CHAPTER 13

ADDRESS, POINTER VARIABLE, DYNAMIC MEMORY ALLOCATION

A computer's memory is divided into slots. Each slot has an address that can hold a value. A memory that holds a value and has the ability to change it when necessary is known as a variable. There are variables that hold values such as integers, characters, or other data types. A variable that holds an address of another variable is known as a pointer variable or simply as a pointer. Pointers provide open access to memory addresses and are beneficial to programmers, but pointers can put the security of a system at risk. For this reason, pointers are a controversial topic. The storage for pointer variables can be allocated during run time (dynamic allocation), as the program requires the memory. The memory for a dynamically allocated variable can be freed after its usage. This allows availability of the storage for other variables. The allocation and de-allocation of memory leads to a great saving of computer memory and a programmer can build their own data structures and manipulate them as desired. Furthermore, this provides other alternatives in building data structures.

ADDRESS OF A VARIABLE

Every variable has an address and a value. Normally when a variable is declared, an address is assigned to the variable by the system (compiler). The address of a variable is accessed by an ampersand **&;** it is known as an ***address operator***.

```
#include <iostream.h>
void main(){
        int x;
        cout <<"ADDRESS OF X IS: "<<&x<<endl;
        x=5;
        cout<<"X IS: "<<x<<endl;
      . }//MAIN
```

Figure 13.1a – Program to show the address of a variable

```
ADDRESS OF X IS: 0x74272400
X IS: 5
```

Figure 13.1b – Output of Figure 13.1a

A POINTER VARIABLE

A regular variable holds a value such as an integer. A pointer variable, or simply a pointer, holds a value as well but its value is the address of another variable. Through this address, the value of the variable can be accessed. In other words, a pointer can indirec... access the value of the variable. An asterisk (*) before the variable name is used declare a pointer variable.

```
#include <iostream.h>
void main(){
        int *x;
        int y;
        x=&y;
        y=5;
        cout<<"X IS: "<<x<<endl;
        cout<<"THE ADDRESS OF Y IS: "<<&y<<endl;
        cout<<"Y IS: "<<y<<endl;
        cout<<"THE CONTENT OF X IS: "<<*x<<endl;
        }//MAIN
```

Figure 13.2a – Address and content of pointer variable

```
X IS: 0x0065FDF0
THE ADDRESS OF Y IS: 0x0065FDF0
Y IS: 5
THE CONTENT OF X IS: 5
```

Figure 13.2b – Output of Figure 13.2a

DECLARING A POINTER VARIABLE

The method used to declare a pointer variable is similar to an ordinary variable except that an asterisk (*) is placed before the variable name. A pointer variable can be declared of a primitive type such as int, char, float, double, as well as other structured types such as an array, structure, or class. In the following example, x is a pointer variable that points to an integer value and, similarly, y is a pointer variable that points to a character value.

```
#include <iostream.h>
void main(){
        int *x;
        char *y;
        int A=5;
        char B='m';
        x=&A;
        y=&B;
        cout<<x<<" IS ADDRESS OF A CONTAINING INTEGER: "<<A<<endl;
        cout<<y<<" IS ADDRESS OF B CONTAINING CHARACTER: "<<B<<endl;
        }//MAIN
```

Figure 13.3a – Declaration of a pointer

```
0x0065FDEC IS ADDRESS OF A CONTAINING INTEGER: 5
m╠╠╠□ IS ADDRESS OF B CONTAINING CHARACTER: m
```

Figure 13.3b – Output of Figure 13.3a

ACCESSING THE CONTENT OF WHAT A POINTER POINTS TO: INDIRECTION OPERATOR

Accessing the content of what a pointer points to is similar to the way the value of a regular variable is accessed with one exception: the usage of an asterisk before the pointer variable. This indirect access of the variable value is known as *indirection* and the asterisk (*) before the variable name is known as the ***indirection operator*** *or* ***de-reference operator.***

```
#include<iostream.h>
void main(){
        int *x;
        char *y;
        int A=5;
        char B='m';
        x=&A;
        y=&B;
        cout<<x<<" IS ADDRESS OF X CONTAINING:  "<<*x<<endl;
        cout<<y<<" IS ADDRESS OF Y CONTAINING:  "<<*y<<endl;
        }//MAIN
```

Figure 13.4a – Use of the indirection operator

```
0x124f241e IS ADDRESS OF X CONTAINING:  5
m□ IS ADDRESS OF Y CONTAINING:  m
```

Figure 13.4b – Output of Figure 13.4a

ACCESSING POINTER VALUE: DE-REFERENCING

In the following program, a pointer variable is declared as **int *x;** and later in the program, the address of the variable **y** is assigned to x with the following statement: **x=&y**. Any change to the value of the variable **y,** such as: **y=y+2,** is accessible by pointer variable **x**. Accessing what the pointer is pointing to, ***x,** is called de-referencing the pointer or simply de-referencing. Similarly, the value of **y** can be changed by changing the content of the pointer **x** in the statement: ***x= *x+1.**

```
#include<iostream.h>
void main(){
        int *x;
        int y;
        y=6;
        x=&y;
        y=y+2;
        *x=*x+1;
        cout<<"X VALUE IS: " <<*x<<endl;
        cout<<"Y IS: "<<y<<endl;
        }//MAIN
```

