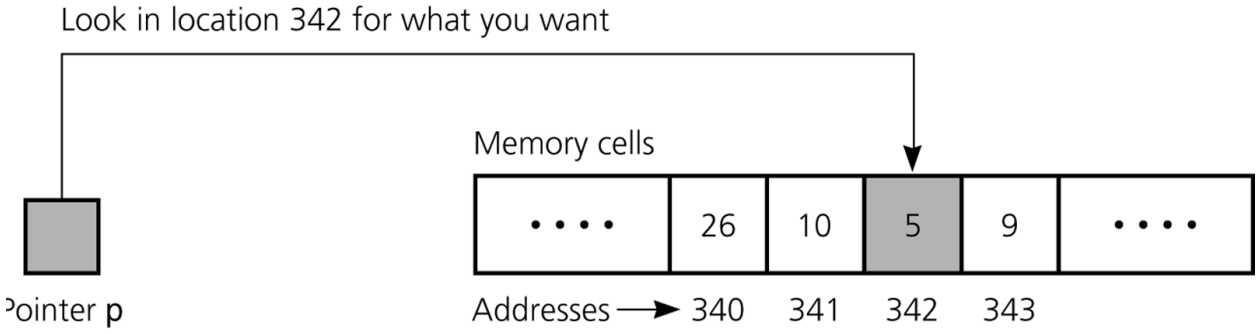
C / C++ Pointers
Dynamic Memory Allocation
Dynamic Arrays

A "pointer" (a.k.a. pointer variable) is a variable whose value represents the runtime machine **address** of some other variable in the program. Pointers allow the easy implementation of **Dynamic Arrays** and **Linked Lists**.



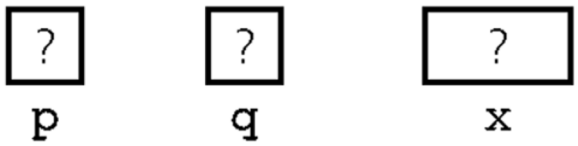
Pointer variables can be used to reference and manipulate the addresses associated with:

- normally created and allocated variables
  - o global variables
    allocated as program begins execution
  - o local variables
    allocated and deallocated as functions are called
- sequential elements within an array
- dynamically allocated items

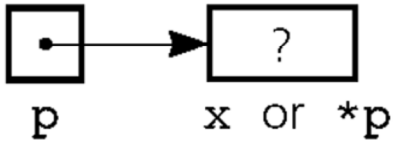
Pointer Usage

(a) declaring pointer variables; (b) pointing to statically allocated memory;
(c) dereferencing a pointer (d) pointing to dynamically allocated memory
(e) dereferencing a pointer (f) copying a pointer

