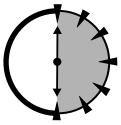
SESSION

A Few More Pointers

Session Checklist

- ✔ Introducing mathematical operations on character pointers
- ✔ Examining the relationship between pointers and arrays
- ✔ Applying this relationship to increase program performance
- ✔ Extending pointer operations to different pointer types
- ✔ Explaining the arguments to `main()` in our C++ program template



30 Min. To Go

The pointer types introduced in Session 13 allow for some interesting operations. Storing the address of a variable only to turn around and use that address more or less like the variable itself, is an interesting party trick, but it has limited use except for the capability to permanently modify variables passed to a function.

What makes pointers more interesting is the capability to perform mathematical operations. Of course, because multiplying two addresses is illogical, it is not allowed. However, the capability to compare two addresses or to add an integer offset opens interesting possibilities that are examined here.

Pointers and Arrays

Some of the same operators applicable to integers are applicable to pointer types. This section examines the implications of this both to pointers and to the array types studied so far.

Operations on pointers

Table 14-1 lists the three fundamental operations that are defined on pointers.

Table 14-1
Three Operations Defined on Pointer Types

Operation	Result	Meaning
`pointer + offset`	pointer	calculate the address of the object integer entries from pointer
`pointer - offset`	pointer	the opposite of addition
`pointer2 - pointer1`	offset	calculate the number of entries between pointer2 and pointer1

(Although not listed in Table 14-1, derivative operators, such as `pointer += offset` and `pointer++`, are also defined as variations of addition.)

The simplistic memory model used in Session 13 to explain the concept of pointers is useful here to explain how these operations work. Consider an array of 32 1-byte characters called `cArray`. If the first byte of this array is stored at address 0x110, then the array would extend over the range 0x110 through 0x12f. While `cArray[0]` is located at address 0x110, `cArray[1]` is at 0x111, `cArray[2]` at 0x112, and so forth.

Now assume a pointer `ptr` located at address 0x102. After executing the expression:

```
ptr = &cArray[0];
```

the pointer `ptr` contains the address 0x110. This is demonstrated in Figure 14-1.

Addition of an integer offset to a pointer is defined so that the relationships shown in Table 14-2 are true. Figure 14-2 also demonstrates why adding an offset `n` to `ptr` calculates the address of the `n`th element in `cArray`.

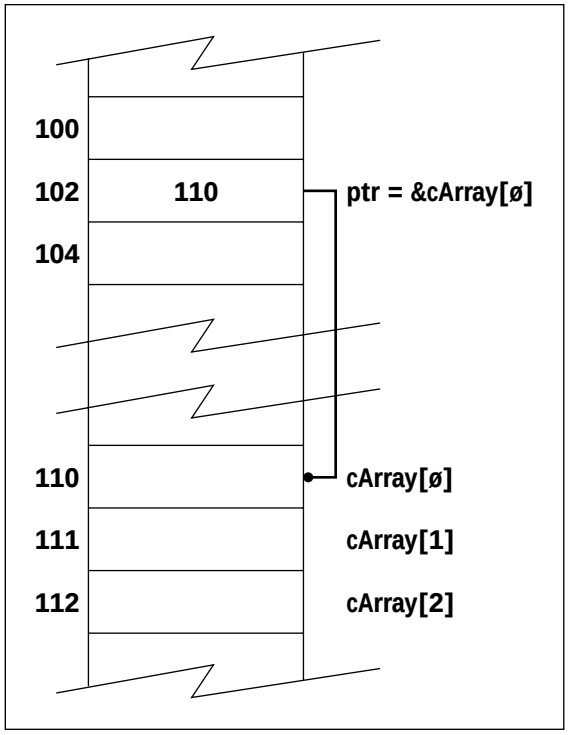


Figure 14-1
After the assignment ptr = &cArray[0] the pointer ptr points to the beginning of the array cArray.