Figure 13.5a – De-referencing a pointer

```
X VALUE IS: 9
Y IS: 9
```

Figure 13.5b – Output of Figure 13.5a

PASS BY POINTER: SWAP FUNCTION

In C language (not C++), ***all*** parameters are passed ***by value***. This means that only the value of a variable is sent to the called function; and any change to this variable's value in the called function has no impact on the original variable. However, passing an address as a value enables a function to change the content of that address (indirectly). Any change to the content of the address impacts the original value of the variable. Therefore, ***pass by pointer*** is necessary in C whenever a function needs to change the value of its original variable. For example, the following swap function exchanges the content of two variables.

```
#include <iostream.h>
void swap(int *x, int *y){
        int temp;
        temp=*x;
        *x=*y;
        *y=temp;  }//SWAP
void main(){
        int x=5;
        int y=6;
        swap(&x,&y);
        cout<<"X IS: "<<x<<endl;
        cout<<"Y IS: "<<y<<endl;
        }//MAIN
```

Figure 13.6a – Swap function using pointers

```
X IS: 6
Y IS: 5
```

Figure 13.6b – Output of Figure 13.6a

REFERENCE VARIABLE: ALIAS VARIABLE

Like a pointer variable, a reference variable contains an address but a reference variable creates an alias of a regular variable (the same address but different names). An ampersand (&) before the variable is used to declare a reference variable and a reference variable must be initialized when it is declared. Once a reference variable is associated with another variable within a program or function, you cannot reference to another variable. Note that the difference between a pointer and a reference variable is that a pointer can have different addresses. A reference variable can have only one address but may have a different name for the same address.

```
#include <iostream.h>
void main(){
        int x=3;
        int &r =x;
        cout<<r<<" IS AN ALIAS VALUE FOR X: "<<x<<endl;
        r=5;
        cout<<" X IS: "<<r<<endl;
        }//MAIN
```

Figure 13.7a – Program using a reference variable

```
3 IS AN ALIAS VALUE FOR X: 3
 X IS: 5
```

Figure 13.7b – Output of Figure 13.7a

PASS BY REFERENCE

In C++, a function can change the value of the original variable if the variable is passed as a reference variable. By default all variables are passed to a function by value. The use of **&** in the function definition indicates that the variable is declared as a reference variable and that any change to the parameter impacts its associated (alias) variable. For example, in the following program variable **x** is associated with reference variable **z** and similarly the variable **y** is associated with **w**; therefore, any change to the **z** and **w** will impact **x** and **y** accordingly.

```
#include<iostream.h>
void swap(int &z, int &w){
        int temp;
        temp=z;
        z=w;
        w=temp;     }//SWAP
void main(){
        int x=5;
        int y=6;
        swap(x,y);
        cout<<"X IS: " <<x<<endl;
        cout<<"Y IS: "<<y<<endl;
        }//MAIN
```

Figure 13.8a – Swap function using pass by reference

```
X IS: 6
Y IS: 5
```

Figure 13.8b – Output of Figure 13.8a

ARRAYS AND POINTERS

The name of an array is an address (constant address) that points to the first element of the array. Therefore, a subscript (index) such as **n** will point to the $\mathbf{n}^{th}$ element of the array. For example, in array **int x[]={2,4,6,8}**, the element **x[3]** contains **8**; therefore, **x+3** points to the same element which is **8**. Similarly, the component **x[0]** contains **2** meaning x points to the first element which is 2. In order to access the content of what a pointer is pointing to, we use * indirection operator (de-referencing operator); therefore, **x[3]** is the same as ***(x+3)**. Note that you cannot change the address of an array. In other words, you cannot use the name of an array as an ***lvalue*** (left of the assignment) because it is a constant pointer.

```
#include <iostream.h>
void main(){
        int x[ ]={2,4,6,8,9};
        cout<<"THE VALUE OF x[3] IS: " <<x[3]<<endl;
        cout<<"THE VALUE OF *(x+3) IS:"<<*(x+3)<<endl;
        }//MAIN
```

Figure 13.9a – Program using arrays and pointers

```
THE VALUE OF x[3] IS: 8
THE VALUE OF *(x+3) IS:8
```

