assigned to marks array, following statement is required:

delete marks;

The following program illustrates the concept of dynamic allocation and deallocation using new and delete operators.

Example 2.26

```
// addition of two vectors
#include <iostream>
using namespace std;
void AddVectors ( int *a, int *b, int *c, int size)
{
for(int i = 0; i < size; i++)
c[i] = a[i] + b[i];
}
void ReadVector( int *vector, int size)
{
for(int i = 0; i < size; i++ )
cin>>vector[i];
}
void ShowVector( int *vector, int size)
{for(int i = 0; i < size; i++)
cout << vector[i] << " ";
}
void main ()
{clrscr();
int vec_size;
int *x, *y, *z;
cout << "Enter size of Vector: ";
cin >>vec_size;
// allocate memory for all the three vectors
x = new int[ vec_size ]; // x becomes array of size vec_size
y = new int[ vec_size ]; // y becomes array of size vec_size
z = new int[ vec_size ]; // z becomes array of size vec size
cout << "Enter elements of vector x: ";
ReadVector( x, vec_size );
cout << "\nEnter elements of vector y: ";
ReadVector( y, vec_size);
AddVectors( x, y, z, vec_size ); // z = x+y
cout<< "\nSummation Vector z = x + y: ";
ShowVector( z, vec_size );
// free memory allocated to all the three vectors
delete x; // memory allocated to x is released
delete y; // memory allocated to y is released
delete z; // memory allocated to z is released
}
```

OUTPUT

```
Enter size of Vector: 5
Enter elements of vector x: 1
2
3
4
5

Enter elements of vector y: 9
8
7
6
5

Summation Vector z = x + y: 10 10 10 10 10
```

In main (),the following statements

```
x = new int[ vec_size ]; //x becomes array of size vec_size
y = new int[ vec_size]; //y becomes array of size vec_size
z = new int[ vec_size]; //z becomes array of size vec_size
```

allocate memory of size vec_size (integer value read previously) to the integer pointer variables x, y, and z respectively. It is equivalent to defining an array of size vec_size statically but the size of the array must be known at compile time. This inflexibility of array definition is circumvented by using dynamic allocation known as programmer-controlled memory management. The following statements

```
delete x; //memory allocated to x is released
delete y; //memory allocated to y is released
delete z; //memory allocated to z is released
```

release the memory of size vec_size (integer value read-previously) allocated to the integer pointer variables x, y, and z respectively. An array defined statically is released automatically by the system whenever the array goes out of scope. But dynamically allocated arrays must be explicitly released by the delete operator.

Similarly, Pointers also permit the creation of multi-dimensional arrays dynamically so that the amount of memory required by the array can be determined at runtime depending on the problem size. A two dimensional array can be thought of as a collection of a number of one dimensional arrays each representing a row. The 2D array is stored in memory in the row major form and it can be created dynamically using the following steps:

1. Define a pointer to pointers matrix variable: int **p;
2. Allocate memory for storing pointers to all rows of a matrix:
   ```
   p = new int *[ row];
   ```
3. Allocate memory for all column elements:
   ```
   for( int i = 0; i < row; i++ )
       p[i] = new int[col ];
   ```

Thus, pointers offer tremendous flexibility in the creation of dynamic variables, accessing and manipulating the contents of memory location and releasing the memory occupied by the dynamic variables, which are no longer needed.

2.7.3 Pointers to constant Objects

Consider the statement

const int* pi; //it is same as int const *pi
It defines pi as a pointer to a constant integer. Let pi be initialized by the statement
int i[20];
pi=i;
i.e" *pi would refer to the integer[0].

Due to the definition of pi statements such as
*pi = 10; or pi [10]= 20;
are invalid. It results in compile time errors. But pi itself can be changed. i.e., a statement such as pi++; is perfectly valid.

2.7.4 Constant Pointers

The statement

int* cost pi =&i:
defines a constant pointer to an integer (assume that i an integer array), In this case, the use of a statement such as

*pi = 10;
is perfectly vaiid, but others that modify the pointer, such as

pi++;
are invalid and result in compile time errors.

Value addition: Interesting Fact
Heading text * and & operator
Body text: Each of the symbols & and * have two distinct roles in C++. • As a prefix, the & symbol is used for the reference operator on objects. For example, &n; • As a suffix, it is used to designate the "reference to" derived type. For example, void swap (int &, int &); • As a prefix, the * symbol is used for the dereference operator on pointers. For example, *p;

- As a suffix, it is used to designate the "pointer to" derived type
  For example,
  int* p;

Summary

- A pointer must be defined so as to point to a particular data type. For example, in the case of int* ptr; Here, *ptr* can hold the address of an integer variable.

