C++ POINTERS

Pointers

A pointer is an object, whose value refers to (or ***points to***) another value stored elsewhere in the computer memory using its memory address.

C++ pointer is a typed variable, whose stored *value* is the *address* of another variable.

Why Pointers?

They allow access to ***explicit memory locations***, which may be necessary in embedded systems.

They are needed to ***dynamically*** allocate memory for data structures of unknown size, accessed only by pointer, since they will ***not have a name***.

They are necessary to ***connect nodes*** in linked data structures – for example, linked lists.

Pointer Syntax

```
char *p;//declares a char pointer

int *q; //declares an int pointer

float *r;  //declares a float ptr

string *s;//declares a string ptr
```

Pointer Syntax

```
int * p, q;  //only p is a pointer
            //variable;
            //q is an int variable

int *p, *q; //to declare two
      //pointers, attach the *
      //to each variable's name
```

As with other types, C++ does ***not*** automatically ***initialize*** variables.

Pointer variables ***must be initialized*** to `nullptr`, unless a valid address is assigned to them at the moment of declaration.

```
int *p = nullptr; //p points to
                  //nothing (yet)
```

Address Of Operator & returns the address of its operand. This address can be assigned to a pointer variable:

```
int x = 5;
int y = 8;
int *p, *q;//declares two int ptrs
p = &x; //sets p to address of x
q = &y; //sets p to address of y
```

Memory State:

Type	Name	Address	Data
...	...	...	...
int	x	0x12345670	5
int	y	0x12345674	8
int pointer	p	0x12345678	0x12345670
int pointer	q	0x1234567C	0x12345674
...	...	...	...

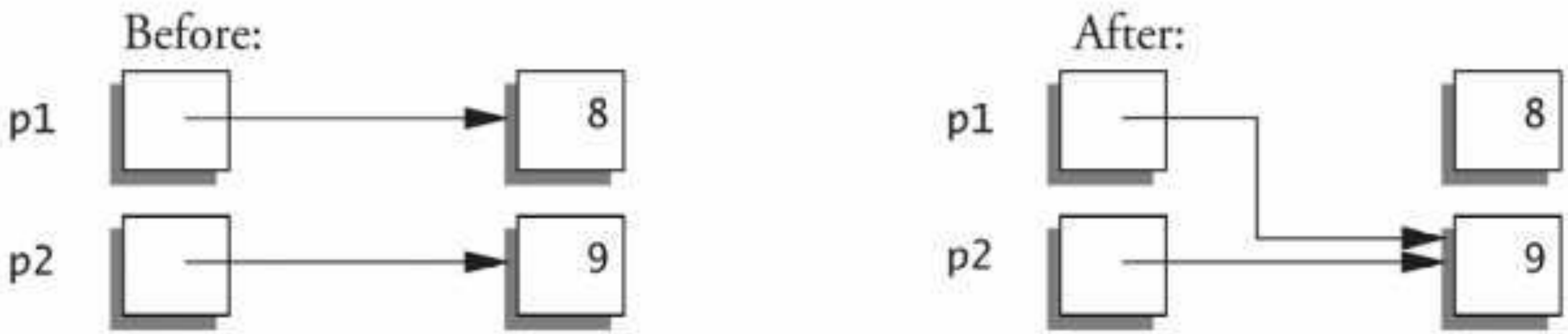
Dereferencing Operator `*` returns the object, to which its operand points.

```
//p points to x, therefore:

cout << *p << endl; // x=5, y=8

y = *p; // x=5, y=5

*p = 4; // x=4, y=5
```

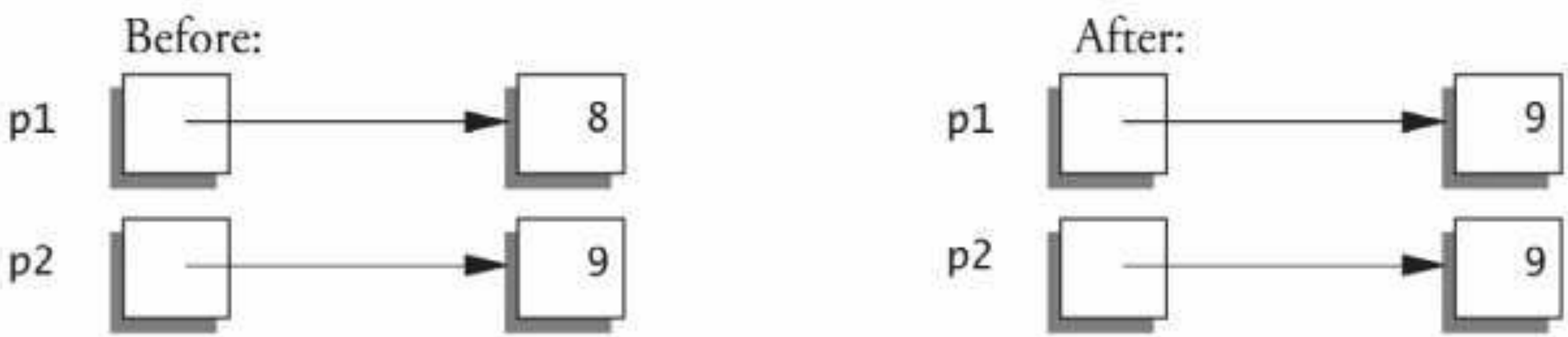
```
int a1 = 8, a2 = 9;
int *p1 = &a1, *p2 = &a2;
```

