Figure 17-4 shows the contents of memory immediately after these two statements are executed. Pretty simple, really — the value of `nArg1` has changed to 10 and `nArg2` to 20 just as you would expect. The main point of this demonstration, however, is the fact that changing the value of `nArg1` and `nArg2` has no effect on the original variables back at `nValue1` and `nValue2`.

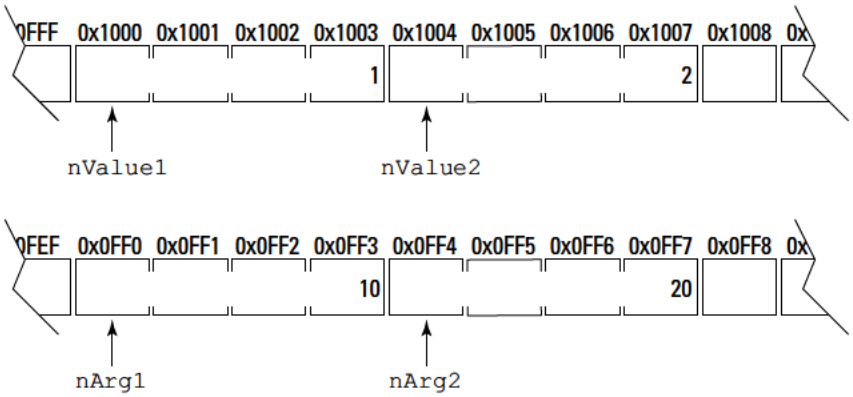


Figure 17-4: The same memory locations immediately prior to returning from `fn(int, int)`.

Contents of memory after executing the two assigning statements:

```
fn(int nArg1, int nArg2)
{
    nArg1   10;
    nArg2   20;
}
```

Passing arguments by reference

So what if I wanted the changes made by `fn()` to be permanent? I could do this by passing not the value of the variables but their address. This is demonstrated by the following snippet (also taken from the PassByReference example program):

```
// fn(int*, int*) - this function takes its arguments
//                  by reference
void fn(int* pnArg1, int* pnArg2)
{
    // modify the value of the arguments
    *pnArg1 = 10;
    *pnArg2 = 20;
}
```

```
int main(int nNumberofArgs, char* pszArgs[])
{
    // initialize two variables and display their values
    int nValue1 = 1;
    int nValue2 = 2;

    fn(&nValue1, &nValue2);

    return 0;
}
```

Notice first that the arguments to `fn()` are now declared not to be integers but pointers to integers. The call to `fn(int*, int*)` passes not the value of the variables `nValue1` and `nValue2` but their address.

In this example, the value of the expression `&nValue1` is 0x1000, and the type is `int*` (which is pronounced "pointer to int").

The state of memory upon entry into this function is shown in Figure 17-5.

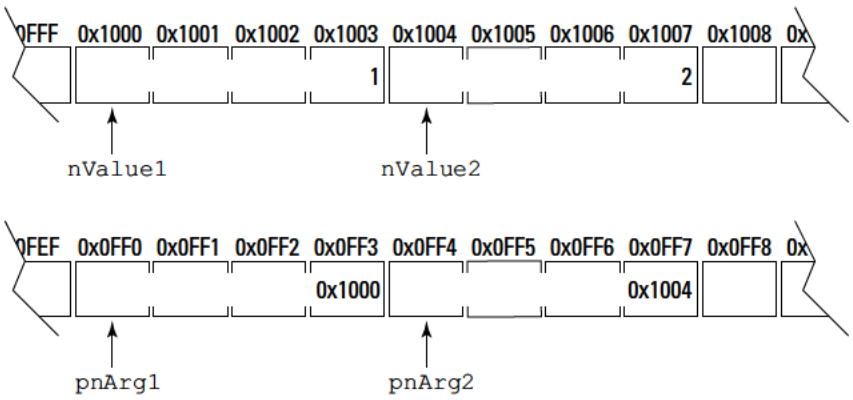


Figure 17-5: The content of memory after the call to `fn(int*, int*)`.

Layout in memory immediately after making the call: `fn(&nValue1, &nValue2)`

The function `fn(int*, int*)` now stores its values at the locations pointed at by its arguments:

```
*pnArg1 = 10;
*pnArg2 = 20;
```

This first statement says "store the value 10 at the `int` location passed to me in the argument `pnArg1`." This stores a 10 at location 0x1000, which happens to be the variable `nValue1`. This is demonstrated graphically in Figure 17-6.

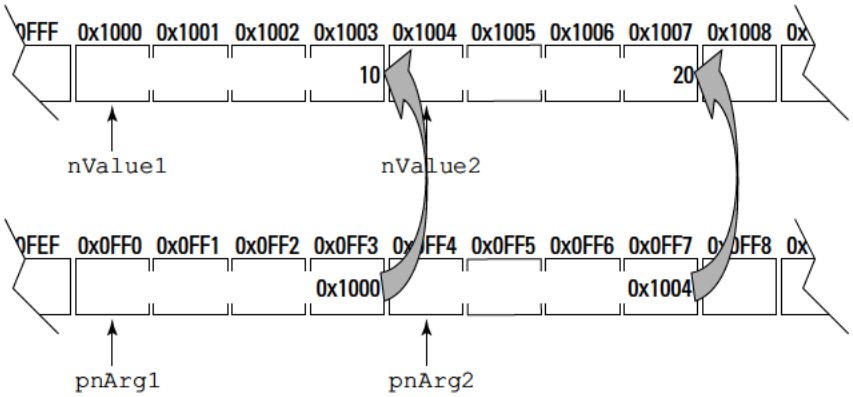


Figure 17-6: The content memory immediately prior to returning from `fn(int*, int*)`.