Figure 13.9b – Output of Figure 13.9a

POINTER ARITHMETIC

Computer memory is structured in a sequential way. By incrementing the address by one, the next memory address is accessible and similarly by decrementing by one the previous address is accessible. If a pointer contains an address, it is possible to add or subtract in order to access other addresses. Pointer arithmetic allows access through the specific address. In the following program, **p** points to the array **x** and through **p** we can access the contents of array **x** and go backwards and forwards by adding or subtracting to the pointer **p**. It is important to note that with a pointer you can only add or subtract but not divide or multiply. Moreover, pointer arithmetic is relative to its base type and is handled by the compiler; the programmer doesn't have to know if the integer data type takes two or four bytes of memory.

```
#include <iostream.h>
void main(){
        int x[ ]={2,4,6,8,9};
        int *p=x;
        p=p+1;
        cout<<"THE VALUE OF p IS:"<<*p<<endl;
        p=p-1;
        cout<<"THE VALUE OF p IS:"<<*p<<endl;
        }//MAIN
```

Figure 13.10a – Program demonstrating pointer arithmetic

```
THE VALUE OF p IS:4
THE VALUE OF p IS:2
```

Figure 13.10b – Output of Figure 13.10a

TRAVERSE AN ARRAY BY ITS POINTER

The use of a pointer allows access to an array's element more efficient than a normally indexed array. Accessing elements of an array by its pointer can be done by adding the index to the array name or by incrementing the pointer. In fact, the compiler will convert an array to its pointer address and the name of the array is a pointer to the first element of the array. At later time, you will learn how to allocate the memory for an array dynamically (at run time) rather than at the compile time (statically).

```
#include <iostream.h>
void main(){
        int x[]={2,4,6,8,0};
        int i=0;
        while (*(x+i)!=0){
                cout<<"*(x+i)= "<<*(x+i)<<endl;
                i++; }//WHILE
        }//MAIN
```

Figure 13.11a – Traversal of an array by its pointer

```
*(x+i)= 2
*(x+i)= 4
*(x+i)= 6
*(x+i)= 8
```

Figure 13.11b – Output of Figure 13.11a

TRAVERSING ARRAYS BY ANOTHER ARRAY POINTER

The name of an array is a constant address (base address), it points to the first element of the array. The array can be accessed by its name and by adding the offset (index). Note: we cannot increment the pointer name because it is constant. However, by assigning an array of pointers to the name of the array, you can increment the pointer by one and access the values of the array. In the following example, the variable **s** contains the address of array **x** and **s** is incremented each time by one.

```
#include<iostream.h>
void traversebypointer(int *s){
        while(*s!=0){cout <<*s<<" ";
        s=s+1; }//WHILE
        }//TRAVERSE BY POINTER
void main(){
        int x[]={2,4,6,8,0};
        traversebypointer(x);   }//MAIN
```

Figure 13.12a – Traversal of an array using pointers

```
2 4 6 8
```

Figure 13.12b – Output of Figure 13.12a

POINTER COMPARISON

Two pointers can be compared to each other using relational operators such as = = , != , >, or <. You may want to compare two pointers to see if one reaches the other. Furthermore, a pointer can be checked for underflow or overflow. Finally, pointer comparison works hand-in-hand with pointer arithmetic (addition and subtraction).

```
#include <iostream.h>
void main(){
        int x[10], *p, *q;
        p=&x[9]; //ADDRESS OF LAST
        q=x; //ADDRESS OF FIRST
        while (q<p) q++;
        cout<<"q: "<<q<<" HAS THE SAME ADDRESS AS p: "<<p<<endl;
        }//MAIN
```

Figure 13.13a – Program using pointer comparison

```
q: 0x0065FDF4 HAS THE SAME ADDRESS AS p: 0x0065FDF4
```

Figure 13.13b – Output of Figure 13.13a

DYNAMIC ALLOCATION OF MEMORY: new

The keyword **new** is used to allocate a memory location for a pointer variable. Until now, the compiler has provided the storage for a variable upon declaration of the variable and a pointer has used the storage of the other variable (like a mushroom, using others storage). How can a pointer have its own storage? The answer is the keyword **new** followed by a data type such as **p=new int**. The keyword **new** enables the programmer to request memory as it is needed. In the following example, three separate storage locations have been allocated during run time. The three pointer variables are declared as integer and later each is assigned the necessary storage. Would there be an equivalent to this program if there were no pointers and no dynamic memory allocation?

```
#include<iostream.h>
void main(){
        int  *x, *y, *z;
        cout<<"ENTER THE FIRST NUMBER: ";
        x=new  int;
        cin>>*x;
        cout<<"ENTER THE SECOND NUMBER:";
        y=new  int;
        cin>>*y;
        z=new int;
        *z= *x + *y;
        cout<<"THE SUM IS: "<<*z<<endl;
        }//MAIN
```

Figure 13.14a – Program using the keyword new to allocate memory

```
ENTER THE FIRST NUMBER: 5
ENTER THE SECOND NUMBER:10
THE SUM IS: 15
```