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




```
*p1 = *p2;
```



```
struct Student
{
    string name;
    int id;
    float gpa;
};

Student student1; //variable of
      //type Student
Student* student1Ptr; //pointer
    //to a variable of type Student
```

```
student1Ptr = &student1; //stores
            //address of student
            //in studentPtr

(*student1Ptr).gpa = 3.5;//stores
            //3.5 in member gpa of
            //student, using . operator

//or use -> member access operator:
student1Ptr->gpa = 3.5; // ->
            //is shorthand for ( *).
```

Dynamic Variables are created at runtime in the memory heap.

These are ***nameless*** variables, and can be only accessed through ***pointers***.

`new` and `delete` operators are used to create and to destroy these dynamic variables.

`new` **operator** allocates nameless memory for a dynamic variable and ***returns a pointer*** to it.

`new` can be used to create both dynamic variables and ***dynamic arrays***:

```
new dataType;//allocates a dynamic
             //variable

new dataType[intValueOrVariable];
  //allocates an array of variables
```

`new` operator

```
int *p = new int; //Creates a
        //nameless variable at
        //runtime on the memory heap
        //and stores its address in p
```

Must ***dereference*** the pointer to access variable:

```
*p = 15;
int x = *p; //dynamic variables
    //cannot be accessed directly
    //by name: they are nameless!
```

`delete` **operator** is needed to avoid **memory leaks**: previously allocated memory that cannot be reallocated. Must ***destroy*** no longer needed dynamic variables to free it up.

```
int *p = new int(5);//initializes
                    //*p to 5

delete p;    //deallocates *p
```

The storage allocated to `*p` is reclaimed.

`delete` operator

...however, the pointer variable `p` ***still exists*** and points to the place in memory that once stored **5**.

`p` is now a ***dangling pointer***, which will cause problems later if dereferenced.

To avoid dangling pointer, set `p` to `nullptr`:

```
p = nullptr; //no dangling pointer
```

Pointer Assignment, Comparison

Value of one pointer variable can be assigned to another pointer of same type.

Two pointer variables of same type can be compared for equality: **`==`** , **`!=`**

However results of relational operators,
`>` , **`>=`** , **`<`** , **`<=`** are ***undefined***.

Pointer Arithmetic is dangerous. It can change values of undefined memory locations ***without warning***.

Address stored in one pointer can be subtracted from that of another to find the ***offset***.

Integer values can be added or subtracted from a pointer variable, ***but not*** multiplied or divided.

Results are different from integer arithmetic: they depend on **`sizeof(variableType)`**.

Dynamic Arrays are created during program execution:

```
int *p;//creates an int pointer

p = new int[6]; //allocates memory
  //for an array of 6 ints and sets
  //p to starting element's address
```

```
*p = 18;  //sets zero element to 18

p++; //points to next array element

*p = 44;  //sets next element to 44
```

Dynamic Array elements can also be accessed with array notation:

```
p[0] = 18; //these commands
p[1] = 44; //have the same result
```

Functions and Pointers

A function can return a value of type pointer:

```
int* testExp(...)
{
    . . .
    //can return a pointer to
    //a dynamic array
}
```

Functions and Pointers

A pointer variable can be passed as a parameter either by value or by reference, in which case **&** is used:

```
void swapPtrs(int* &p, int *q) {
  //can make the caller function's
  //p point elsewhere, but not q
  int *tmp = p;
  p = q;
  q = tmp;
}
```

Functions and Pointers

```
int main() {
   int x = 3;
   int y = 4;
   int *p = &x;
   int *q = &y;
   swapPts(p, q);
   cout << *p << " " << *q << endl;
}

//prints 4 4
```

Pointers vs References

In a function call **&** indicates that an argument is **passed by reference**:

```
foo(int & x); int y = 5; foo(y);
```

The formal parameter is a reference (or alias) to the actual parameter, thus `foo` can change that variable by referring to its memory location using alias `x`.

Though address of a variable is passed it is used to create an alias on the fly. But only a **pointer variable can store this address for future use!**