Contents of memory after executing the two assignment statements:

```
fn(int* pnArg1,int* pnArg2)
{
    *pnArg1   10;
    *pnArg2   20;
}
```

Putting it together

The complete `PassByReference` program appears as follows:

```
//
//  PassByReference - demonstrate passing arguments to a
//                    function both by value and by
//                    reference.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// fn(int, int) - demonstrate a function that takes two
//                arguments and modifies their value
void fn(int nArg1, int nArg2)
{
```

```
    // modify the value of the arguments
    nArg1 = 10;
    nArg2 = 20;
}

// fn(int*, int*) - this function takes its arguments
//                  by reference
void fn(int* pnArg1, int* pnArg2)
{
    // modify the value of the arguments
    *pnArg1 = 10;
    *pnArg2 = 20;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // initialize two variables and display their values
    int nValue1 = 1;
    int nValue2 = 2;
    cout << "The value of nArg1 is " << nValue1 << endl;
    cout << "The value of nArg2 is " << nValue2 << endl;

    // now try to modify them by calling a function
    cout << "Calling fn(int, int)" << endl;
    fn(nValue1, nValue2);
    cout << "Returned from fn(int, int)" << endl;
    cout << "The value of nArg1 is " << nValue1 << endl;
    cout << "The value of nArg2 is " << nValue2 << endl;

    // try again by calling a function that takes
    // addresses as arguments
    cout << "Calling fn(int*, int*)" << endl;
    fn(&nValue1, &nValue2);
    cout << "Returned from fn(int*, int*)" << endl;
    cout << "The value of nArg1 is " << nValue1 << endl;
    cout << "The value of nArg2 is " << nValue2 << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The following is the output from this program:

```
The value of nArg1 is 1
The value of nArg2 is 2
Calling fn(int, int)
Returned from fn(int, int)
The value of nArg1 is 1
```

```
The value of nArg2 is 2
Calling fn(int*, int*)
Returned from fn(int*, int*)
The value of nArg1 is 10
The value of nArg2 is 20
Press any key to continue . . .
```

This program declares the variables `nValue1` and `nValue2` and initializes them to 1 and 2, respectively. The program then displays their value just to make sure. Next, the program calls the `fn(int, int)`, passing the value of the two variables. That function modifies the value of its arguments, but this has no effect on `nValue1` and `nValue2` as demonstrated by the fact that their value is unchanged after control returns to `main()`.

The second call passes not the value of `nValue1` and `nValue2` but their address to the function `fn(int*, int*)`. This time, the changes to `pnArg1` and `pnArg2` are retained even after control returns to `main()`.

Notice that there is no confusion between the overloaded functions `fn(int, int)` and `fn(int*, int*)`. The types of the arguments are easily distinguished.

Playing with Heaps of Memory

One of the problems addressed in Chapter 16 was that of fixed-size arrays. For example, the `concatenate()` function concatenated two ASCIIZ strings into a single string. However, the function had to be careful not to overrun the target array in the event that there wasn't enough room to hold the combined string. This problem would have gone away if `concatenate()` could have allocated a new array that was guaranteed to be large enough to hold the concatenated string.

That's a great idea, but how big should I make this target array — 256 bytes, 512 bytes? There's no right answer since there's no way to know at compile time how big to make the target array so that it has enough room to hold all possible concatenated strings. You can't know for sure until runtime how much memory you will need.

Do you really need a new keyword?

C++ provides an extra area in memory just for this purpose, known by the somewhat cryptic name of the *heap*. A programmer can allocate any amount of memory off of the heap using the keyword `new`, as in the following example snippet:

```
char* pArray = new char[256];
```

This example carves a block of memory large enough to hold 256 characters off of the heap. The `new` keyword returns a pointer to the newly created array. Unlike other variables, heap memory is not allocated until runtime, which means the array size is not limited to constants that are determined at compile time — they can also be variables that are computed at runtime.

It may seem odd that the argument to `new` is an array while what is returned is a pointer. I will have a lot more to say about the relationship between pointers and arrays in the next chapter.

Thus, I could have said something like the following:

```
int nSizeOfArray = someFunction();
char* pArray = new char[nSizeOfArray];
```

Here the size of the array is computed by `someFunction()`. Obviously this computation can't occur until the program is actually executing. Whatever value `someFunction()` returns is used as the size of the array to be allocated in the next statement.

A more practical example is the following code snippet that makes a copy of an ASCIIZ string (assuming you consider copying a string as practical):

```
int nLength = strlen(pszString) + 1;
char* pszCopy = new char[nLength];
strncpy(pszCopy, nLength, pszString);
```

The first statement calls the string function `strlen()`, which returns the length of the string passed it not including the terminating NULL character. The `+ 1` adds room for the terminating NULL. The next statement allocates room for the copy off of the heap. Finally, the third string uses the string function `strncpy()` to copy the contents of `pszString` into `pszCopy`. By calculating how big an array you need to store the copy, you are guaranteed that `pszCopy` is large enough to hold the entire string.

Don't forget to clean up after yourself

Allocating memory off of the heap is a neat feature, but it has one very big danger in C++: If you allocate memory off of the heap, you must remember to return it.

You return memory to the heap using the `delete` keyword as in the following:

```
char* pArray = new char[256];

// ...use the memory all you want...

// now return the memory block to the heap
delete[] pArray;
pArray = NULL;
```

The `delete[]` keyword accepts a pointer that has been passed to you from the new keyword and restores that memory to the heap.

Use `delete[]` to return an array. Use `delete` (without the open and closed brackets) when returning a single object to the heap.

If you don't return heap memory when you are done with it, your program will slowly consume memory and eventually slow down more and more as the operating system tries to fulfill its apparently insatiable gluttony. Eventually, the program will come to a halt when the O/S can no longer satisfy its requests for memory.

Returning the same memory to the heap twice is not quite as bad. That causes the program to crash almost immediately. It is considered good programming practice to zero out a pointer once you have deleted the memory block that it points to for two very good reasons:

- **Deleting a pointer that contains a NULL has no effect.**
- **NULL is never a valid address.** Trying to access memory at the NULL location will always cause your program to crash immediately, which will tip you off that there is a problem and make it a lot easier to find.

You don't have to delete memory if your program will exit soon — all heap memory is restored to the operating system when a program terminates. However, returning memory that you allocate off the heap is a good habit to get into.

Looking at an example

The following ConcatenateHeap program is a version of the `concatenate()` function that allocates its memory off of the heap:

```
//
//  ConcatenateHeap - similar to ConcatenateString except
//                    this version stores the concatenated
//                    string in memory allocated from the
//                    heap so that we are guaranteed
//                    that the target array is always
//                    large enough
//
```

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;


// concatenateString - concatenate two strings together
//                     into an array allocated off of the
//                     heap
char* concatenateString(const char szSrc1[],
                        const char szSrc2[])
{
    // allocate an array of sufficient length
    int nTargetSize = strlen(szSrc1) + strlen(szSrc2) + 1;
    char* pszTarget = new char[nTargetSize];

    // first copy the first string into the target
    int nT;
    for(nT = 0; szSrc1[nT] != '\0'; nT++)
    {
        pszTarget[nT] = szSrc1[nT];
    }

    // now copy the contents of the second string onto
    // the end of the first
    for(int nS = 0; szSrc2[nS] != '\0'; nT++, nS++)
    {
        pszTarget[nT] = szSrc2[nS];
    }

    // add the terminator to szTarget
    pszTarget[nT] = '\0';

    // return the results to the caller
    return pszTarget;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // Prompt user
    cout << "This program accepts two strings\n"
         << "from the keyboard and outputs them\n"
         << "concatenated together.\n" << endl;

    // input two strings
    cout << "Enter first string: ";
    char szString1[256];
    cin.getline(szString1, 256);

    cout << "Enter the second string: ";
    char szString2[256];
    cin.getline(szString2, 256);
```

```
    // now concatenate one onto the end of the other
    cout << "Concatentate second string onto the first"
         << endl;
    char* pszT = concatenateString(szString1, szString2);

    // and display the result
    cout << "Result: <"
         << pszT
         << ">" << endl;

    // return the memory to the heap
    delete[] pszT;
    pszT = NULL;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This program includes the `#include` file `cstring` to gain access to the `strlen()` function. The `concatenateString()` function is similar to the earlier versions, except that it returns the address of a block of heap memory containing the concatenated string rather than modify either of the strings passed to it.

Declaring the arguments as `const` means that the function promises not to modify them. This allows the function to be called with a `const` string as in the following snippet:

```
char* pFullName = concatenateString("Mr. ", pszName);
```

The string `"Mr. "` is a `const` character array in the same sense that 1 is a `const` integer.

The first statement within `concatenateString()` calculates the size of the target array by calling `strlen()` on both source strings and adding 1 for the terminating `null`.

The next statement allocates an array of that size off of the heap using the `new` keyword.

The two `for` loops work exactly like those in the earlier concatenate examples by copying first `szSrc1` into the `pszTarget` array and then `szSrc2` before tacking on the final terminating `null`.

The function then returns the address of the `pszTarget` array to the caller.

The `main()` function works the same as in the earlier Concatenate program by prompting the user for two strings and then displaying the concatenated result. The only difference is that this version returns the pointer returned by `concatenateString()` to the heap before terminating by executing the following snippet:

```
delete pszT;
pszT = NULL;
```

The output from running this program is indistinguishable from its earlier cousins:

```
This program accepts two strings
from the keyboard and outputs them
concatenated together.

Enter first string: this is a string
Enter the second string: THIS IS ALSO A STRING
Concatentate second string onto the first
Result: <this is a stringTHIS IS ALSO A STRING>
Press any key to continue . . .
```

The subject of C++ pointers is too vast to be handled in a single chapter. The next chapter examines the relationship between arrays and pointers, a topic I glossed over in the final example programs in this chapter.

Chapter 18

Taking a Second Look at C++ Pointers

In This Chapter

- Defining operations on a pointer
- Comparing pointer addition with indexing an array
- Extending arithmetic to different types of pointers
- Sorting out constant pointers from pointers to constants
- Reading the arguments to a program

Chapter 17 introduced the concept of a pointer variable as a variable designed to contain the address of another variable. I even went so far as to suggest a couple of uses for pointer variables. However, you've only begun to see the myriad ways that pointer variables can be used to do some pretty cool stuff and really confuse you at times as well.

This chapter examines carefully the relationship between pointers and arrays, a topic that I brushed over in the last chapter.

Pointers and Arrays

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

Operations on pointers

Table 18-1 lists the three fundamental operations that are defined on pointers.

Table 18-1 Three Operations Defined on Pointer Type Variables

Operation	*Result*	*Meaning*
`pointer + offset`	pointer	Calculate the address of the object offset entries from the pointer
`pointer++`	pointer	Move the pointer over one entry
`pointer2 - pointer1`	offset	Calculate the number of entries between pointer2 and ponter1

Although not listed in Table 18-1, operations that are related to addition, such as `pointer += offset`, are also defined. Subtraction is defined as well, since it is merely a variation on addition.

The simple memory model used to explain pointers in Chapter 17 will work here to explain how these operations work. Consider an array of 32 one-byte characters called `cArray`. If the first byte of this array is stored at address 0x1000, then the last location will be at 0x101F. While `cArray[0]` will be at 0x1000, `cArray[1]` will be at 0x1001, `cArray[2]` at 0x1002, and so forth.

Now assume a pointer `pArray` is located at location 0x1100. After executing the expression

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

the pointer `pArray` will contain the value 0x1000 (see Figure 18-1). By the way, you read this as "`pArray` gets the address of `cArray` sub 0."

Adding a value `n` to `pArray` generates the address of `cArray[n]`. For example, consider the case where `n` equals 2. In that case, `pArray + 2` generates the address 0x1002, which is the address of `cArray[2]`. This correspondence is demonstrated in Table 18-2. Figure 18-2 shows this graphically.

Table 18-2 The Correspondence between Pointer Offsets and Array Elements

Offset	*Result*	*Corresponds to...*
+ 0	0x1000	`cArray[0]`
+ 1	0x1001	`cArray[1]`
+ 2	0x1002	`cArray[2]`
...	...	`...`
+ n	0x1000 + n	`cArray[n]`

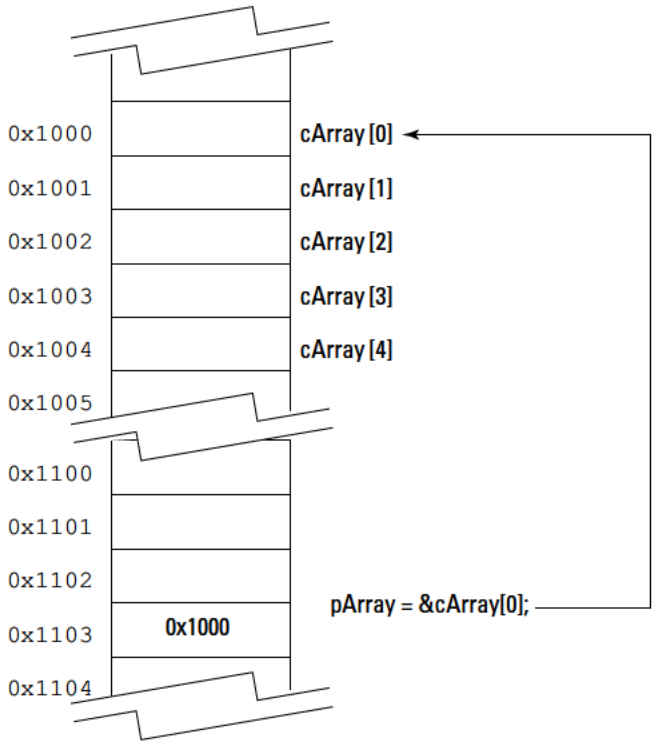


Figure 18-1: After the assignment `pArray = &cArray[0]` the pointer `pArray` points to the beginning of the array `cArray`.

Pointer addition versus indexing into an array

The claim