Table 14-2
Pointer Offsets and Arrays

Offset	Result	Corresponds to . . .
+ 0	0x110	`cArray[0]`
+ 1	0x111	`cArray[1]`
+ 2	0x112	`cArray[2]`
. . .	. . .	. . .
+ *n*	0x110+ *n*	`cArray[`*n*`]`

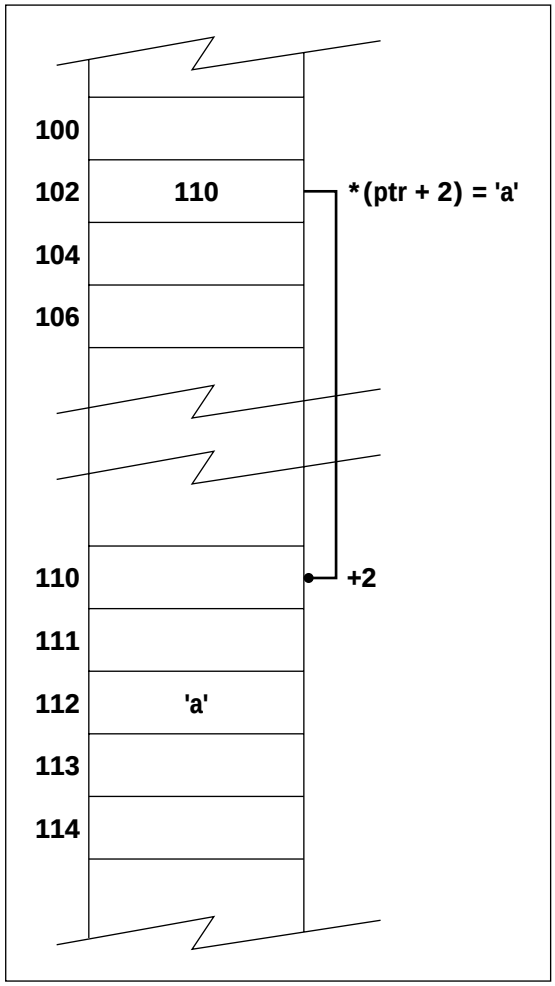


Figure 14-2
The expression ptr + i results in the address of cArray[i].

Thus, given that

```
char* ptr = &cArray[0];
```

then

```
*(ptr + n) ← corresponds with → cArray[n]
```

Because * has higher precedence than addition, *ptr + n adds n to the character which ptr points to. The parentheses are needed to force the addition to occur before the indirection. The expression *(ptr + n) retrieves the character pointed at by the pointer ptr plus the offset n.

In fact, the correspondence between the two forms of expression is so strong that C++ considers array[n] nothing more than a simplified version of *(ptr + n) where ptr points to the first element in array.

```
array[n] -- C++ interprets as → *(&array[0] + n)
```

To complete the association, C++ takes a second short cut. Given

```
char cArray[20];
```

then

```
cArray == &cArray[0]
```

That is, the name of an array without any subscript present is the address of the array itself. Thus, we can further simplify the association to

```
array[n] → C++ interprets as → *(array + n)
```

This is a powerful statement. For example, the displayArray() function from Session 11, which is used to display the contents of an array of integers, could be written:

```
// displayArray - display the members of an
//                array of length nSize
void displayArray(int nArray[], int nSize)
{
    cout << "The value of the array is:\n";

    // point at the first element of nArray
    int* pArray = nArray;
    while(nSize--)
    {
        cout.width(3);

        // output the integer pointed at by pArray...
cout << i << ": " << *pArray << "\n";
```

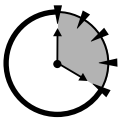
```
            // ...and move the pointer over to the next
            // member of nArray
            pArray++;
        }
        cout << "\n";
    }
```

The new `displayArray()` begins by creating a pointer to an integer `pArray` that points at the first element of `nArray`.

According to our naming convention, the `p` indicates pointer.

The function then loops through each element of the array (using `nSize` as the number of entries in the array). On each loop, `displayArray()` outputs the current integer, that is, the integer pointed at by `pArray` before incrementing the pointer to the next entry in `pArray`.

This use of pointers to access arrays is nowhere more common than in the accessing of character arrays.



20 Min. To Go

Character arrays

Session 11 also explained how C++ uses a character array with a null character at the end to serve as a quasistring variable type. C++ programmers often use character pointers to manipulate such strings. The following code examples compare this technique to the earlier technique of indexing in the array.

