# Pointers

C++ uses an asterisk * to denote a pointer.

If ***Item*** is then a pointer to such an Item object has the type ***Item***

```
Item *item_ptr;
```

declares item_ptr as a pointer variable to an Item object.

For example:

```
int *pa;
```

# Pointer: Creating dynamic objects

```
pa = new int;
```

creates a new dynamic object of type ***int***

and assigns its location to the pointer variable ***pa***.

The dynamic objects that we create are kept in an area of computer memory called the free store (or the heap).

# Pointers: Deleting dynamic objects

```
delete pa;
```

disposes of the dynamic object to which ***pa*** points and returns the space it occupies to the free store so it can be used again.

After this delete statement is executed, the pointer variable *pa* is undefined and so should not be used until it is assigned a new value.

# Pointers: Following pointers, and NULL

***item_ptr** denotes the object to which ***item_ptr** points.

The action of taking ***item_ptr** is called dereferencing the pointer ***item_ptr**.

# Pointers: NULL Pointers

If a pointer variable item_ptr has no dynamic object to which it currently refers, then it should be given the special value

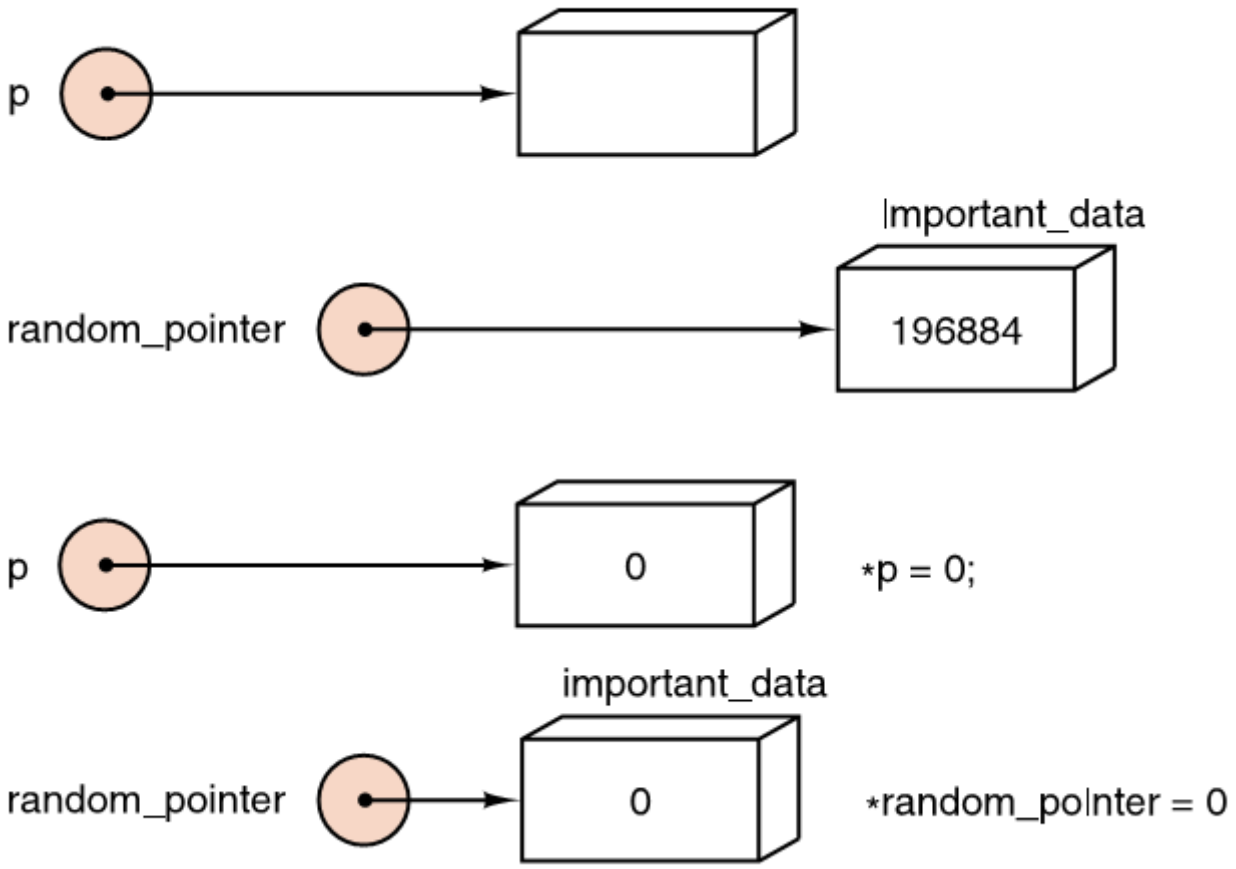
```
item_ptr = NULL;
```

In diagrams we reserve the electrical ground symbol for NULL pointers.

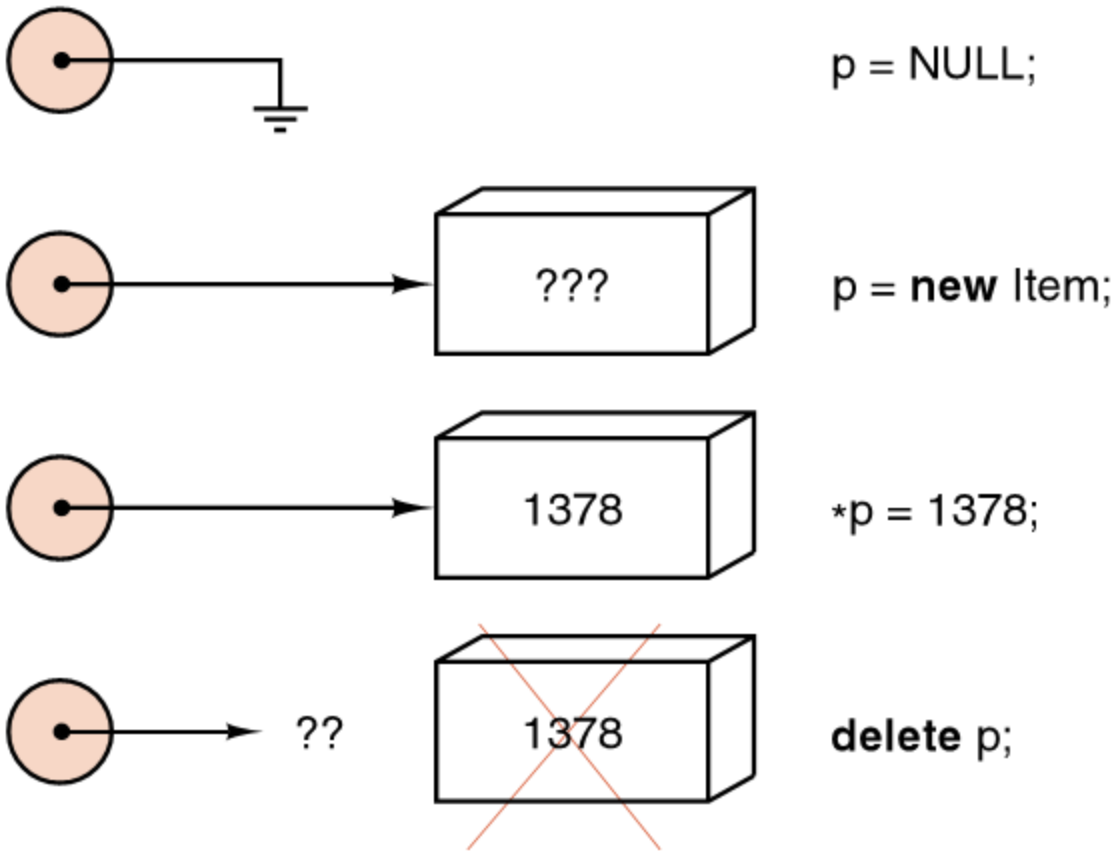
The value NULL is used as a constant for all pointer types .

Programming Precept: Uninitialized or random pointer objects should always be reset to NULL. After deletion, a pointer object should be reset to NULL.