Figure 13.14b – Output of Figure 13.14a

DE-ALLOCATION OF MEMORY: delete

When the allocated memory is not needed anymore, it can be freed so that it is available for future use. To de-allocate the memory the keyword delete is used followed by the pointer variable such as **delete p;**. The following program finds the maximum of two numbers using pointer variables instead of regular variables. The memory is allocated as the number is entered (run time) and the memory is released (deleted) when there is no further need for the memory.

```
#include <iostream.h>
void main(){
        int  *x, *y, *max;
        cout<<" ENTER THE FIRST NUMBER: ";
        x=new  int;
        cin>>*x;
        cout<<"ENTER THE SECOND NUMBER:";
        y=new  int;
        cin>>*y;
        max=new int;
        if (*x > *y) {*max=*x; delete x; }
        else{ *max=*y; delete y; }
        cout<<"THE MAXIMUM IS: "<<*max<<endl;
        }//MAIN
```

Figure 13.15a – Program using the keyword delete to de-allocate memory

```
ENTER THE FIRST NUMBER: 10
ENTER THE SECOND NUMBER:20
THE MAXIMUM IS: 20
```

Figure 13.15b – Output of Figure 13.15a

DYNAMIC ALLOCATION OF ARRAY

The memory for an array can be allocated dynamically similar to the allocation of a simple variable by using the keyword **new**. The size of the requested array is indicated in a bracket after its data type.

```
#include <iostream.h>
void main(){
      int *p,*ptr;
      p=new int [5];
      int i;
      cout<<"ENTER ARRAY VALUES: ";
      for (i=0;i<5;i++)cin>>*(p+i);
      cout<<"DISPLAY THE ARRAY BY POINTER: ";
      for( ptr=p; ptr<p+5; ptr++)cout<<*ptr<<" ";
      }//MAIN
```

Figure 13.16a – Program with dynamic allocation of memory

```
ENTER ARRAY VALUES: 1 2 3 4 5
DISPLAY THE ARRAY BY POINTER: 1 2 3 4 5
```

Figure 13.16b – Output of Figure 13.16a

PROBLEM WITH STATIC ARRAYS

One problem with static arrays is that the size of the array must be known ahead of time. What happens if there is more data than the array (insufficient memory) or what happens if a large array is declared but only a few locations are needed (waste of memory)? This problem is resolved by dynamic allocation and de-allocation of memory.

```
#include <iostream.h>
void main(){
        int size;
        cout<<"ENTER THE SIZE OF THE ARRAY: ";
        cin>>size;
        double *p= new double [size];
        cout<<"FILL THE ARRAY WITH "<<size <<" COMPONENTS: ";
        for(double *ptr=p; ptr<p+size; ptr++)cin>>*ptr;
        cout<<"DISPLAY THE ARRAY: ";
        for(int i=0; i<size;i++){cout<<p[i]<<" ";}
}//MAIN
```

Figure 13.17a – Dynamic allocation of memory for an array

```
ENTER THE SIZE OF THE ARRAY: 4
FILL THE ARRAY WITH 4 COMPONENTS: 4 3 2 1
DISPLAY THE ARRAY: 4 3 2 1
```

Figure 13.17b – Output of Figure 13.17a

DE-ALLOCATION OF ARRAY

The keyword **delete** de-allocates the memory for the array that was previously allocated by the keyword **new**. The de-allocation of memory for an array is the same as allocation of memory for a simple data type except that an open and close bracket is used before the variable name. However, there is no need to indicate the size of the array because the program already knows it. The general syntax is: **delete [] pointervariable;.**

```
#include <iostream.h>
void main(){
        int *p=new int [5];
        cout<<"ENTER FIVE INTEGERS: ";
        for(int i=0; i<5;i++) cin>>*(p+i);
        cout<<"DISPLAY: ";
        for(int j=0; j<5;j++) cout<<*(p+j)<<" ";
        delete [] p;
        char *str=new char[5];
        cout<<endl<<"ENTER FIVE CHARACTERS: ";
        for(int k=0; k<5;k++) cin>>*(str+k);
        cout<<"DISPLAY: ";
        for(int n=0; n<5;n++) cout<<*(str+n)<<" ";
        delete [] str;
        }//MAIN
```

Figure 13.18a – De-allocation of array memory

```
ENTER FIVE INTEGERS: 1 2 3 4 5

    DISPLAY: 1 2 3 4 5
    ENTER FIVE CHARACTERS: a b c d e
    DISPLAY: a b c d e
```

Figure 13.18b – Output of Figure 13.18a

NAME OF ARRAY AS A CONSTANT ADDRESS

The name of an array is an address; therefore, by passing the name to a function, any change to the array will impact the content of the original array. In the following program, the name of array **x** is passed as a parameter. Note that in the **readdata( )** function, **&n** is used instead of **n** in order to pass its address.

```
#include<iostream.h>
void readdata(int x[],int* n){
        cout <<"ENTER MAXIMUM OF 5 INTEGERS INTO ARRAY: ";
        *n=0;
        while(*n<5){
                cin>>x[*n];
                (*n)++;} }//READDATA
void printdata (int x[], int n){
        for(int i=0;  i<n; i++) cout<<"x["<<i<<"]="<<x[i]<<endl;
}//PRINTDATA
void main(){
        int x[5],n;
        readdata(x,&n);
        printdata(x,n);
        }//MAIN
```