```
pArray = &cArray[0];
*(pArray + 2) = 'c';
```

is the same as

```
cArray[2] = 'c';
```

Before you can respond to this claim, I need to explain how to read the first code snippet. Take it one step at a time. You already know to read the first expression: `pArray = &cArray[0]` means "`pArray` gets the address of `cArray` sub 0."

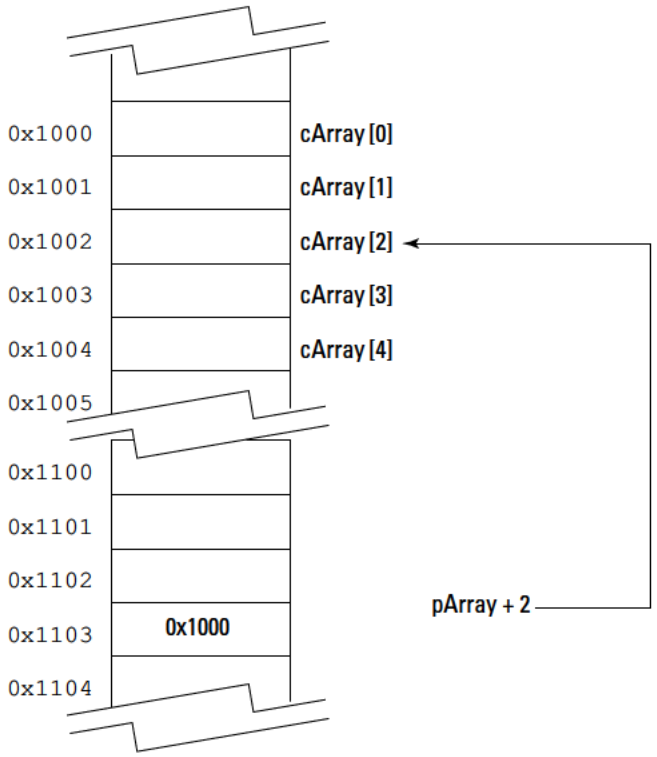


Figure 18-2: If `pArray` points to the beginning of `cArray`, then `pArray + 2` points to `cArray[2]`.

To interpret the second expression, remember that `pArray + 2` generates the value 0x1002, and it is of type `char*`. `*(pArray + 2)` on the left-hand side of an assignment operator says, "store a `'c'` in the `char` pointed at by `pArray + 2`." This is demonstrated graphically in Figure 18-3.

The parentheses around `*(pArray + 2)` are necessary because unary `*` has higher precedence than addition. The expression `*pArray + 2` retrieves the character pointed at by `pArray` and adds 2 to it. Adding the parentheses forces the addition to occur first and the unary operator to be applied to the result.

In fact (here comes the kicker), the correspondence between the two forms of expression is so strong that C++ considers `cArray[n]` nothing more than a shorthand for `*(pArray + n)` where `pArray` points to the first element in `cArray`:

```
cArray[n] is interpreted as *(&cArray[0] + n)
```

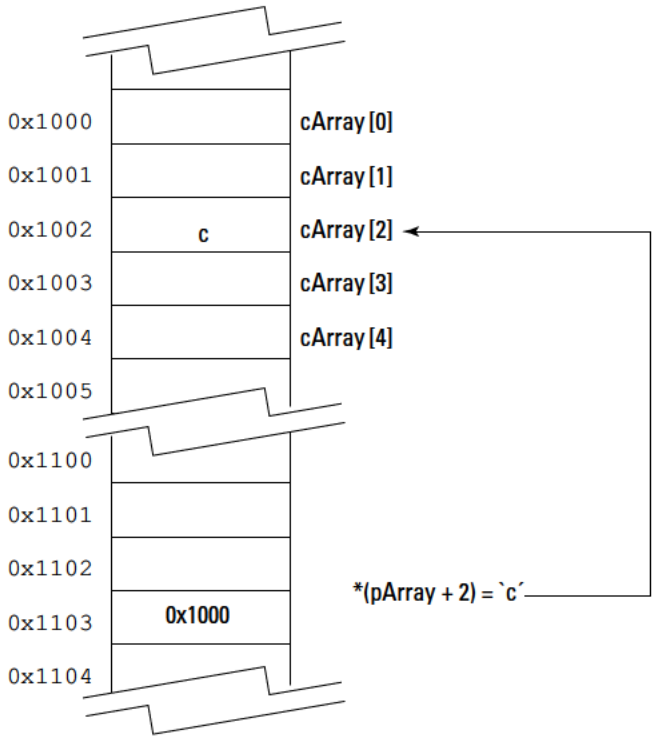


Figure 18-3: The expression `*(pArray + 2) = 'c'` stores a `'c'` in `cArray[2]`.

To complete this association, C++ takes another shortcut by making the second, following interpretation:

```
cArray is interpreted as &cArray[0]
```

That is, an array name when it appears without a subscript is interpreted as the address of the first element of the array; thus the following:

```
cArray[n] is interpreted as *(cArray + n)
```

In fact, the C++ compiler considers the expression on the left nothing more than some human shorthand for the expression on the right.

So, if I can treat the name of an array as though it were a pointer (which it is, by the way), can I use the index operator on pointer variables? Absolutely. Thus, the following is perfectly legal:

```
char cArray[256];
char* pArray = cArray;
pArray[2] = 'c';
```

That is how I was able to write expressions like the following in Chapter 17:

```
int nTargetSize = strlen(szSrc1) + strlen(szSrc2) + 1;
char* pszTarget = new char[nTargetSize];

// first copy the first string into the target
int nT;
for(nT = 0; szSrc1[nT] != '\0'; nT++)
{
    pszTarget[nT] = szSrc1[nT];
}
```

The variable `pszTarget` is declared as `char*` (read "pointer to a char") because that's what `new char[nTargetSize]` returns. The subsequent `for` loop assigns values to elements in this array using the expression `pszTarget[nT]`, which is the same as accessing `char` elements pointed at by `pszTarget + nT`.

By the way, the `psz` prefix is the naming convention for "pointer to an ASCIIZ string." An ASCIIZ string is a character array that ends with a terminating `null` character.

Using the pointer increment operator

The following is what you might call the pointer arithmetic version of the `concatenateString()` function from the ConcatenateHeap program from Chapter 17. This version is part of the program ConcatenatePtr on the enclosed CD-ROM.

In fact, you were dealing with pointer arithmetic in Chapter 17 as well, but the pointer arithmetic was written using array indexing.

C++ programmers love their pointers. The following explicit pointer version of `concatenateString()` is much more common than the array index version in Chapter 17.

```
// concatenateString - concatenate two strings together
//                     into an array allocated off of the
//                     heap
char* concatenateString(const char* pszSrc1,
                        const char* pszSrc2)
{
    // allocate an array of sufficient length
    int nTargetSize = strlen(pszSrc1)+strlen(pszSrc2)+1;
```

```
    char* pszTarget = new char[nTargetSize];

    // first copy the first string into the target
    char* pszT = pszTarget;
    for(; *pszSrc1 != '\0'; pszT++, pszSrc1++)
    {
        *pszT = *pszSrc1;
    }

    // now copy the contents of the second string onto
    // the end of the first
    for(; *pszSrc2 != '\0'; pszT++, pszSrc2++)
    {
        *pszT = *pszSrc2;
    }

    // add the terminator to szTarget
    *pszT = '\0';

    // return the unmodified address of the array
    // to the caller
    return pszTarget;
}
```

This version of `concatenateString()` starts out exactly like the earlier ConcatenateHeap version from Chapter 17. The difference between this version and its predecessor lies in the two `for` loops. The version in Chapter 17 left the pointer to the target array, `pszTarget`, unchanged and incremented an index into that array.

The version that appears here skips the intermediate step of incrementing an index and simply increments the pointer itself. First, it checks to make sure that `pszSrc1` doesn't already point to the `null` character that indicates the end of the source character string. If not, the assignment within the `for` loop

```
*pszT = *pszSrc1;
```

says retrieve the character pointed at by `pszSrc1` and store it into the location pointed at by `pszT`. This is demonstrated graphically in Figure 18-4.

The increment clause of the `for` loop

```
pszT++, pszSrc1++
```

increments both the source pointer, `pszSrc1`, and target pointer, `pszT`, to the next character in the source and destination arrays. This is demonstrated by Figure 18-5.

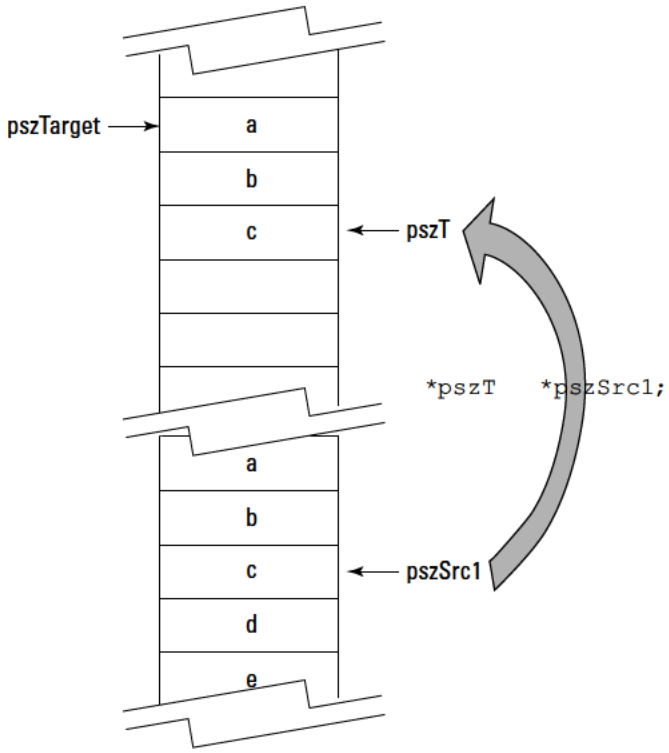


Figure 18-4: The expression `*pszT = *pszSrc1` copies the character pointed at by `pszSrc1` to the location pointed at by `pszT`.

The remainder of the program is identical to its Chapter 17 predecessor, and the results from executing the program are identical as well:

```
This program accepts two strings
from the keyboard and outputs them
concatenated together.

