POINTERS:

```
main() {
int x , y ;
x= 10 ;
y= 6 ;
…
}
```

- For the small code above, we declared 2 integer type identifiers or variables.
- The C++ compiler, allocates a memory cell for the variable x and another for the variable y.
- To display the contents of the memory cell x, we write:
  Cout << "Value of x = " << x << endl ;

- How does Compiler form identifiers?

- To understand the underlying design, we have to understand how memory is addressed.
- What is the smallest built-in data type that we can declare in C++?
  - Int, char, float, or double ?
- Answer: Byte
- How many bits does a byte have ? 8-bits.
- Hence, the bits are grouped as 8-bytes, and the minimum addressable or declarable unit is a CHARACTER.

Now, let's think of memory as consecutive 1-dimensional array structure!

```
 -------------------------------------------------------------------------------
|   |         |         |         |         |         |         |         | ….
-------------------------------------------------------------------------------
Byte 0    1        2         3        4         5        6          7    ….
```

Usually an integer data type holds 4 bytes in 32-bit systems.
How does the machine address an integer type?
Available 4 bytes are grouped to form a memory location for an integer data type.

Well, when we declare an integer type, the machine will use 4-bytes. It may for example, allocate the first 4 bytes (0-3 bytes) to the variable declared as x, and bytes 4-7 may be allocated to variable y.

When we write:

```
x = 10 ;      //  the value 10 is copied into the location of  (0-3) bytes.
y = 5 ;       //   the value 5 is copied into the location of   (4-7) bytes.
```

And, when we have a statement, such as   cout << x << “ “ << y << endl ;
Compiler prints the contents of bytes 0-3 and bytes 4-7.

So, the variables x and y are interpreted as identifiers of locations.
You can at any given time change the contents of the memory location.
For example, by typing

x= 200 ;   you can change the contents of the identifier x.

What else can we call an identifier? Address !!
Indeed an identifier specifies an address of a memory location.
Since, its type is declared, Compiler knows how many byte should be taken into consideration when the beginning address of a memory location is associated with an identifier.

Conclusion: Compilers interpret Identifiers or Variables of data types as Addresses of memory cells.

Example:

```
// This program uses the & operator to determine a variable's
// address and the sizeof operator to determine its size.
#include <iostream.h>
void main(void){
        int X = 25;
        cout << "The address of X is " << &X << endl;
        cout << "The size of X is " << sizeof(X) << " bytes\n";
        cout << "The value in X is " << X << endl;
}
```

Program Output:

```
The address of X is 0x8f05
The size of X is 2 bytes
The value in X is 25
```

Pointers: are variables that contain memory address as their values. Normally a variable directly refers to a value in a memory cell. A pointer on the other hand, has the address of a variable. In this sense, **a pointer indirectly references a value in a memory cell**.

How to declare a Pointer variable? Similar to variable declarations.

```
int*   P;   or   int  *P ;
```

when declaring more than one pointer, **be careful**

```
int* P, Q ;   // declares only one pointer and one integer variable
int *P, *Q ; // declares both P and Q as pointers.
```

Or

```
Typedef int * ptrType ;
PtrType P, Q;

int   *yptr;
int  y= 5;  // static memory allocation
yptr = & y ;       //  & is the address operator.  The address of y is copied to the
Yptr
cout << “*yptr=” <<*yptr << “yptr = “ << yptr << “&y = “ << &y << “ y=”
<< y <<endl;
```

Program output would look like

```
*yptr= 5   yptr=0xfff0fff2  &y= 0xfff0fff2    y= 5
```

Note: yptr and &y   have the same address value, while *yptr and y has the same value.