Pointer vs. array-based string manipulation

The `concatString()` function was declared in the Concatenate example in Session 11 as:

```
void concatString(char szTarget[], char szSource[]);
```

The prototype declaration describes the type of arguments that the function accepts, as well as the return type. This declaration appears the same as a function definition with no function body.

To find the null at the end of the szTarget array, the concatString() function iterated through szTarget string using the following while loop:

```
void concatString(char szTarget[], char szSource[])
{
    // find the end of the first string
    int nTargetIndex = 0;
    while(szTarget[nTargetIndex])
    {
        nTargetIndex++;
    }

    // ...
```

Using the relationship between pointers and arrays, concatString() could have been prototyped as follows:

```
void concatString(char* pszTarget, char* pszSource);
```

The sz refers to a string of characters that ends in a 0 (null).

The pointer version of concatString() contained in the program ConcatenatePtr is written:

```
void concatString(char* pszTarget, char* pszSource)
{
    // find the end of the first string
    while(*pszTarget)
    {
        pszTarget++;
    }

    // ...
```

The while loop in the array version of concatString() looped until szTarget[nTargetIndex] was equal to 0. This version iterates through the array by incrementing nTargetIndex on each pass through the loop until the character pointed at by pszTarget is null.

The expression ptr++ is a shortcut for ptr = ptr + 1.

Upon exiting the `while` loop, `pszTarget` points to the null character at the end of the `szTarget` string.

It is no longer correct to say "the array pointed at by `pszTarget`" because `pszTarget` no longer points to the beginning of the array.

The complete concatString() example

The following is the complete `ConcatenatePtr` program:

```
// ConcatenatePtr - concatenate two strings
//                  with a " - " in the middle
//                  using pointer arithmetic
//                  rather than array subscripts
#include <stdio.h>
#include <iostream.h>

void concatString(char* pszTarget, char* pszSource);

int main(int nArg, char* pszArgs[])
{
    // read first string...
    char szString1[256];
    cout << "Enter string #1:";
    cin.getline(szString1, 128);

    // ...now the second string...
    char szString2[128];
    cout << "Enter string #2:";
    cin.getline(szString2, 128);

    // ...concatenate a " - " onto the first...
    concatString(szString1, " - ");

    // ...now add the second string...
    concatString(szString1, szString2);

    // ...and display the result
    cout << "\n" << szString1 << "\n";
```

```
30.
31.      return 0;
32.  }
33.
34.  // concatString - concatenate *pszSource onto the
35.  //                end of *pszTarget
36.  void concatString(char* pszTarget, char* pszSource)
37.  {
38.      // find the end of the first string
39.      while(*pszTarget)
40.      {
41.          pszTarget++;
42.      }
43.
44.      // tack the second to the end of the first
45.      // (copy the null at the end of the source array
46.      // as well - this terminates the concatenated
47.      // array)
48.      while(*pszTarget++ = *pszSource++)
49.      {
50.      }
51.  }
```

The `main()` portion of the program does not differ from its array-based cousin. The `concatString()` function is significantly different, however.

As noted, the equivalent declaration of `concatString()` is now based on `char*` type pointers. In addition, the initial `while()` loop within `concatString()` searches for the terminating null at the end of the `pszTarget` array.

The extremely compact loop that follows copies the `pszSource` array to the end of the `pszTarget` array. The `while()` clause does all the work, executing as follows:

1. Fetch the character pointed at by `pszSource`.
2. Increment `pszSource` to the next character.
3. Save the character in the character position pointed at by `pszTarget`.
4. Increment `pszTarget` to the next character.
5. Execute the body of the loop if the character is not null.

After executing the empty body of the `while()` loop, control passes back up to the `while()` clause itself. This loop is repeated until the character copied to `*pszTarget` is the null character.

Why bother with array pointers?

The sometimes-cryptic nature of pointer-based manipulation of character strings might lead the reader to wonder why. That is, what advantage does the `char*` pointer version of `concatString()` have over the easier-to-read index version?

The pointer version of `concatenate()` is much more common in C++ programs than the array version from Session 11.