Figure 13.19a – Name of array passed as a constant address

```
ENTER MAXIMUM OF 5 INTEGERS INTO ARRAY: 1 2 3 4 5
x[0]=1

x[1]=2
x[2]=3
x[3]=4
x[4]=5
```

Figure 13.19b – Output of Figure 13.19a

SORT BY POINTERS

Most sorting techniques require the exchange of data either at the point of the data comparisons (exchange sort) or at the end of each pass (selection sort). However, sorting becomes very slow with a large set of composite data (structures or classes). One solution is to exchange a pointer to the data instead of exchanging the data. For example, rather than exchange ten fields of data only the pointer is exchanged. In a ***pointer sort***, an array of pointers points to each element of the array. Whenever an exchange is necessary, only the pointer's values are swapped, which effectively changes the order of the elements of the array.

POINTER BUBBLE SORT

In the following bubble sort, an array of pointers is associated to the address of the array. Instead of exchanging the data, the pointer addresses are exchanged. The sorted data is accessed through the pointer and the original array remains untouched. Can you imagine if you had to sort the employees by their name and exchange all the fields such as addresses and other personal records?

```
#include <iostream.h>
void main(){
        const int n=5;
        int item[n]={2,5,3,1,8};
        int *p[n];
  int i,j;
        int *temp;
        int sortedflag=0;
         for(i=0;i<n;i++) p[i]=item+i; //INITIALIZING POINTER ARRAY
         for(i=0;i<n;i++)cout<<*p[i]<<" ";
                while (!sortedflag){
                        sortedflag=1;
                        for(j=0;j<n-1;j++ ){
                                if (*p[j]>*p[j+1]){
                                   temp=p[j];
                                        p[j]=p[j+1];
                                        p[j+1]=temp;
                                        sortedflag=0;   }//SWAP
                        }//J
                }//I
        cout<<endl<<"SORTED ARRAY: ";
        for(i=0;i<n;i++)cout<<*p[i]<<" ";
}//MAIN
```

Figure 13.20a – Bubble sort using pointers

```
2 5 3 1 8
SORTED ARRAY: 1 2 3 5 8
```

Figure 13.20b – Output of Figure 13.20a

PROBLEMS WITH POINTERS

POINTER TO A WRONG PLACE: While a pointer can easily be set anywhere, it can also be set to an unwanted address. Moreover, it is possible that a pointer is never initialized but is used.

DANGLING POINTER: What happens if two pointers point to the same storage and the storage for one of the pointers is deleted. Now what is the other pointer pointing to? Obviously the storage is lost and no longer exists. You cannot attempt to modify a de-allocated memory space after the memory is deleted. The result could be disastrous and the program could crash.

NO MORE STORAGE FOR A POINTER: The keyword **new** sets aside the storage for a pointer and assigns its address to the pointer variable. The available storage comes from an area of memory known as ***free store or heap***. What happens if the heap runs out of memory? The compiler can set a NULL address to the pointer variable or an exception can be raised and handled by the programmer with an ***exception handler***. What happens if a pointer is initialized to a NULL value and you try to de-reference the pointer?

TANGLED PROGRAM: Careless use of pointers can result in tangled, poor programming designs where several pointers point to each other here and there. This makes tracking of the program difficult. Tangled pointers raise concerns similar to the historical situation that ***goto statements*** created one time in the programming community.

POINTER INITIALIZATION

Like an ordinary variable, a pointer variable can be initialized to a value (address). The addresses are generated by the system (compiler). Although the addresses are in the form of integer numbers, C++ programmers cannot assign an integer value to a pointer. However, a NULL (zero) is a special value that can be assigned to a pointer; which means it is pointing to nowhere in memory. The NULL is commonly used to initialize a pointer. Setting a pointer to a NULL is the same as setting a regular variable to zero.

```
#include <iostream.h>
void main(){
        int *p=new int;
        if (p==NULL)cout<<" NO MORE STORAGE"<<endl;
        else{ *p=5;
              cout<<" P CONTAINS: "<<*p<<endl;}//ELSE
}//MAIN
```

Figure 13.21a – Program to initialize a pointer if not NULL

```
P CONTAINS: 5
```

