

C++ Pointers

August 9, 2010

Remember Memory?

We've discussed:

- ▶ Variables that associate portions of program memory with a name in our program.

```
int x(10);
```

- ▶ References that pass the memory locations of variables to functions, instead of a copy of the variable value.

```
ostream& print( ostream& os, const className& rhs );
```

- ▶ Arrays, that associate a region of memory *holding one or more same-type variables* with a name in our program.

```
double dblArray[5] = { 3.2, 4 };
```

It's time we dealt with memory directly — using *pointers*.

The Pointer Type

Pointer Type Holds the memory address of a variable.

A memory address requires the same number of bits as an `int` (most hardware platforms).

Not all pointer types are the same! C++ distinguishes pointer variables by the *type* of variable they “point to.”

An `int` pointer is different than a `double` pointer, which is different than a `char` pointer...

Declaring Pointer Variables

Warning

```
1 // A pointer to an integer
2 int* ptrInt;
3 // A pointer to a double
4 double* ptrDbl;
5
6 // Two pointers to integers
7 int *ptrI1, *ptrI2;
8
9 // They aren't the same!
10 bool b( ptrInt == ptrDbl );
```

```
int* p1, p2, p3;
```

Is 1 pointer and two integer variables!

Declare multiple (same type) pointers like this:

```
int *p1, *p2, *p3;
```

```
10: error: comparison between distinct pointer types 'int*' and 'double*' lacks a cast
```

None of the pointer variables contain valid values (memory locations) yet!

Defining Pointer Variable Values

```
1 // An integer variable X, with value 3
2 // and pX holds the address of X
3 int X(3), *pX( &X );
4
5 // A double variable Y, uninitialized,
6 // and pY holds the address of Y
7 double Y, *pY = &Y;
8
9 // Pointer definition across
10 // two lines. pY2 points to the
11 // address of Y as well.
12 double* pY2; // right now pY2 is garbage
13 pY2 = &Y;    // NOW, it points to Y.
14
15 // The compiler doesn't like this...
16 double* pX2;
17 pX2 = &X;
```

The & Operator

Determines the memory address of its right-hand argument.

Frequently called the “Address-Of” Operator

```
17: error: cannot convert 'int*' to 'double*' in assignment
```

Question 1.

Reference Type vs Address-Of Operator

When is & used to specify a reference?

- ▶ When the left-hand token is a data type (intervening whitespace ignored).

```
ostream& print( ostream& os, const className& rhs );
```

When is & the address-of operator?

- ▶ When the right-hand token is an already defined variable.

```
int X, *pX( &X );
```

Reference Type vs Address-Of Operator

```
1 #include <iostream>
2 using namespace std;
3 void function( int call_by_value , int& reference )
4 {
5     cout << "_____The_function_reference_is_at_" <<
6         (unsigned long)&reference << endl;
7     cout << "The_function_call_by_value_is_at_" <<
8         (unsigned long)&call_by_value << endl;
9 }
10
11 int main()
12 {
13     int x(3);
14     cout << "_____The_main_x_is_at_" <<
15         (unsigned long)&x << endl;
16     function( x, x );
17     return 0;
18 }
```

Where is the & an address-of operator?

Where is the & a reference type indicator?

What is the “address” of a reference?

```
          The main x is at 140736809145964
The function reference is at 140736809145964
The function call_by_value is at 140736809145916
```

Dereferencing Pointer Variables

The * Operator

“Dereferences” a pointer variable to *use* its value.

“Dereferences” a pointer variable to *change* its value.

```

1 #include <iostream>
2 using namespace std;
3 void function( int* pInt )
4 {
5     cout << "Function_pInt_points_to_" <<
6         (unsigned long)(pInt) << endl;
7     cout << "_Function_integer_is_at_" <<
8         (unsigned long)(pInt) <<
9         endl <<
10        "_____has_a_value_of_" <<
11        *pInt << endl;
12
13    *pInt = *pInt + 1;
14 }
15 int main()
16 {
17     int X(3), *pX( &X );
18     cout << "_____The_main_X_value_is_" <<
19         X << endl;
20     cout << "_____The_main_X_is_at_" <<
21         (unsigned long)( &X ) << endl;
22     cout << "_____The_main_pX_points_to_" <<
23         (unsigned long)(pX) << endl;
24     function( &X );
25     cout << "_____The_main_X_value_s_now_" <<
26         X << endl;
27     return 0;
28 }

```

```

The main X value is 3
The main X is at 140735845412756
The main pX points to 140735845412756
Function pInt points to 140735845412756
Function integer is at 140735845412756
has a value of 3
The main X value s now 4

```

Pointer Comparisons

```

4 void print_pointers( int* pX, int* pInt )
5 {
6     cout << "Address_compare:_";
7     if( pX == pInt ) {
8         cout << "pX_==_pInt" << endl;
9     } else {
10        cout << "pX_!=_pInt" << endl;
11    }
12    cout << "Value-At-Address_compare:_";
13    if( *pX == *pInt ) {
14        cout << "**pX_==_*pInt" << endl;
15    } else {
16        cout << "**pX_!=_*pInt" << endl;
17    }
18 }
19 int main()
20 {
21     int Y(5), X(3), *pX( &X ), *pInt( &X );
22
23     print_pointers( pX, pInt );
24     cout << endl;
25     pInt = &Y;
26     print_pointers( pX, pInt );
27     cout << endl;
28     Y = 3;
29     print_pointers( pX, pInt );
30
31     return 0;
32 }

```

Address Comparison:

$$p == q$$

Value-at-Address Comparison

$$*p == *q$$

Address compare: pX == pInt
Value-At-Address compare: *pX == *pInt

Address compare: pX != pInt
Value-At-Address compare: *pX != *pInt

Address compare: pX != pInt
Value-At-Address compare: *pX == *pInt

Question 2.