The answer is partially historic and partially human nature. As complicated as it might appear to the human reader, a statement such as line 48 can be converted to an amazingly small number of machine-level instructions. Older computer processors were not very fast by today's standards. When C, the progenitor of C++, was created some 30 years ago, saving a few computer instructions was a big deal. This gave C a big advantage over other languages of the day, notably Fortran, which did not offer pointer arithmetic.

In addition, programmers like to generate clever program statements to combat what can be a repetitively boring job. Once C++ programmers learn how to write compact and cryptic but efficient statements, there is no getting them back to searching arrays with indices.

Do not generate complex C++ expressions in order to create a more efficient program. There is no obvious relationship between the number of C++ statements and the number of machine instructions generated. For example, the following two expressions might generate the same amount of machine code:

```
*pszArray1++ = '\0';

*pszArray2 = '\0';
pszArray2 = pszArray2 + 1;
```

In the old days, when compilers were simpler, the first version would definitely have generated fewer instructions.

Operations on different pointer types

The two examples of pointer manipulation shown so far, `concatString(char*, char*)` and `displayArray(int*)`, have a fundamental difference.

It is not too hard to convince yourself that `szTarget + n` points to `szTarget[n]` when you consider that each `char` in `szTarget` occupies a single byte. After all, if `szTarget` were stored at 0x100, then the sixth element is located at 0x105 (0x100 + 5 equals 0x105).

Because C++ arrays begin counting at 0, `szTarget[5]` is the sixth element in the array.

It is not obvious that pointer addition works for `nArray` because each element in `nArray` is an `int` that occupies 4 bytes. If the first element in `nArray` is located at 0x100, then the sixth element is located at 0x114 (0x100 + (5 * 4) = 0x114).

Fortunately for us, in C++ `array + n` points at `array[n]` no matter how large a single element of `array` might be.

A good analogy is a city block of houses. If all of the street addresses on each street were numbered consecutively with no gaps, then house number 1605 would be the sixth house of the 1600 block. In order not to confuse the mail carrier too much, this relationship is true no matter how big the houses might be.

Differences between pointers and arrays

Despite the equivalent types, there are some differences between an array and a pointer. For one, the array allocates space for the data whereas the pointer does not:

```
void arrayVsPointer()
{
    // allocate storage for 128 characters
    char cArray[128];

    // allocate space for a pointer
    char* pArray;
}
```

Here cArray occupies 128 bytes, the amount of storage required to store 128 characters. pArray occupies only 4 bytes, the amount of storage required by a pointer.

The following function does not work:

```
void arrayVsPointer()
{
    // access elements with an array
    char cArray[128];
    cArray[10] = '0';
    *(cArray + 10) = '0';

    // access an 'element' of an array
    // which does not exist
    char* pArray;
    pArray[10] = '0';
    *(pArray + 10) = '0';
}
```

The expressions cArray[10] and *(cArray + 10) are equivalent and legal. The two expressions involving pArray don't make sense. While they are both legal to C++, the uninitialized pArray contains some random value. Thus, this second pair of statements attempts to store a 0 character somewhere randomly in memory.

This type of mistake is generally caught by the CPU resulting in the dreaded segment violation error that you see from time to time issuing from your favorite applications.

A second difference is that cArray is a constant whereas pArray is not. Thus, the following for loop used to initialize the array cArray does not work:

```
void arrayVsPointer()
{
char cArray[10];
for (int i = 0; i < 10; i++)
{
    *cArray = '\0';    // this makes sense...
    cArray++;          // ...this does not
}
}
```

The expression cArray++ makes no more sense than does 10++. The correct version is:

```
void arrayVsPointer()
{
char cArray[10];
char* pArray = cArray;
for (int i = 0; i < 10; i++)
{
    *pArray = '\0';    // this works great
    pArray++;
}
}
```

Arguments to the Program

Arrays of pointers are another type of array of particular interest. This section examines how to use these arrays to simplify your program.