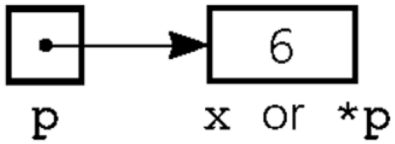
(a)
```
int *p, *q;
int      x;
```



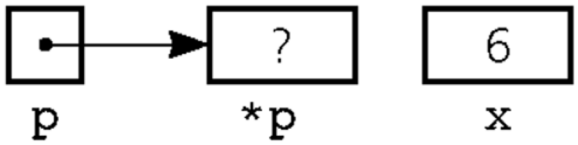
(b)
```
p = &x;
```



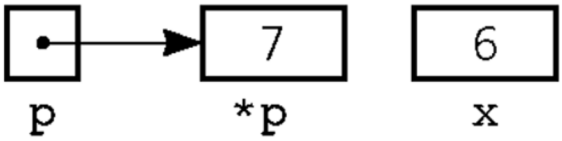
(c)
```
*p = 6;
```



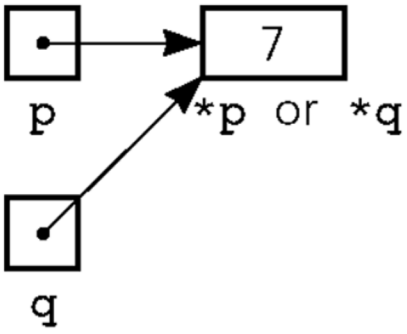
(d)
```
p = new int;
```



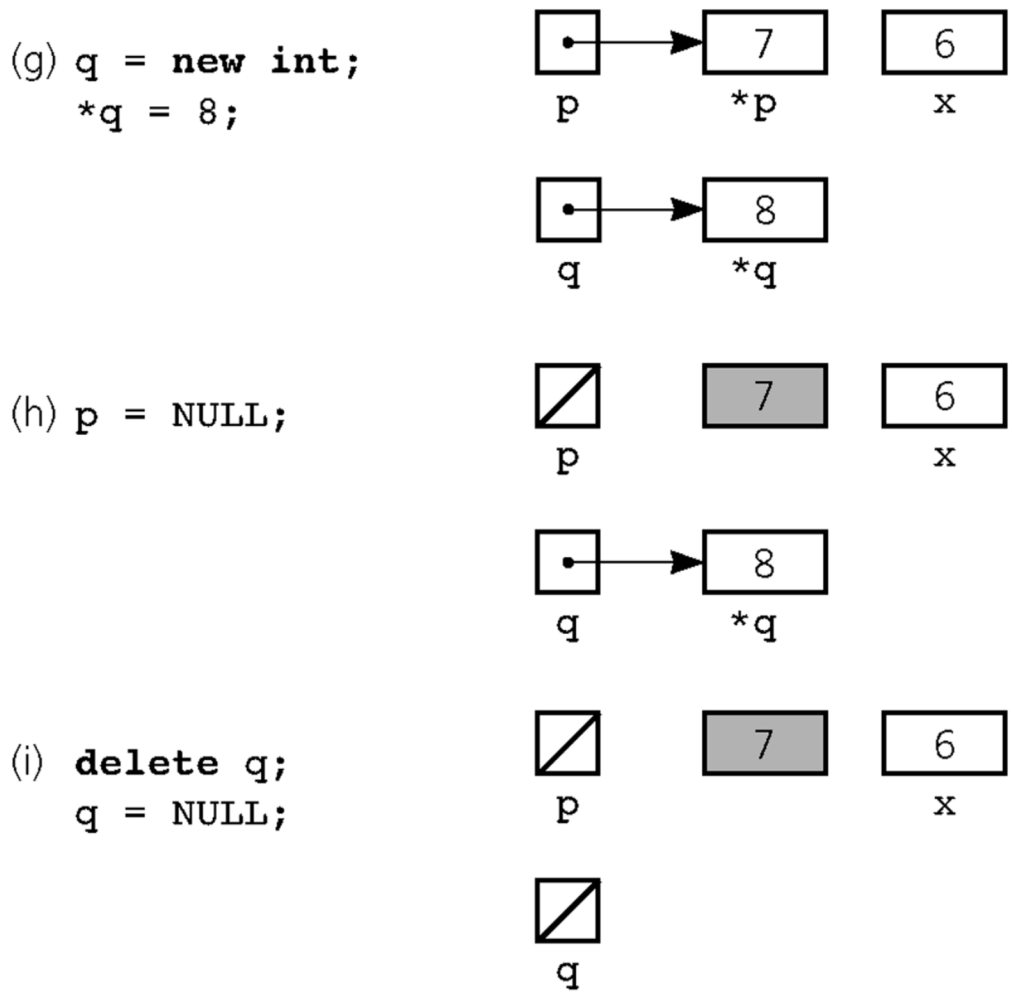
(e)
```
*p = 7;
```



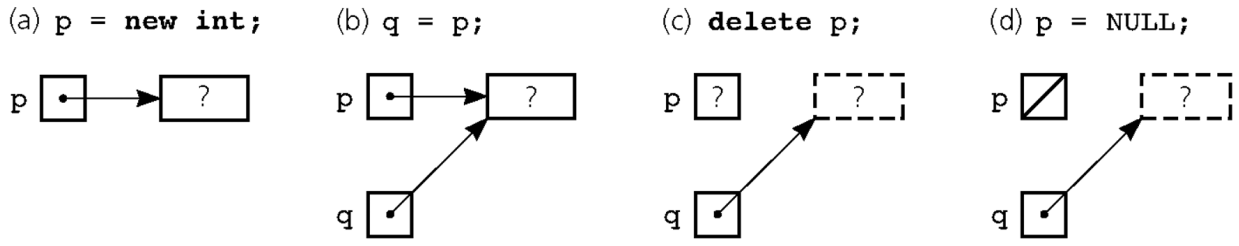
(f)
```
q = p;
```



(g) Allocating memory dynamically and assigning a value;
(h) assigning *NULL* to a pointer variable; (i) deallocating memory

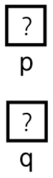


An incorrect pointer to a de-allocated node

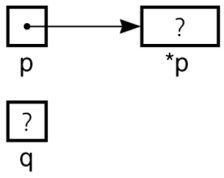


Programming with pointer variables and dynamically allocated memory

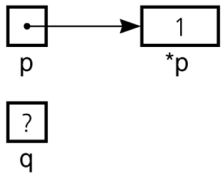
```
int *p, *q;
```



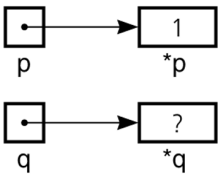

```
p = new int;        // Allocate a cell of type int.
```



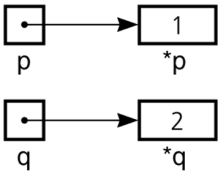

```
*p = 1;             // Assign a value to the new cell.
```



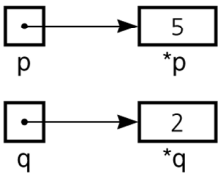

```
q = new int;        // Allocate a cell of type int.
```



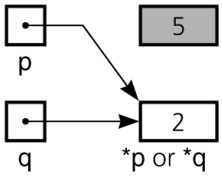

```
*q = 2;             // Assign a value to the new cell.

cout << *p << " "   // Output line contains: 1 2
     << *q << endl; // These values are in the
                    // cells to which p and q point.
```