# Pointers


p
important_data
random_pointer
196884
p
0
*p = 0;
important_data
random_pointer
0
*random_pointer = 0

# Pointers



# Dynamically Allocated Arrays

```
item_array = new Item[array_size];
```

creates a dynamic array of ***Item*** objects, indexed: 0 to array_size - 1

Example:

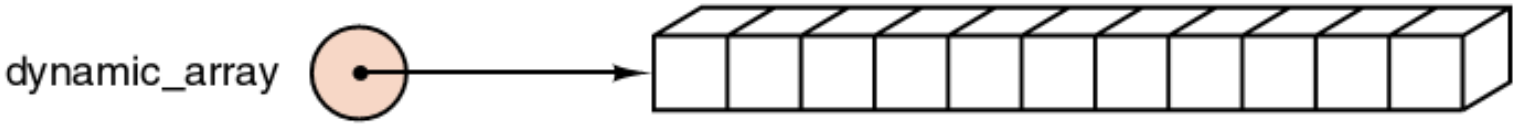
```
int size, *dynamic_array, i;
cout << "Enter an array size: ";
cin >> size;
dynamic_array = new int[size];
for (i = 0; i < size; i++) dynamic_array[i] = i;
```

# Dynamically Allocated Arrays

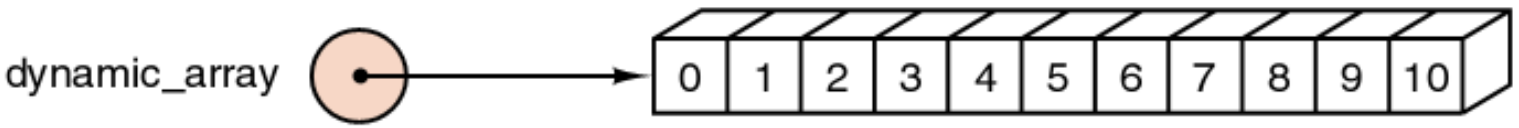
Example:

```
int size, *dynamic_array, i;
cout << "Enter an array size: ";
cin >> size;
dynamic_array = new int[size];
for (i = 0; i < size; i++) dynamic_array[i] = i;
```

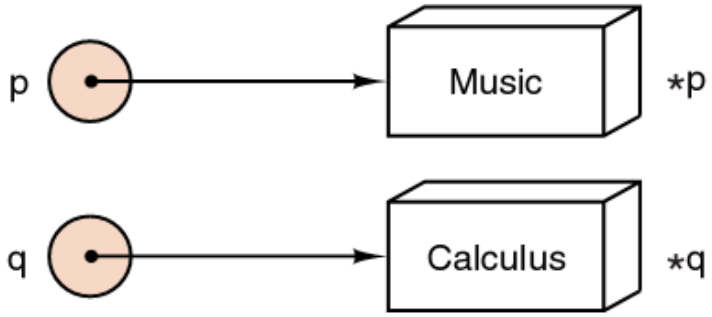
dynamic_array = **new int** [size];