- The ampersand & is a unary operator used to denote the address of a variable. For example

  ```
  int a;
  int* b;
  b = &a;
  ```

  results in variable a and pointer b referring to the same memory location. The value of *&a* is said to **'point to a.'**

- The unary operator * can be thought of as the inverse of &, and should be read as **'the object pointed to by'**. It is known as the **de-referencing operator**. For example, **(&a)* is equivalent to variable 'a' since it is the object pointed to by the address of 'a', in other words, 'a' itself. The expression *&a* is not a variable so a value cannot be assigned to it. Thus, an object in memory can be denoted in more than one way. For example,

  ```
  int*p;
  int a;
  *p=20; or a = 20;
  ```

  This is because *p is same as *&a i.e. a.

- A pointer variable can be assigned the address of ordinary variable. A pointer variable can be assigned the value of another pointer variable. A pointer variable can be assigned a null. An integer quantity can be added to or subtracted from a pointer variable. One pointer variable can be subtracted from another provided both pointers point to elements of same array. Two pointer variable can be compared provided both pointers point to object of same data type. Other arithmetic operations on pointers are not allowed.

- The values can be exchanged between two pointers by assigning their addresses or their values.

- Void Pointers do not have any type associated with them and can hold the address of any type of variable.

- A pointer is a variable whose value either is 0 or is the address of some other variable or object. If its value is 0, we say the pointer is **null**. Otherwise, we say that the pointer points to the variable or object whose address it stores. An uninitialized pointer is called a **dangling pointer**.

- A pointer becomes a **Wild Pointer** when it is pointing to an unallocated memory or when it is pointing to a data item whose memory is already released. Side effects of such pointers are creation of **garbage memory and dangling reference**.

- Pointers are often passed to a function as argument by **reference**. When an argument is passed is passed by reference, the address of a data item is passed to a function. Any change that is made to the data item will be recognized in both the function and calling routine.

- When an object is passed to a function that should not change it, it should be **passed by constant reference**. This has the advantage of preventing unintentional changes to the argument without the disadvantage of duplicating it.

- The address of array element (i+1) can be expressed as either &x[i] or as (x+i). Thus we have two different ways to write the address of any array element:
    - we can write actual array element, preceded by an ampersand; or
    - we can write an expression in which subscript is added to the array name.

- Since, the name of an array indicates the starting address of the array, the expressions arrayname [i] and (arrayname+i) point to the ith row of the array. Therefore, * (arrayname+ i) +j points to the jth element in the ith row of the array. The subscript j actually acts as an offset to the base address of the ith row. The two dimensional array elements can also be accessed by using the notation arrayname[i][j].

- As a pointer variable always contains an address, an **array of pointers** is a collection of addresses. An array of pointers is useful for holding a pointer to a list of strings. They can be utilized in implementing algorithms involving excessive data movements.

- A double pointer can be declared as a pointer of a pointer. This pointer can be declared by using ** operator.

- The allocation of memory space for data storage during the course of program execution is called dynamic memory allocation. The new operator allocates a specified amount of memory during runtime and returns a pointer to that memory location. It returns NULL, if memory allocation is unsuccessful. Delete operator is used to return the memory allocated by the new operator back to the memory pool.

Exercise

Q 9.1 Under what situations, the use of pointers is indispensable?

Q 9.2 What is a pointer?

Q 9.3 What is the difference between a dangling pointer and a null pointer?

Q 9.4 What is a reference? Why it must always be initialized?

Q 9.5 What's wrong with the following declaration?

```
int &r=44;
```

Q 9.6 What are the two uses of the & symbol in C++?

Q 9.7 What are the two uses of the * symbol in C++?

Q 9.8 What does the reference operator and a dereference operator do?

Q 9.9 What is the difference between passing an argument by reference and passing it by a constant reference?

Q 9.10 What are the different arithmetic operations that can be performed on pointer variables?

Q 9.11 Consider the following definitions:

```
int *a, *b, c; float *e; char *p;
```

The pointer variables a, b, and c are initially pointing to memory locations 100, 150, and 50 (assume) respectively. What is the address stored in the pointer variable (a, b, and c) on execution of the following statements?

```
a++;
b = --a;
cout « *b++;
cout « *++p;
e++;
a = &c;
```

Q 9.12 Write a program for sorting names of persons by swapping pointers instead of data. Use Bubble sort algorithm for sorting.