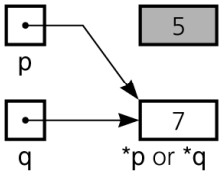


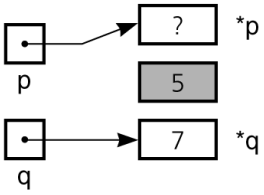

```
*p = *q + 3;        // The value in the cell to which
                    // q points, 2 in this case, and 3
                    // are added together. The result is
                    // assigned to the cell to which
                    // p points.

cout << *p << " "   // Output line contains: 5 2
     << *q << endl;
```

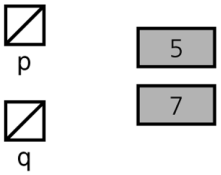


```
p = q;              // p now points to the same cell as q.
                    // The cell p formerly pointed to is
                    // lost; it cannot be referenced.

cout << *p << " "   // Output line contains: 2 2
     << *q << endl;

*p = 7;             // The cell to which p points (which
                    // is also the cell to which q points)
                    // now contains the value 7.

cout << *p << " "   // Output line contains: 7 7
     << *q << endl;

p = new int;        // This changes what p points to,
                    // but not what q points to.

delete p ;          // Return to the system the cell to
                    // which p points.

p = NULL;           // Set p to NULL, a good practice
                    // following delete.

q = NULL;           // The cell to which q previously
                    // pointed is now lost. You cannot
                    // reference it.
```