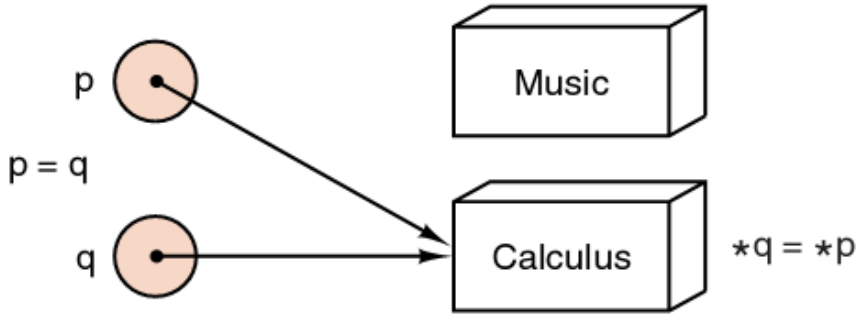


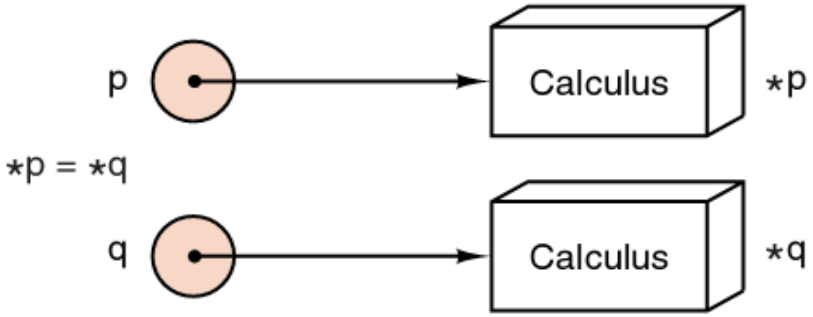

**for** (i=0; i<size; i++) dynamic_array[i ] = i;



# Assignment of Pointer Variables



# Addresses of automatic objects

If ***x*** is a variable of type ***Item***, then ***&x*** is a value of type ***Item**** that gives the address of ***x***.

```
Item *ptr = &x;
```

establish a pointer, ptr, to the object x.

# Address of an array

The address of the initial element of an array is found by using the array's name without any attached [ ] operators.

Given an array ***Item x[20]*** , the assignment

```
Item *ptr = x;
```

sets up a pointer ptr to the initial element of the array x.

Expression `ptr = &(x[0]);`

Could also be used to nd this address.

# Pointers to structures

If ***p*** is a pointer to a structure object that has a data member called the data, then we could access this data member with the expression

**(*p).data**

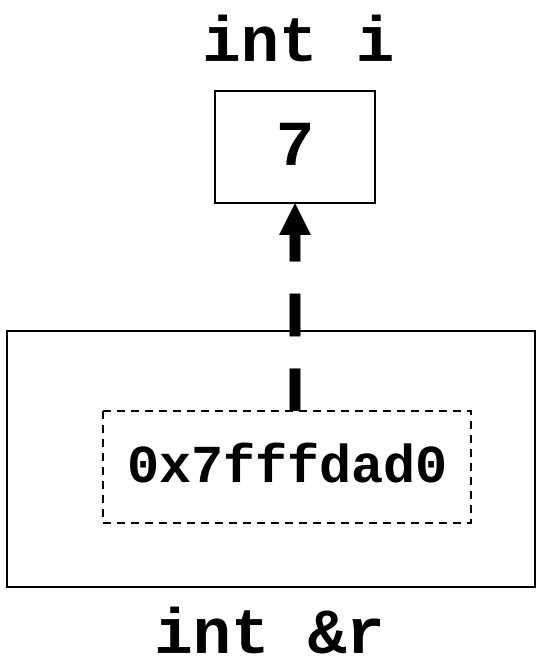
but C++ provides the operator -> as a shorthand, so we can replace the expression ***(*p). data*** by the equivalent, but more convenient, expression

**p->data**

# Untangling Operator Syntax

| Symbol | Used in a declaration | Used in a definition |
|---|---|---|
| unary **&** (ampersand) | reference<br>`int i = 3;`<br>`int &r = i;` | address-of<br>`p = &i;` |
| unary ***** (star) | pointer<br>`int * p;` | dereference (get what's pointed to)<br>`*p = 7;` |
| **->** (arrow) | | member access via pointer<br>`cp->add(3);` |
| **.** (dot) | | member access (same syntax for either reference or object)<br>`c.add(3);` |
| [] (square bracket) | array dimensions<br>`int a[3];` | array indexing<br>`cout << a[0] << endl;` |

# What's a Reference in C++?




- A variable holding an address
  - Of what it "refers to" in memory
- But with a nicer interface
  - An alias to the object
  - Hides indirection from programmer
- Must be typed
  - Checked by compiler
  - Again can only refer to the type to which it can point

```
int &r = i; // can only refer to int
```

- Must always refer to something
  - Must be initialized, cannot be changed
  - More restricted than Java references