Pointer Arithmetic

When calculating with pointers something special happens...

```
1 char* pChar( 0 );
2 int* pInt( 0 );
3 double* pDbl( 0 );
4
5 for( int k=0; k<3; k++ ) {
6     cout << "k=" << k << "==" << endl;
7
8     cout << "pChar_begins_at_" << (unsigned long)(pChar++);
9     cout << "after_pChar++:" << (unsigned long)(pChar)
10    << endl;
11    cout << "pInt_begins_at_" << (unsigned long)(pInt);
12    cout << "after_++pInt:" << (unsigned long)(++pInt)
13    << endl;
14    cout << "pDbl_begins_at_" << (unsigned long)(pDbl);
15    pDbl += 1;
16    cout << "after_pDbl+=1:" <<
17    (unsigned long)(pDbl) << endl;
18 }
```

```
== k 0 ==
pChar begins at 0 after pChar++: 1
pInt begins at 0 after ++pInt: 4
pDbl begins at 0 after pDbl+=1: 8
== k 1 ==
pChar begins at 1 after pChar++: 2
pInt begins at 4 after ++pInt: 8
pDbl begins at 8 after pDbl+=1: 16
== k 2 ==
pChar begins at 2 after pChar++: 3
pInt begins at 8 after ++pInt: 12
pDbl begins at 16 after pDbl+=1: 24
```

Run Program

Pointer Arithmetic

Each value of 1 added to a pointer, “skips” over 1 variable in the pointed-to-memory.

```

1 char* pChar( 0 );
2 int* pInt( 0 );
3 double* pDbl( 0 );
4
5 for( int k=0; k<3; k++ ) {
6     cout << "==" << k << "==" << endl;
7
8     cout << "pChar_begins_at_" << (unsigned long)(pChar++);
9     cout << "_after_pChar++:" << (unsigned long)(pChar)
10    << endl;
11    cout << "pInt_begins_at_" << (unsigned long)(pInt);
12    cout << "_after_++pInt:" << (unsigned long)(++pInt)
13    << endl;
14    cout << "pDbl_begins_at_" << (unsigned long)(pDbl);
15    pDbl += 1;
16    cout << "_after_pDbl+=1:" <<
17    (unsigned long)(pDbl) << endl;
18 }

```

```

== k 0 ==
pChar begins at 0 after pChar++: 1
pInt begins at 0 after ++pInt: 4
pDbl begins at 0 after pDbl+=1: 8
== k 1 ==
pChar begins at 1 after pChar++: 2
pInt begins at 4 after ++pInt: 8
pDbl begins at 8 after pDbl+=1: 16
== k 2 ==
pChar begins at 2 after pChar++: 3
pInt begins at 8 after ++pInt: 12
pDbl begins at 16 after pDbl+=1: 24

```

Run Program

Question 3.

Const and Pointers

- ▶ The value of the pointer cannot be changed:

```
int x, y;  
int* const p( &x );  
p = &y; //disallowed  
*p = 3; //OK
```

- ▶ The value of the *variable located by the pointer* cannot be changed:

```
int x, y;  
const int* p( &x );  
p = &y; //OK  
*p = 3; //disallowed
```

- ▶ Nothing associated with the pointer may be changed:

```
int x, y;  
const int* const p( &x );  
p = &y; //disallowed  
*p = 3; //disallowed
```

Question 4.

Arrays and Pointers

The compiler treats the name of an array as a constant pointer to the array's memory.

We write: `int intArray[12];`

The compiler does:

```
int* const intArray( MEMORY_SPACE_FOR_12_INTEGERS );
```

Arrays are beneficial: they allow easy allocation of space for large chunks of data.

Pointers, on the other hand, require the programmer to **“find” memory on their own**, and then **initialize the pointer value correctly**.

Arrays and Pointers

You may treat pointers to memory space as arrays to the same memory space. Pointers may be passed as arrays in function arguments, and [] may be used with pointers — just like arrays.

```
1 double* pDbl( 0 );
2
3 for( int k=0; k<3; k++ ) {
4     cout << "==" << k << "==" << endl;
5
6     cout << "pDbl points at " << (unsigned long)(pDbl);
7     cout << "(pDbl + " << k << ") points at "
8         << (unsigned long)(pDbl + k)
9         << endl;
10    cout << "&pDbl[" << k << "] points at "
11        << (unsigned long)&pDbl[k]
12        << endl;
13 }
```

```
== k 0 ==
pDbl points at 0 (pDbl + 0) points at: 0
&pDbl[0] points at: 0
== k 1 ==
pDbl points at 0 (pDbl + 1) points at: 8
&pDbl[1] points at: 8
== k 2 ==
pDbl points at 0 (pDbl + 2) points at: 16
&pDbl[2] points at: 16
```

finis