Enter first string: this is a string
Enter the second string: SO IS THIS
Concatentate first string onto the second
Result: <this is a stringSO IS THIS>
Press any key to continue . . .
```

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 `concatenateString()` have over the easier-to-read index version?

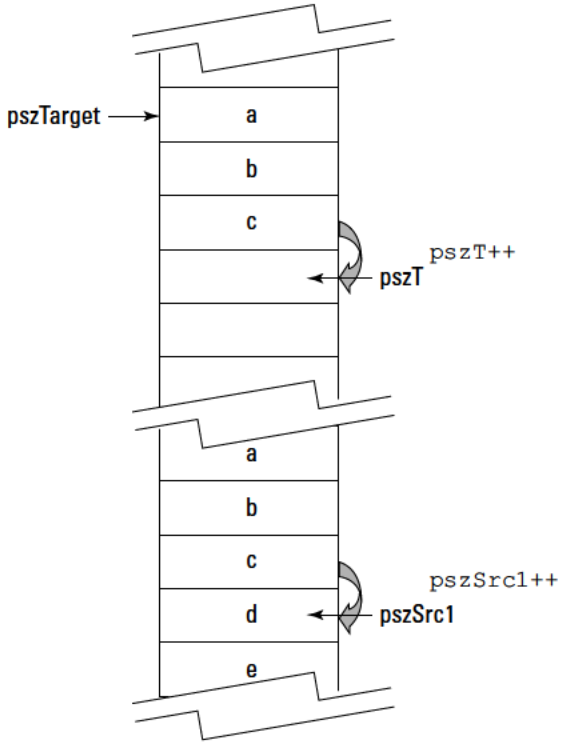


Figure 18-5: The increment clause of the `for` loop increments both source and destination pointers to the next location in the array.

Note: "Easier-to-read" is a matter of taste. To a seasoned C++ programmer, the pointer version is just as easy to fathom as the index version.

The answer is partially historic and partially human nature. As complicated as it might appear to the human reader, a statement such as `*pszT = *pszSrc1` can be converted into an amazingly small number of machine instructions. Older computer processors were not very fast by today's standards. When C, the progenitor of C++, was introduced to the world some 40 years ago, saving a few computer instructions was a big deal. Pointer arithmetic gave C a big advantage over other languages of the day, notably Fortran, which did not offer pointer arithmetic. This, more than any other single feature, did more to advance C and later C++ over its competitors.

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 scanning arrays with indices.

Don't fall into the trap of cramming as much as you can into a single C++ statement, thinking that a few C++ source statements will generate fewer machine instructions that will, therefore, execute faster. In the old days, when compilers were simpler, that may have worked, but today there is no obvious relationship between the number of C++ instructions and the number of machine instructions generated. For example, the expression

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

does not necessarily generate machine instructions that are any different from the following expression that is both easier to read and easier to debug:

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

Today's optimizing compilers generate minimal amounts of code.

Operations on Different Pointer Types

It's not too hard to convince yourself that `pszTarget + n` points to `pszTarget[n]` when each element in the array is 1 byte in length as is the case for `char` strings. After all, if `cArray` is located at 0x1000, then `cArray[5]` must be at 0x1005.

It is not so obvious that pointer addition works for arrays of objects other than 1-byte characters. Consider an array `nArray` of `int`s. Since an `int` occupies 4 bytes in Code::Blocks/gcc, if `nArray` is located at 0x1000, then `nArray[5]` will be located at 0x1000 + (5 * 4) or 0x1014.

Hexadecimal 0x14 is equal to 20 decimal.

Fortunately for us, in C++, `array + n` points to `array[n]` no matter how large a single element of array might be. C++ makes the necessary conversions to ensure that this relationship is true.

Constant Nags

Chapter 14 introduced the concept of `const` variables. For example, the following

```
const double PI = 3.14159;
```

declares a constant variable PI. Constant variables must be initialized when created and cannot be changed later just like numbers like 2 and 3.14159.

The concept of const-ness can be applied to pointers as well, but the question is, where does the const keyword go? Consider the following three declarations. Which of these are legal?

```
const char* pszArray1;
char const* pszArray2;
char* const pszArray3;
```

It turns out all three are legal, but one of them has a different meaning than the other two. The first two variables, pszArray1 and pszArray2, are both pointers to constant char arrays. This means that you can modify the pointers, but you cannot modify the characters that they point at. Thus, the following is legal:

```
pszArray1 = new char[128];    // this is OK
```

But the following is not:

```
(*pszArray1) = 'a';           // not legal
```

By comparison, pszArray3 is a constant pointer to a char array. In this case, you cannot change the pointer once it has been declared. Therefore, you must initialize it when declared since you won't get a chance later as in the following:

```
char* const pszArray3 = new char[128];
```

Once declared, the following is not legal:

```
pszArray3 = pszArray1;        // not legal - you
                              // can't change pszArray3
```

But you can change the characters that it points to, like this:

```
char* const pszArray3 = new char[128];
(*pszArray3) = 'a';           // legal
```

A single pointer can be both constant and point to constant characters:

```
const char* const pszMyName = "Stephen";
```

The value of this pointer cannot be changed nor can the characters that it points to.

As a beginning programmer, do you really need to worry about all these constant declarations? The answer is, "Sometimes." You will get a warning if you do the following:

```
char* pszMyName = "Stephen";
```

Because you could conceivably try to modify my name by putting `*pszMyName` (or the equivalent `pszMyName[n]`) on the left-hand side of an assignment operator. The proper declaration is

```
const char* pszMyName = "Stephen";
```

Differences Between Pointers and Arrays

With all the similarities, one might be tempted to turn the question around and ask, "What's the difference between a pointer and the address of an array?" There are basically two differences:

- An array allocates space for the objects; a pointer does not.
- A pointer allocates space for the address; an array does not.

Consider these two declarations:

```
int nArray[128];
int* pnPtr;
```

Both `nArray` and `pnPtr` are of type pointer to `int`, but `nArray` also allocates space for 128 `int` objects, whereas `pnPtr` does not. You can consider `nArray` to be a constant address in the same way that 3 is a constant `int`. You can no more put `nArray` on the left-hand side of an assignment than you can 3. The following is not allowed:

```
nArray = pnPtr;  // not allowed
```

Thus, `pnPtr` is of type `int*`, whereas `nArray` is actually of type `int* const`.

My main() Arguments

Now you've come far enough to learn the last secret of the program template that you've been using: What are the arguments to `main()`?