Arrays of pointers

Just as arrays may contain other data types, an array may contain pointers. The following declares an array of pointers to ints.

```
int* pnInts[10];
```

Given the above declaration, pnInt[0] is a pointer to an int value. Thus, the following is true:

```
void fn()
{
    int n1;
    int* pnInts[3];
    pnInts[0] = &n1;
    *pnInts[0] = 1;
}
```

or

```
void fn()
```

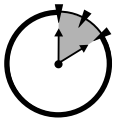
```
{
    int n1, n2, n3;
    int* pnInts[3] = {&n1, &n2, &n3};
    for (int i = 0; i < 3; i++)
    {
        *pnInts[i] = 0;
    }
}
```

or even

```
void fn()
{
    int* pnInts[3] = {(new int),
                      (new int),
                      (new int)};
    for (int i = 0; i < 3; i++)
    {
        *pnInts[i] = 0;
    }
}
```

The latter declares three `int` objects off the heap.

The most common use for arrays of pointers is to create arrays of character strings. The following two examples show why arrays of character strings are useful.



10 Min. To Go

Arrays of character strings

Suppose a function that returns the name of the month corresponding to an integer argument passed it is needed. For example, if the program is passed the value 1, it responds by returning a pointer to the string "January".

The function could be written as follows:

```
// int2month() - return the name of the month
char* int2month(int nMonth)
{
    char* pszReturnValue;

    switch(nMonth)
    {
```

```
        case 1: pszReturnValue = “January”;
                break;
        case 2: pszReturnValue = “February”;
                break;
        case 3: pszReturnValue = “March”;
                break;
        // ...and so forth...
        default: pszReturnValue = “invalid”;
    }
}
```

When passed a 1 for the month, control would pass to the first `case` statement and the function would dutifully return a pointer to the string “January”; when passed a 2, “February”; and so on.

The `switch()` control command is similar to a sequence of `if` statements.

A more elegant solution uses the integer value for the month as an index to an array of pointers to the names of the months. In use, this appears as follows:

```
// int2month() - return the name of the month
char* int2month(int nMonth)
{
    // first check for a value out of range
    if (nMonth < 1 || nMonth > 12)
    {
return << “invalid”;
        return;
    }

    // nMonth is valid - return the name of the month
    char* pszMonths[] = {“invalid”,
                         “January”,
                         “February”,
                         “March”,
                         “April”,
                         “May”,
                         “June”,
```

```
                                "July",
                                "August",
                                "September",
                                "October",
                                "November",
                                "December"};
    return pszMonths[nMonth];
}
```

Here `int2month()` first checks to make sure that `nMonth` is a number between 1 and 12, inclusive (the `default` clause of the `switch` statement handled that for us in the previous example). If `nMonth` is valid, the function uses it as an offset into an array containing the names of the months.

The arguments to main()

You have seen another application of arrays of pointers to strings: the arguments to `main()`.

The arguments to a program are the strings that appear with the program name when you launch it. For example, suppose I entered the following command at the MS-DOS prompt:

```
MyProgram file.txt /w
```

MS-DOS executes the program contained in the file `MyProgram.exe` passing it the arguments `file.txt` and `/w`.

The use of the term *arguments* is a little confusing. The arguments to a program and the arguments to a C++ function follow a different syntax but the meaning is the same.

Consider the following simple program:

```
// PrintArgs - write the arguments to the program
//             to the standard output
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* pszArgs[])
{
    // print a warning banner
```

```
    cout << "The arguments to " << pszArgs[0] << "\n";

    // now write out the remaining arguments
    for (int i = 1; i < nArg; i++)
    {
        cout << i << ":" << pszArgs[1] << "\n";
    }

    // that's it
    cout << "That's it\n";
    return 0;
}
```

As always, the function `main()` accepts two arguments. The first argument is an `int` that I have called `nArgs`. This variable is the number of arguments passed to the program. The second argument is an array of pointers of type `char*` that I have called `pszArgs`. Each of these `char*` elements points to an argument passed to the program.

Consider the program `PrintArgs`. If I invoke the program

```
PrintArgs arg1 arg2 arg3 /w
```

from the command line of an MS-DOS window, `nArgs` would be 5 (one for each argument). The first argument is the name of the program itself. Thus, `pszArgs[0]` points to `PrintArgs`. The remaining elements in `pszArgs` point to the program arguments. The element `pszArgs[1]` points to `arg1`, `pszArgs[2]` to `arg2`, and so on. Because MS-DOS does not place any significance on `/w`, this string is also passed as an argument to be processed by the program.