Example:

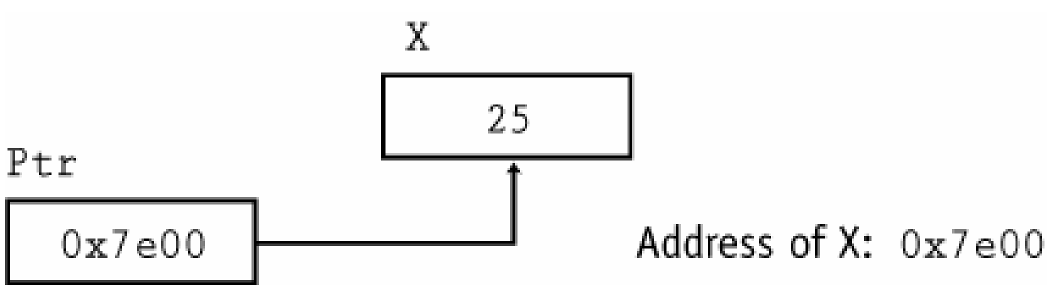
```
// This program stores the address of a variable in a pointer.
#include <iostream.h>

void main(void)
{
      int X = 25;
      int *Ptr;

      Ptr = &X;   // Store the address of X in Ptr
      cout << "The value in X is " << X << endl;
      cout << "The address of X is " << Ptr << endl;
}
```

Program Output:

```
The value in X is 25
The address of X is 0x7e00
```



```
// This program demonstrates the use of the indirection
// operator.
#include <iostream.h>
void main(void)
{
	int X = 25;
	int *Ptr;

	Ptr = &X;   // Store the address of X in Ptr
	cout << "Here is the value in X, printed twice:\n";
	cout << X << "  " << *Ptr << endl;
	*Ptr = 100;
	cout << "Once again, here is the value in X:\n";
	cout << X << "  " << *Ptr << endl;
}
```

Program Output:

```
Here is the value in X, printed twice:
25  25
Once again, here is the value in X:
100  100
```

Example:

```
void main(void){
	int X = 25, Y = 50, Z = 75;
	int *Ptr;
	cout << "Here are the values of X, Y, and Z:\n";
	cout << X << "  " << Y << "  " << Z << endl;
	Ptr = &X;  // Store the address of X in Ptr
	*Ptr *= 2; // Multiply value in X by 2
	Ptr = &Y;  // Store the address of Y in Ptr
	*Ptr *= 2;  // Multiply value in Y by 2
	Ptr = &Z;   // Store the address of Z in Ptr
	*Ptr *= 2;  // Multiply value in Z by 2
	cout << "Once again, here are the values of X, Y, and Z:\n";
	cout << X << "  " << Y << "  " << Z << endl;
}
```

Program Output:

```
Here are the values of X, Y, and Z:
25  50  75
Once again, here are the values of X, Y , and Z:
50  100  150
```

MEMORY ALLOCATION:
There are two kinds of memory allocation schemes:

- STATIC and DYNAMIC

DYNAMIC MEMORY – is divided as Stack and Heap memory. The Stack portion of the dynamic memory is utilized when function calls are made. Clearly, Recursive functions also utilize Stack memory. Heap Memory is utilized when dynamic memory allocation calls are made.

For example:

```
int *P ;
P = new int ;  // allocates a memory cell from heap memory and returns its address
to be copied in P.
*P = 7 ;       // the memory cell whose address is in P  stores the val. 7
```

Assignments:
Like other variables, type compatibility is important.
Ex:

```
int *P, *Q, X;
P = X ;    // incompatible types, X is integer type while P is a pointer,
           // which can contain an integer  type data in the memory cell
           // it points.
*P = X ; // okey
```

Pointers and Arrays

```
int x =1 , y=2 , z[4] ;
int *ip ;                    // ip is a pointer to int

ip = & x;     // ip points to x
y= *ip ;      // y is now 1
*ip =0 ;      //  x is now 0
*ip  += 1   // x is now  1
  z[0] = 5;
  z[1] = 10 ;
  z[2] = 7 ;
  z[3] = 9 ;

ip = &z[0] ;  // ip now points to z[0] not x any more
cout << *ip ;  // results  with   5  and *ip == (ip+0)[0]== ip[0] == z[0]
cout << *(ip +1) ; // yields 10  and  *(ip+1) ==(ip+1)[0]==ip[1]== z[1]
cout << *(ip + 3) ; // yields 9  and   *(ip + 3) == (ip+3)[0]==ip[3]== z[3]
cout << *ip + 2 ;   // yields  5+ 2 , i.e. the output is 7
```

```
ip++ ;
cout << *ip ;  // results  with   10
cout << *(ip +1) ; // yields 7
cout << *(ip + 3) ; // yields  some garbage value.
```