Passing Parameters by Reference: C++

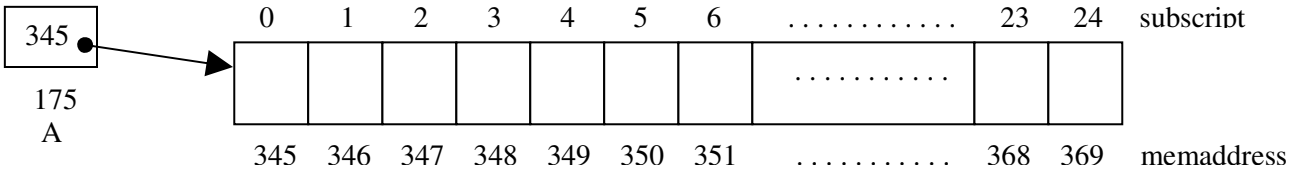
```
//  Reference parameters
void CalcMany_Ref(const int n, int &squared, int &cubed)
{
   squared = n * n;           // Implicit indirection
   cubed   = squared * n;
}

int main()
{
   int n = 3;
   int n2, n3;

   CalcMany_Ref( n, n2, n3);    // Implicit "reference"

   cout << n << n2 << n3 << endl;
   return(0);
}
```

Passing Parameters by Address (Pointers): C or C++

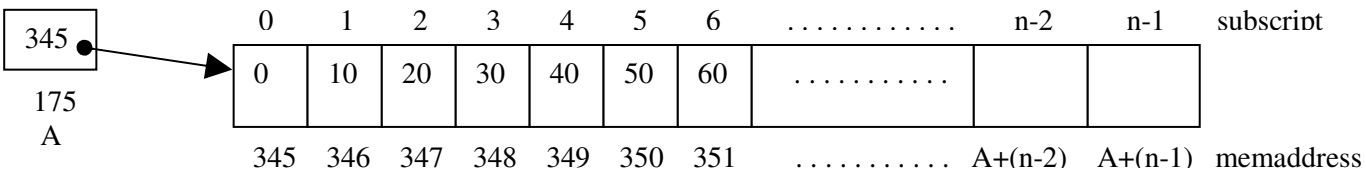
```
//   Pointer parameters
void CalcMany_Ptr(const int n, int *squared, int *cubed)
{
   *squared = n * n;          //Explicit * indirection
   *cubed   = (*squared) * n;
}

int main()
{
   int n = 3;
   int n2, n3;

   CalcMany_Ptr( n, &n2, & n3);   // Explicit & address-of

   cout << n << n2 << n3 << endl;
   return(0);
}
```

Dynamic Arrays

```
int     *A;           // Base pointer

A = new int[25];      // array of 25 integers
                      // note: [ ] square brackets
```




```
// ------------------------------------------------
// Alternative: Wait until runtime to determine size of A

int     *A;           // Base pointer

int      n;
cin >> n;             // input, say, 100

A = new int[n];       // array of  n (100) integers

// ------------------------------------------------
// Once A is allocated, reference just like an array:

for(int i=0; i < n; ++i)
{
   A[i] = i * 10;
}
```




```
// ------------------------------------------------
// Delete dynamic array

delete []  A;      // the [ ] are mandatory
```

Array Notations & Pointer Arithmetic

```
int  *A;            // "Dynamic" Array
                    // Size determined on creation
A = new int[100];
```

`A[0] = 16;`	`*A = 16;`
`A[1] = 31;`	`*(A + 1) = 31;`
`A[i]` <--->	`*(A + i)`

A is a (variable) pointer to the first [0] item in array

```
// Regular Array Declaration

int  B[100];
```