```
int main(int nNumberOfArgs, char* pszArgs[])
```

These point to the arguments of the program. The first argument is the number of arguments to the program, including the name of the program itself. The second argument is an array of pointers to character strings representing the arguments themselves. Arrays of pointers? What?

Arrays of pointers

If a pointer can point to an array, then it seems only fitting that the reverse should be true as well. Arrays of pointers are a type of array of particular interest.

The following declares an array of ten pointers to integers:

```
int* pInt[10];
```

Given this declaration, then `pInt[0]` is a pointer to an integer. The following snippet declares an array of three pointers to integers and assigns them values:

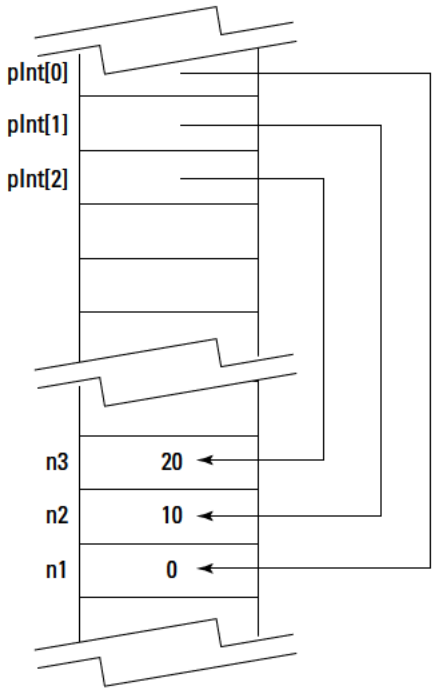
```
void fn()
{
    int n1, n2, n3;
    int* pInts[3] = {&n1, &n2, &n3};

    for(int n = 0; n < 3; n++)
    {
        // initialize the integers
        *pInts[n] = n * 10;
    }
}
```

After the declaration, `pInts[0]` points to the variable `n1`, `pInts[1]` points to `n2`, and `pInts[2]` points to `n3`. Thus, an expression like

```
*pInts[1] = 10;
```

sets the `int` pointed at by `pInts[1]` (that would be `n2`) to 10. The effect of the `for` loop in the prior snippet is to initialize `n1`, `n2`, and `n3` to 0, 10, and 20, respectively. This is shown graphically in Figure 18-6.



The effects of executing the following:

```
int n1, n2, n3;

int* pInt[3]    {&n1, &n2, &n3};
for(int n    0; n < 3, n++)
{
    *pInt[n]    n * 10;
}
```

Figure 18-6: The effects of setting up and using an array of three pointers to integers.

Arrays of arguments

Returning to the `main()` example, the arguments to the program are the strings that are passed to the program when it is executed. Thus, if I execute `MyProgram` as

```
MyProgram file1 file2 /w
```

the arguments to the program are `file1`, `file2`, and `/w`.

Although technically not an argument, C++ includes the name of the program as the first "argument."

Switches are not interpreted, so `/w` is passed to the program as an argument. However, the special symbols `<", ">` and `|` are interpreted by the command line interpreter and are not passed to the program.

The following simple PrintArgs program displays the arguments passed to it by the command line interpreter:

```
// PrintArgs - print the arguments to the program
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int nNumberofArgs, char* pszArgs[])
{
    for(int n = 0; n < nNumberofArgs; n++)
    {
        cout << "Argument " << n
             << " is <" << pszArgs[n]
             << ">" << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

Now the trick is how to pass arguments to the program.

Passing arguments to your program through the command line

The easiest and most straightforward way is to simply type the arguments when executing the program from the command line prompt:

```
PrintArgs file1 file2 /w
```

Doing so generates the following output:

```
C:\Beginning_Programming-CPP\PrintArgs\bin\Debug>PrintArgs file1 file2 /w
Argument 0 is <printargs>
Argument 1 is <file1>
Argument 2 is <file2>
Argument 3 is </w>
Press any key to continue . . .
```

The difficulty to this approach is knowing where the executable is stored. Code::Blocks creates the executable program during the Build step in a subdirectory of the directory containing the project. Whether you used the default installation location shown in the preceding code or not, you can always find the project directory by selecting Project➪Properties. The default Project Settings tab of the dialog box that pops up displays the path to the project file, as shown in Figure 18-7.

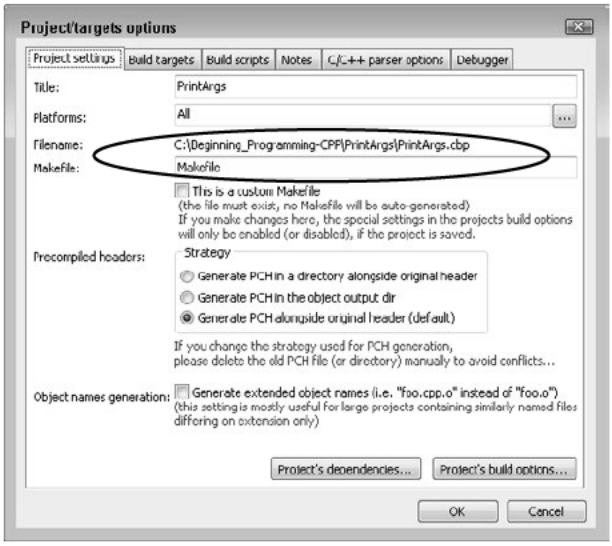


Figure 18-7: The Code::Blocks Project Settings tab of the Project/Target Options dialog box contains the path to the project file.

Select the Build Targets tab to find the path to the executable file, as shown in Figure 18-8.

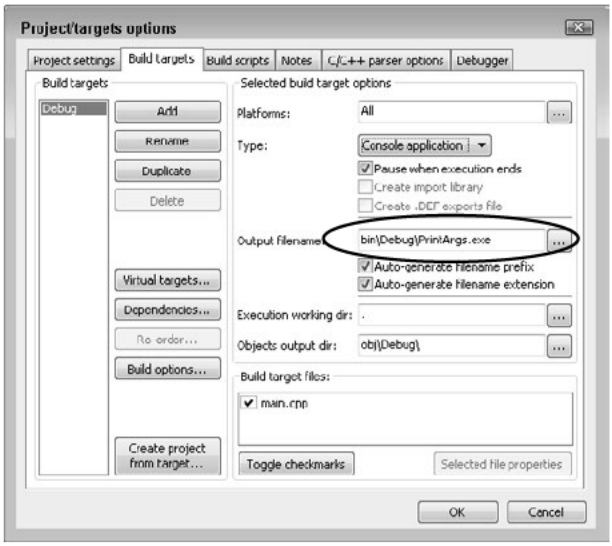


Figure 18-8: The Build Targets tab indicates the name and location of the executable.

If you are using Windows, open an MS-DOS window by selecting Start➪Programs➪Accessories➪Command Prompt (this is for Windows XP and Vista; the details differ slightly depending upon which version of Windows you are using). Navigate to the proper window using the CD command ("CD" stands for Change Directory).