Another example:

```
#include <stream.h>
int pointplay(int * Q) ;  // prototype
main(){
   int *P, A, B;
   A= 5;
   P = &A ;
   cout << "P=" << P << "&P =" << &P << "&A" << &A << endl;
   cout << "*P=" << *P << "A=" << A << endl;
   pointplay(P) ;
   cout << "P=" << P << "&P =" << &P << "&A" << &A << endl;
   cout << "*P=" << *P << "A=" << A << endl;
}
int pointplay(int * Q) {
   cout << "Q=" << Q << "&Q =" << &Q << endl;
   cout << "*Q=" << *Q <<  endl;
   *Q = 10 ;
}
```

Example:

```
// This program shows an array name being dereferenced with the *
// operator.
#include <iostream.h>
void main(void){
      short Numbers[] = {10, 20, 30, 40, 50};
      cout << "The first element of the array is ";
      cout << *Numbers << endl;
}
```

Program Output:

The first element in the array is 10

Example:

```
#include <iostream.h>

void main(void){
      int Numbers[5];
      cout << "Enter five numbers: ";
      for (int Count = 0; Count < 5; Count++)
           cin >> *(Numbers + Count);
      cout << "Here are the numbers you entered:\n";
      for (int Count = 0; Count < 5; Count++)
           cout << *(Numbers + Count)<< " "<<endl;
}
```

Some mathematical operations may be performed on pointers.

- The ++ and – operators may be used to increment or decrement a pointer variable.
- An integer may be added to or subtracted from a pointer variable. This may be performed with the +, - ,+=, or -= operators.

A pointer may be subtracted from another pointer.

```
// This program uses a pointer to display the contents
// of an integer array.
#include <iostream.h>

void main(void){
        int Set[8] = {5, 10, 15, 20, 25, 30, 35, 40};
        int *Nums, Index;
        Nums = Set;
        cout << "The numbers in Set are:\n";
        for (Index = 0; Index < 8; Index++){
          cout << *Nums << " ";
          Nums++;
        }
        cout << "\nThe numbers in Set backwards are:\n";
        for (Index = 0; Index < 8; Index++){
          Nums--;
          cout << *Nums << " ";
        }
}
```

Program Output:

```
The numbers in Set are:
5 10 15 20 25 30 35 40
The numbers in Set backwards are:
40 35 30 25 20 15 10 5
```

PASSING ARRAYS as an argument:

When an array is passed as an argument to a function, the function sees a pointer. Hence, the array passing mechanism acts as pass by reference

Example:

```
#include <iostream.h>
int ArrayIsByReference(int []) ;  //Prototype
void main()  {
int A[4] ;
A[1] = 2; A[2] = 3; A[3] = 4;

ArrayIsByReference(A) ;
Cout << “The value of A[0] << A[0] << endl ;
```

```
Cout << "The Value of *A (=A[0]) is =" << *A << endl ;
}
void ArrayIsByReference (int *Ptr) {
    *Ptr = *(Ptr+1) *    *(Ptr+2) *   *(Ptr+3) ;
  }
```

Note *(A+0) = *(A) = *A = A[0] ;   and *(A+1)=A[1] ;
You can use ++ or – on pointer variables .

Function: **sizeof**

cout << sizeof(char) << sizeof(int) << sizeof(long) << sizeof(double)

1   2   4   4   8 int , long and double is machine depenendable. On pentiums they are 4 bytes. On 386 and 486 machines they are 2 bytes (6 bit).