Q 9.13 Consider the function show ( ), which is defined as follows:

```
void show( int a, int b, int c)
        {
           cout «a<< " " << b « " " « c;
        }
        int*i,j;
        i= &j;
        j= 2;
        int k[]= { 1, 2, 3 };
```

What is the output of the following statements:
(Note that actual parameters are evaluated from right to left

while assigning them to formal parameters)

```
show( *i, j, *k);
show ( *i, *i++, *i);
show( *k, *k++, *k++ );
```

Q 9.14 What are the differences between pointers to constants and constant pointers? Give examples.

Q 9.15 Define the following:
(a) Wild pointers
(b) Garbage
(c) Dangling reference.
Consider the following program:

```
#include <iostream.h>
void main()
{
int * a;
const int *b;
int *const p;
int c = 2, d = 3;
cout «a; b = &c;
P = &d; *b = 10;
b = new int[10];
int *b = 10;
delete b;
cout « *b;
a = new int[10];
a[9] = 20;
a[10] = 30;
a = new int[5];
a++;
 ++b;
cout « *a;
}
```

Observe the above program carefully and find out where all garbage, dangling reference, and wild pointers exist. Identify statements which are treated as errorneous by the compiler.

Q 9.16 Fill the last column of the table

Pointer variable	Pointer values	Pointer increment	Pointer value after increment
char*a;	10	a++;/++a;a=a+3;	
int*b;	10	b++;/++b; b=b+2;	
long*c;	10	c++;/++c; c=c+3;	
float*d;	10	d++;/++d; d=d+2;	
double*e;	10	e++;/++e; e=e+2;	

Q 9.17 Are the following declarations same
```
char *p;
char a[20];
```

Q 9.18 Which operator finds the L value of a variable.

Q 9.19 Write a program for multiplication of two matrices using pointers.

Q 9.20 Explain the constructs supported by C++ for runtime memory management. Write an interactive program processing marks of students using C++ memory management operators.

Q 9.21 Explain syntax for defining pointers to functions. Write a program which supports the following:
```
a = compute( sin, 1.345)
b = compute( log, 150)
c = compute( sqrt, 4.0)
```

Glossary

Address operator "&": a unary operator. The expression &variable name evaluates the address of its operand.

Array of Pointers: As a pointer variable always contains an address, an array of pointers is a collection of addresses.

Constant Reference: When an object is passed to a function that should not change it, it should be passed by constant reference. This has the advantage of preventing unintentional changes to the argument without the disadvantage of duplicating it.

Dangling pointer: A pointer that has not been initialized or has been de-allocated with the delete operator.

Delete Operator: This operator is used to return the memory allocated by the new operator back to the memory pool.

Dereferencing: It is the process of accessing and manipulating data stored in the memory location pointed to by a pointer.

Dynamic memory allocation: The allocation of memory space for data storage during the course of program execution is called dynamic memory allocation.

Garbage Memory: The memory becomes garbage memory when a pointer pointing to a memory object (data item) is lost; i.e., it indicates that the memory item continues to exist, but the pointer to it is lost; it happens when memory is not released explicitly.

Indirection operator "*": a unary operator which is used to dereference pointers.

New Operator: It allocates a specified amount of memory during runtime and returns a pointer to that memory location. It returns NULL, if memory allocation is unsuccessful.

Null pointer: A pointer whose value is 0.

Pointer: a variable whose value is the memory address of an object.

Reference: A reference is a synonym for an existing object.

Void pointer: These pointers do not have any type associated with them and can hold the address of any type of variable. The format for declaring a void pointer is as void *v_ptr;

Wild Pointer: A pointer becomes a Wild Pointer when it is pointing to an unallocated memory or when it is pointing to a data item whose memory is already released.

References

Work Cited
1. R.G. Dromey, How to solve it by Computer, Pearson Education
2. B. A. Forouzan and R. F. Gilberg, Computer Science, A Structured Approach using C++, Cengage Learning, 2004.

Suggested Reading
1. S.S. Khandara, Programming in C, C++, S.Chand
2. John R. Hubbard, Data Structure with C++, Tata Mc Graw Hill

Web links
1. Wekepedia.com
2. Google.com