Figure 13.21b – Output of Figure 13.21a

POINTERS: COMPACT, POWERFUL, AND ELEGANT, BUT CONFUSING

The use of pointers enables programmers to make compact, elegant and powerful (less typing) programs. However, these features can lead to some confusion among beginner programmers who may not understand them. For instance, the function **strcpy( )** copies one string, namely **t,** to another string, **s**. This enables us to fit the whole **strcpy(** function into one line. If you attempt to write the string copy function using an array and indexes, it may become three times larger than the one written below using pointers to a string.

```
#include<iostream.h>
void strcpy (char *s , char *t){ while (*s++=*t++);}

void main(){
    char *str2 = new char [20];
    strcpy(str2,"Alireza Ebrahimi");
    cout<<"STR2 IS NOW: "<<str2<<endl;
}//MAIN
```

Figure 13.22a – string copy function using pointers

```
STR2 IS NOW: Alireza Ebrahimi
```

Figure 13.22b – Output of Figure 13.22a

SELF REFERENTIAL STRUCTURE

A structure may contain many fields and different data types; however, when one of the fields points to the structure itself, it is known as a self- referential or recursive structure. In the following example, the structure name is node and it contains two fields, one of type integer, known as info, and the second, known as next, pointing to the structure itself. Self-referential structures are used to create dynamic data structures such as linked lists.

```
#include <iostream.h>
struct node {
        int info;
        node *next; };
void main(){
        node *p,*q;
        p=new node;
        (*p).info=8;
        (*p).next=NULL;
        q=new node;
        (*q).info=3;
        (*q).next=p;
        cout<<"q HAS INFO OF: "<<(*q).info<<endl;
        cout<<"p HAS INFO OF: " <<(*(*q).next).info;
        cout<<" ACCESSED BY q"<<endl;
        }//MAIN
```

Figure 13.23a – Program using a self-referential structure to link two nodes

```
q HAS INFO OF: 3
p HAS INFO OF: 8 ACCESSED BY q
```

Figure 13.23b – Output of Figure 13.23a

ACCESS A POINTER FIELD: p->next INSTEAD OF (*p).next

The member access operator **.** (***dot operator***) has a tighter bound (hierarchy) than the ***pointer operator*** * ; therefore, *p.next would be mistakenly evaluated as p.next first and then be redirected. The parenthesis gives priority to the indirection operator. A better shorthand notation of the ***point-to operator*** is to simply use **->**, which is a dash followed by a greater than sign. It does the same job of accessing a pointer member. The above linked list program can be rewritten using p->next instead of (*p).next.

```
#include <iostream.h>
struct node{
        int info;
        node *next; };  //REMEMBER SEMICOLON
void main(){
        node *p,*q;
        p=new node;
        p->info=8;
        p->next=NULL;
        q=new node;
        q->info=3;
        q->next=p;
        cout<<"q HAS INFO OF: "<<q->info<<endl;
        cout<<"p HAS INFO OF: " <<p->info;
        cout<<" ACCESSED BY q"<<endl;
        }//MAIN
```

Figure 13.24a – Linked list program using -> to access pointer members

```
q HAS INFO OF: 3
p HAS INFO OF: 8 ACCESSED BY q
```

Figure 13.24b – Output of Figure 13.24a

LINKED LIST: A HOME MADE ARRAY

To make a linked list, allocate one storage, then allocate another storage and connect (link) these two storages (nodes) together. Then, allocate more storage and link again if necessary. This is the story of a linked list. A list that is created by the programmer one node at a time as the need arises is called a linked list. The keyword **new** allocates either one memory location (storage) or a pre-defined number (array) of storages. To allocate an array of storages, the number (size) of the array must be given at the time when the keyword new is used. What happens if we don't know how much storage we need? The solution is the linked list; get a node and link it to an existing one. The storage for the linked list has at least two fields (parts) and one of them is a pointer that points to the next node (storage). Therefore, in order to build a linked list, we need a ***structure*** data type (struct or class), which allows us to put together more than one field.

```
#include <iostream.h>
struct node{
        int info;
        node *next;};
void main(){
        node *p,*q;
        p=new node;
        p->info=6;
        p->next =NULL;
        cout<<p->info<<endl;
        q= new node;
        q->info=8;
        p->next=q;
        cout<<p->info<<endl<<p->next->info<<endl;
        }// MAIN
```

Figure 13.25a – Simple linked list program

```
6
6
8
```

Figure 13.25b – Output of Figure 13.25a

BASICS OF INSERTION SORT

The algorithm for the insertion sort is to insert incoming data into the right spot. There are three possibilities where to insert incoming data: at the beginning, at the end, or anywhere in the list.

```
#include <iostream.h>
struct node{
        int info;
        node *next;};
void main(){
        node *lst,*p,*q,*t;
        lst=NULL;
        cout<<"ENTER A NUMBER: ";
        int x;
        while(cin>>x){
q=new node;
q->info=x; q->next=NULL;
if (lst==NULL) lst=q;
else if(q->info < lst->info){ q->n e xt=lst;lst=q; }//INSERT BEFORE
```

```
else{           p=lst;
                    t=p;
                    while(p->next!=NULL){
                            if(q->info > p->info){
                                    t=p; p=p->next; }//IF
                            else break;}//WHILE
                                    if(p->next!=NULL){
                                            q->next=p;
                                            t->next =q; }//INSERT IN BETWEEN
                                    else p->next=q; //INSERT AT END;
                             }//ELSE
        cout<<"ENTER A NUMBER: "; }//WHILE
        cout<<"SORTED LIST: ";
        for(t=lst; t!=NULL; t=t->next) cout<<t->info<<" ";
        }//MAIN
```

