POINTERS : using pointers, aliases, arrays, function pointers, handles

---

USING POINTERS

Pointers are indispensable and are frequently used in C programs, but they tend to be one of the stumbling blocks for new programmers and can crash a program in a heartbeat if misused. The concept of a pointer is simple; a pointer is a variable which contains the computer address of something in memory. You define a pointer as follows:

```
int *myPtr;       // define a pointer to an integer variable type.
```

Another method of defining a pointer looks like this:

```
int* myPtr;       // another way to define a pointer.
```

You assign values to pointers, and dereference pointers as follows:

```
int i = 5;         // declare integer and assign value of 5.
int j;             // declare integer.
int *ip;           // declare pointer to type int.

ip = &i;           // address of 'i' stored in variable 'ip'.
j = *ip;           // dereference pointer to return value at address.
cout << i << j; // value of both 'i' and 'j' are the same.
```

Pointers are a necessary evil in C programming. They provide you with a lot of power, but one bad pointer can ruin your day. If you pass a bad pointer to a Toolbox function like PaintRect() your program will likely crash. Here's a simple snippet which shows the basics of declaring a pointer and assigning it the address of a variable:

```
// pointer.cp
#include <iostream.h>

void myPrintAddr( int j );   // pass by value

main()

   int i = 1;
   int *iPtr;

   iPtr = &i;                      // &i returns address of 'i'
   cout << "The value of i is " << i << endl;
   cout << "The address of i is " << iPtr << endl;
   cout << "The value at address " << iPtr << " is "
      << *iPtr << endl;          // *iPtr 'dereferences' pointer iPtr
   myPrintAddr( i );

   return 0;


void myPrintAddr( int j )

   int *jPtr;

   jPtr = &j;
   cout << "The value of j is " << j << endl;
   cout << "The address of j is " << jPtr << endl;
   cout << "Value's the same, address is different!" << endl;

// end pointer.cp
```

When incrementing pointers, the pointer is incremented by the size of the object it points to. For example, if myPtr

points to a data structure which has a size of X bytes, myPtr++ will add X to the variable myPtr.

ALIASES

C++ allows you to create aliases (also known as references). As the name implies, an alias is just another name for the same item. You can use aliases to pass function variables by reference rather than by value. This can alleviate a lot of headaches. If you find yourself having to dereference pointer variables, try using aliases instead.

```
// alias.cp
/*
    Compare results to "pointer.cp" snippet above.
*/

#include <iostream.h>

void myPrintAddr( int &j );   // pass by reference

main()

   int i = 1;
   int *iPtr;

   iPtr = &i;
   cout << "The value of i is " << i << endl;
   cout << "The address of i is " << iPtr << endl;
   myPrintAddr( i );
   cout << "They match!" << endl;

   return 0;


void myPrintAddr( int &j )

   int *jPtr;

   jPtr = &j;
   cout << "The value of j is " << j << endl;
   cout << "The address of j is " << jPtr << endl;

// end alias.cp
```

ARRAYS

In C, an array is a pointer to a sequential block of related data types. For example, "float myArray[10]" would define a pointer "myArray" to the beginning of a block of memory sufficient to hold 10 variables of type float. The first item in an array is accessed with a "0" subscript. Therefore, "int i[5]" defines an array of 5 integers, i[0], i[1], i[2], i[3], and i[4]. Here's an example showing the use of arrays:

```
// array.cp
#include <iostream.h>

#define kSize 5

main()

   int    one[kSize] = 2,4,6,8,10;
   int    two[2][kSize] =  1,2,3,4,5,11,12,13,14,15 ;
```

```
    cout << "Single Dimensional Array" << endl;
    for( int i = 0; i<kSize; i++ )

        cout << i << " = " << one[i] << endl;


    cout << endl << "Multiple Dimensional Array" << endl;
    for( i = 0; i<kSize; i++ )

        cout << "0," << i << " = " << two[0][i] << endl;
        cout << "1," << i << " = " << two[1][i] << endl;


    return 0;

// end array.cp
```

POINTERS TO FUNCTIONS

You can define pointers to functions which can be used in arrays, returned by functions, or used to call the associated functions themselves.

```
// complex.cp
#include <iostream.h>
#include <stdlib.h>

int myCallFunction( int (*function)( int ), int mySeed );
int myRandom( int theSeed );

main()

    int mySeed = 1;
    int result;

    while( mySeed )

        cout << "Enter seed (zero to exit): ";
        cin >> mySeed;
        if( mySeed )

            result = myCallFunction( myRandom, mySeed );
            cout << "Result: " << result << endl;



    return 0;


int myCallFunction( int (*function)( int ), int input )

    int random;

    random = function( input );
    return random;


int myRandom( int theSeed )

    int result;
```

```
    srand( theSeed );
    result = rand();
    return result;

// end complex.cp
```

MACINTOSH HANDLES

The MacOS makes use of handles. Like pointers, they're not difficult to understand, but they can be the source of headaches when your debugging code. Handles are, simply, pointers to pointers.

```
int myInt;                  // declares integer.
int *myIntPtr;              // declares pointer.
int **myIntH;               // declares handle.
```

You normally don't declare handles as shown above. You typically use standard MacOS definitions like:

```
Handle myH;                 // generic MacOS handle.
TEHandle myTE;              // handle to MacOS TE data structure.
```

Then use Toolbox functions that return pointers to "master pointers" like:

```
myH = NewHandle( 1024 );        // returns handle to 1024-sized block.
myTE = NewTE( dRect, vRect ); // returns handle to TE structure.
```

Why handles? That's easy - memory fragmentation. The MacOS is constantly on the look for items which it can move to fill-in memory gaps. Handles allow the operating system to do just that. Should an item be moved, the value of the master pointer is updated, but the value of the handle remains the same.