`B[0] = 16;`	`*B = 16;`
`B[1] = 31;`	`*(B + 1) = 31;`
`B[i]` <--->	`*(B + i)`

B is a (constant) pointer to the first [0] item in array

```
// Assignments?

A = B;          // OK.  A points to first elt of B

B = A;          // Illegal: B is a constant pointer


Reference to  &A     // OK - Address of the variable A
Reference to   A     // OK - Address of first item
                     //      in array that A points to

Reference to  &B     // Error - B is a constant
Reference to   B     // OK - Address of first item of B
```

Equivalency of array names and pointers
A pointer that points to an item in an array can be referenced in C++ using similar notations to using the array name itself

```
////////////////////////////////////////////////
// ------------------------------------------
char  name[50];          // char array: ASCIIZ C-string

strcpy_s(name, 50, "Fred Flintstone");

cout << "The name is: " << name << endl;
// Outputs [The name is: Fred Flintstone\n]

// ------------------------------------------
char  *p;   // is uninitialized. No buffer is allocated.

strcpy_s(p, 50, "Fred Flintstone"); // illegal - no buffer
strcpy(p, "Fred Flintstone");       // illegal - no buffer

p = name;       // p points to start of array 'name'

cout << "The name is: " << p << endl;

////////////////////////////////////////////////
// Individual array item subscript access allowed
// ------------------------------------------
int len = strlen(name);

cout << "The name is: " ;
for(int i = 0; i < len; ++i)
   cout << name[i];
cout << endl;

// ------------------------------------------
int len = strlen(p);

cout << "The name is: " ;
for(int i = 0; i < len; ++i)
   cout << p[i];
cout << endl;
```

```
////////////////////////////////////////////////
// C-string output Using direct pointer access
```

```
p = name;
cout  <<  p  << endl;
```

```
p = name;

while(*p != 0)
{
   cout << *p;
   p++;
}
cout << endl;
```

```
p = name;

while(*p != 0)
{
   cout << *p++;
}
cout << endl;
```

```
for(p=name; *p != 0; p++)
{
   cout << *p;
}
cout << endl;
```

```
for(p=name; *p != 0; p++) cout << *p;
cout << endl;
```

```
////////////////////////////////////////////////
//      "Fred Flintstone"
//       0123456789111111
//                 012345
// ----------------------------------------
// ---------- pointer arithmetic ------------
// ----------------------------------------

// Outputs [The tail of the string is: Flintstone\n]

cout << "The tail of the string is: "
     << (p + 5) << endl;

// ----------------------------------------
cout << "The tail of the string is: "
     << (name + 5) << endl;

////////////////////////////////////////////////
cout << "The tail of the string is: "
     <<  &name[5] << endl; // address of item in name

// ----------------------------------------
cout << "The tail of the string is: "
     <<  &p[5] << endl; // address of item in name

////////////////////////////////////////////////
p = &name[5];

cout << "The tail of the string is: " <<  p  << endl;
```

Selecting fields from a class or struct object
(*p). vs. p-> notations

```
class TwoFields {
public:
   int   m_field1;
   float m_field2;
};
```

```
struct TwoFields {
  int    m_field1;
  float  m_field2;
};
```

```
TwoFields  MyStruct;    // Allocate a TwoFields instance
TwoFields  *pTF = &MyStruct;   // Create a ptr to MyStruct

MyStruct.m_field1 = 15;   // Standard '.' field-select
(*pTF).m_field1  = 15; //   using a (*dereferenced) ptr

pTF -> m_field1  = 15; // alternate equivalent notation
                //   -> is "preferred" notation

pTF -> m_field2  = 3.5;
```

```
void PrintTwoFields( const TwoFields *p)
{
   cout << "Field 1: " <<   p -> m_field1 << endl;
   cout << "Field 2: " <<   p -> m_field2 << endl;
}

int main()
{
   TwoFields  MyStruct = { 15, 3.5 };

   PrintTwoFields( & MyStruct );

   TwoFields  *pTF = new TwoFields;
   pTF->m_field1 = 5;
   pTF->m_field2 = 123.45;
   PrintTwoFields( pTF );

   return(0);
}
```