The same is not true of the redirection symbols "<", ">" and "|". These are significant to MS-DOS and are not passed to the program.

There are several ways to pass arguments to a function under test. The easiest way is to simply execute the program from the MS-DOS prompt. Both the Visual C++ and rhide debuggers provide a mechanism for passing arguments during debug.

In Visual C++, select the Debug tab in the Project Settings dialog box. Input your arguments in the Program Arguments edit window as shown in Figure 14-3. The next time you start the program, Visual C++ passes these arguments.

Done!

In rhide, select Arguments... under the Run menu. Enter any arguments in the edit window that appears. This is demonstrated in Figure 14-4.

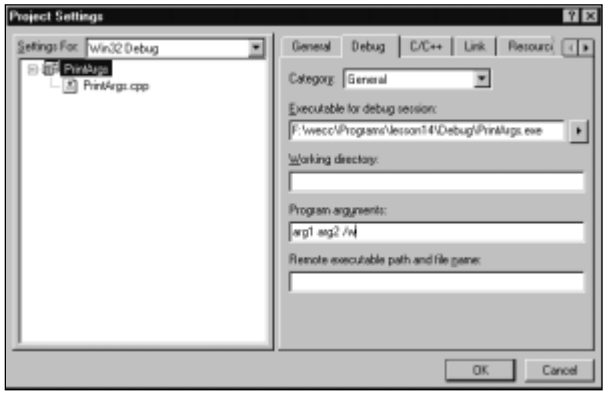


Figure 14-3
Visual C++ uses the Project Settings to pass arguments to the program under debug.

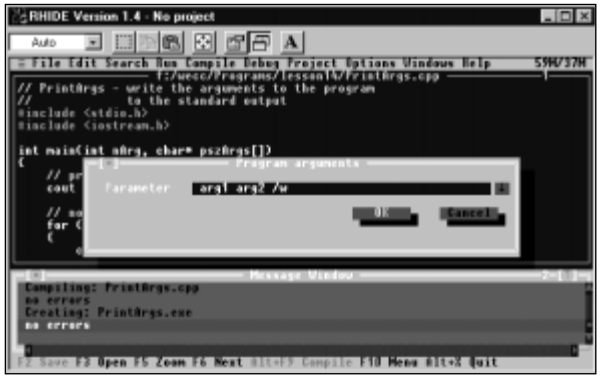


Figure 14-4
In rhide, the program arguments are found in the Run menu.

REVIEW

All languages base array indexing upon simple math operations on pointers; however, by allowing the programmer to have direct access to these types of operations, C++ gives the programmer tremendous semantic freedom. The C++ programmer can examine and use the relationship between the manipulation of arrays and pointers to her own advantage.

In this session you saw that

- Indexing in an array involves simple mathematical operations performed on pointers. C++ is practically unique in enabling the programmer to perform these operations himself or herself.
- Pointer operations on character arrays offered the greatest possibility for performance improvement in the early C and C++ compilers. Whether this is still true is debatable; however, the use of character pointers is a part of everyday life now.
- C++ adjusts pointer arithmetic to account for the size of the different types of objects pointed at. Thus, while incrementing a character pointer might increase the value of the pointer by 1, incrementing a double pointer would increase the value by 8. Incrementing the pointer of class object might increase the address by hundreds of bytes.
- Arrays of pointers can add significant efficiencies to a program for functions which convert an integer value to some other constant type, such as a character string or a bit field.
- Arguments to a program are passed to the `main()` function as an array of pointers to character strings.

QUIZ YOURSELF

1. If the first element of an array of characters `c[]` is located at address 0x100, what is the address of `c[2]`? (See "Operations on Pointers.")
2. What is the index equivalent to the pointer expression `*(c + 2)`? (See "Pointer vs. Array-Based String Manipulation.")
3. What is the purpose of the two arguments to `main()`? (See "The Arguments to main().")