Figure 13.26a – Insertion sort

```
ENTER A NUMBER: 5
ENTER A NUMBER: 3
ENTER A NUMBER: 8
ENTER A NUMBER: 4
ENTER A NUMBER: 9
ENTER A NUMBER: □
SORTED LIST: 3 4 5 8 9
```

Figure 13.26b – Output of Figure 13.26a

BUILDING DYNAMIC DATA STRUCTURES

Understanding the concept of pointers enables a programmer to build and use dynamic data structures instead of its equivalent counterpart of static data structures. The most common data structures are stacks, queues, trees, and graphs. Other data structures such as sets, symbol tables, and hashing can be implemented by using pointers. Finally, you may want to build your own data structures to fulfill your needs. The next chapter is devoted wholly to the concept of data structures

BUILDING A SIMPLE STACK

A stack is a kind of data structure where data is stored and retrieved in the fashion of last in first out. The following example uses pointers to nodes to implement a stack. This program takes in numbers and pushes them to a stack; when there is no more input, the contents of the stack is displayed.

```
#include <iostream.h>
struct node{
        int info;
        node *next;};
void main(){
        node *stack, *p;
        int x;
        stack = NULL;
        cout<<"ENTER A NUMBER: ";
        while(cin>>x){
                p=new node;
                p->info=x;
                p->next=stack;
                stack=p;
                cout<<"ENTER A NUMBER: "; }//WHILE
        while(stack!=NULL){
                cout<<stack->info<<endl;
                stack=stack->next;              }//WHILE
                }//MAIN
```

Figure 13.27a – A simple stack using pointers

```
ENTER A NUMBER: 5
ENTER A NUMBER: 4
ENTER A NUMBER: 3
ENTER A NUMBER: 2
ENTER A NUMBER: □
2
3
4
5
```

Figure 13.27b – Output of Figure 13.27a

POINTER TO AN OBJECT

Like a pointer to an integer or a char, a pointer can point to an object and manipulate it in the same way as a pointer to any other data type. The keyword **new** is used to create an object dynamically and its counterpart keyword, **delete,** returns the allocated memory back to free store (heap). In the following program, a class is defined as **employee** and a pointer to an object is instantiated from the class ***employee.** The storage for the object is allocated on free store by the keyword **new**. After the task is done with the object, its storage is returned using the keyword **delete**. Recall that at the time of object creation, the class constructor is called to initialize the data members **employee(50,100)**. And the member data such as **hoursworked** and **hourlywage** can be allocated dynamically and then de-allocated. The use of a constructor and a destructor to allocate and de-allocate

storage for the member data is useful. The allocation and de-allocation of member data is not shown in the program below.

```
#include<iostream.h>
class employee{
        int hoursworked;
        double hourlywage;
        double grosspay;
        public:
                employee (int hw, double hr){ hoursworked =hw; hourlywage=hr; }
                void computegross(){ grosspay=hoursworked*hourlywage; }
                void display(){ cout<<"GROSS PAY IS: $"<<grosspay<<endl;} };
void main(){
        employee *ebrahimi;
        ebrahimi= new employee(500, 5.00);
        ebrahimi->computegross();
        ebrahimi->display();
        }//MAIN
```

Figure 13.28a – Program using a pointer to an object

```
GROSS PAY IS: $2500
```

Figure 13.28b – Output of Figure 13.28a

C PROGRAMMER AND POINTERS

C programmers begin to learn about pointers in the early stages of learning and programming more than any other high level languages, even C++. The basic input routine of C, such as **scanf ("%d",&x);**, requires the address of a variable. In situations where an array is used since the name of array is an address the ampersand is not required such as in the following example: **char name[10]; scanf("%s",name);**

C LANGUGE POINTERS AND DIFFERENCES WITH C++

To allocate memory dynamically, C uses the **malloc( )** function, which is equivalent to the **new** operator of C++. Similarly, to free a storage, the function **free( )** is used, which is the equivalent of the **delete** operator of C++. The general form of memory allocation in C is as follows: **void * malloc**(numberofbytes); this function returns a pointer to the new memory; if there is no available memory, it returns null. The general form for freeing memory is: **void free** (*p); . Note that in C, a void pointer is automatically converted to the type of the pointer on the left side of the assignment. To ensure system portability, instead of number of bytes for integer or double, you might want to use the sizeof( ) routine to do the job for you. The sizeof( ) function is important when allocating

memory for a user defined type; it determines how many bits need to be allocated for a user-defined type like a struct or a class. For example, the following C code allocates ten memory locations of type **float**.

float *p=malloc (10*sizeof(float));
if (p==NULL){printf(" NO MORE MEMORY\n"); exit(1); }//EXCEPTION

```
#include <stdio.h>
#include <stdlib.h>
void main(){
        int *p;
        p=(int*)malloc(sizeof(int));
        *p=5;
        printf("%d\n",*p);
        free(p);
        }//MAIN
```

Figure 13.29a – C program to allocate and de-allocate memory

```
5
```

Figure 13.29b – Output of Figure 13.29a

PROGRAMMING LANGUAGES AND POINTERS

In the early stages of computers, all programmers had to work directly with memory addresses (physical addresses) and manipulate the addresses to complete a task. Today, operating systems and compilers provide facilities that free programmers from dealing directly with memory. Computer languages are divided into two categories: low level and high level. The division between them depends on how closely related the language is to the computer's physical entities such as addresses and their means of access. Low-level machines and assembly languages are heavily pointer oriented, thereby enabling programs to manipulate computer components with their addresses. The early high-level computer languages, such as Fortran and Cobol, did not include the concept of the pointer and distanced themselves from directly dealing with memory addresses. Later on, the concept of the pointer was added to languages such as PL/I and Pascal. The C language, as a hybrid language, integrated pointers into the language by implementing an array as a pointer, as well as other low-level bit-wise operations such as and (&), or (|), and left shift (<<) operators. Whether or not to include pointers in a language is the subject of much debate. Some argue that their inclusion leads to security leaks, makes programs vulnerable to bugs, and creates instability by allowing programmers to have free access to memory. As a result, some of the new languages like Java do not include pointer

variables in the language design but instead have some built-in routines that do the same job.

HACKERS LOVE POINTERS

A pointer variable can point anywhere, possibly at the beginning of array or at the end of a file. Pointers can connect two lists or point to another part of the memory. Pointers can be created while a program is running, and, similarly, can be eliminated. This pointer manipulation gives hackers a key to what they want to do. Since every component has an address and every value is in a memory location, memory can be accessible by the addresses and by manipulation of the pointers. Pointers give programmers the power to create complex data structures. One other reason hackers are attracted to pointers is that they can write programs with pointers that no one else can easily figure out and not know what is happening. Finally, hackers want to break into systems but they do not want to leave any footsteps behind.

COMPLICATED POINTERS

Pointers are more complicated than just pointing to a simple data type such as integer, and structured data types such as arrays and classes. A pointer can point to another pointer, to a function, or be the return value of a function. With an array of pointers, each component can point to its designated data type. You can have a pointer to a constant or a constant pointer (instead of a variable pointer). You can convert one pointer type to another implicitly or explicitly. The address 0x0 (hexadecimal zero) is called the NULL pointer and constant 0 (zero) can be used to initialize a pointer to NULL. The following two statements initialize a pointer variable to NULL.

```
int *p=NULL;
int *p= 0;
```

TOPICS NOT COVERED HERE

The topic of pointers opens a new chapter on how to represent data and how to solve problems. It is a different look and view than otherwise would be created by simulation. Exploring dynamic data structures can be several book volumes long and there is ongoing research and new findings. Data structures cover topics starting with array and linked list and defining abstract data types such as stacks and queues, and later building trees and graphs. Looking for better algorithms and/or optimizing a current algorithm is a major concern in the field of data structures. In fact, there are courses in undergraduate and graduate schools that cover the topics of data structures and algorithms and many Ph.D. dissertations are focused on this topic.

CLOSING REMARKS AND LOOKING AHEAD

A pointer is just a variable that holds a value but its value happens to be an address, such as the address of another variable. The unary operator **&** is used to extract the address of a variable; ***** is used to declare a pointer and also is used as an ***indirection operator*** or de-referencing operator to access the value of what a pointer is pointing to. The keyword **new** dynamically allocates storage for a pointer. An array can be allocated dynamically, eliminating the need for declaring the size of an array ahead of time, which leads to memory waste (too much) or insufficient memory (too little). The keyword **delete** returns the allocated memory back for reuse and avoids shortage of memory. The arithmetic operations of addition and subtraction are used to increment and decrement pointers. The use of pointers introduces new approaches to solving problems. For instance, ***pointer sort*** optimizes the exchange sort by only exchanging the pointers rather than a large amount of data each time. In addition, dynamic data allocation gives alternatives to what could only be done statically now can be done dynamically. Having more than one choice enables programmers to decide which one of the two to use in a particular situation. Moreover, dynamic allocation of memory makes data structures flexible but more complex and was not possible otherwise. In the next chapter we will examine the concept of data structures with a variety of abstract data types or simply ADT. With known functionality, ADT become a tool in problem solving. Picking the right one leads to an efficient design and prevents reinvention of what is already tested and there.