Using the directory path provided in Figure 18-7, I would enter the following:

```
CD \Beginning_Programming-CPP\PrintArgs\bin\Debug
PrintArgs file1 file2 /w
```

The details for Linux and Macintosh will be slightly different but similar.

Passing arguments to your program from the Code::Blocks environment

You can pass arguments to your program from within Code::Blocks itself by selecting Project➪Set Projects' Arguments. This opens the dialog box shown in Figure 18-9. Enter the arguments into the Program Arguments entry field.

Figure 18-9: You can set up the project to pass arguments to the program when executed from Code::Blocks.

Executing the program from Code::Blocks opens a command line window with the following contents:

```
Argument 0 is <C:\Beginning_Programming-CPP\PrintArgs\bin\Debug\PrintArgs.exe>
Argument 1 is <file1>
Argument 2 is <file2>
Argument 3 is </w>
Press any key to continue . . .
```

This technique is a lot easier, but it works only from within the Code::Blocks environment. However, this is the only way to pass arguments to your program when using the Code::Blocks debugger. I talk about the debugger in Chapter 20.

Passing arguments to your program through Windows

In Windows, there is one final way of passing arguments to a program. Windows executes a program with no arguments if you double-click the name of the executable file. However, if you drag a set of files and drop them on the program's executable filename, Windows executes the program, passing it the name of the files as its arguments.

To demonstrate, I created a couple of dummy files in the same directory as the PrintArg.exe file called `file1.txt` and `file2.txt`, as shown in Figure 18-10.

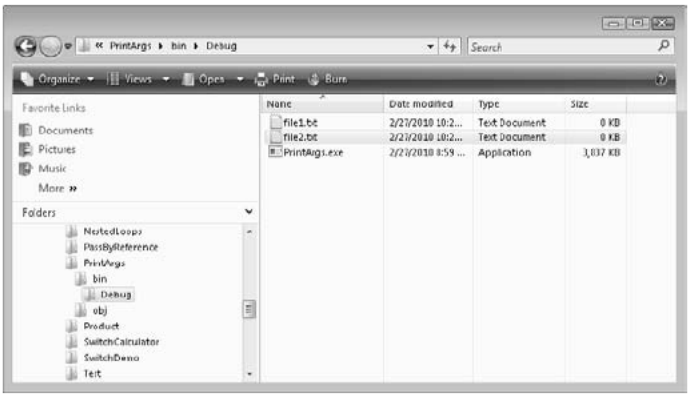


Figure 18-10: I created two dummy files in the same directory as the PrintArgs.exe executable.

I then selected both files and dragged and dropped them onto the PrintArgs.exe filename. Figure 18-11 shows the result.

Windows does not pass the filenames to the program in any particular order. In particular, it does not necessarily pass them in the order that they appear in the directory list or the order that you selected them.

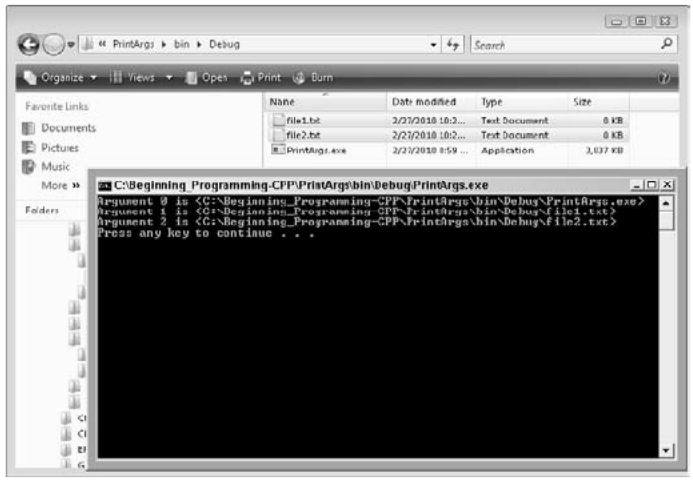


Figure 18-11: Dropping the two filenames on the PrintArgs.exe filename instructs Windows to launch the program and pass the name of the files as arguments to the program.

This chapter and its predecessor are not easy for a beginner. Don't despair if you are feeling a little uncertain right now. You may need to reread this section. Make sure that you understand the examples and the demonstration programs. You should find yourself growing more and more comfortable with the concept of pointer variables as you make your way through the remainder of the book.

Chapter 19

Programming with Class

In This Chapter

- Grouping data using parallel arrays
- Grouping data in a class
- Declaring an object
- Creating arrays of objects

Arrays are great at handling sequences of objects of the same type, such as `int`s or `double`s. Arrays do not work well, however, when grouping different types of data such as when we try to combine a Social Security number with the name of a person into a single record. C++ provides a structure called the *class* (or *struct*) to handle this problem.

Grouping Data

Many of the programs in earlier chapters read a series of numbers, sometimes into an array, before processing. A simple array is great for standalone values. However, many times (if not most of the time), data comes in groups of information. For example, a program may ask the user for his first name, last name, and Social Security number. Alone, any one of these values is not sufficient — only in the aggregate do the values make any sense.

You can store associated data of different types in what are known as *parallel arrays*. For example, I might use an array of strings called `pszFirstNames` to hold people's first names, a second `pszLastNames` to hold the last names, and a third `nSocialSecurities` to hold the corresponding Social Security number. I would store the data such that any given index `n` points to the data for a given individual.

Thus, my personal data might be at offset 3. In that case, `szFirstNames[3]` would point to "Stephen," `szLastNames[3]` would point to "Davis," and `nSocialSecurityNumbers[3]` would contain . . . well, you get the idea. This is shown in Figure 19-1.

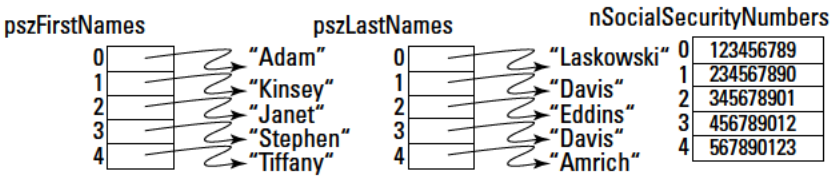


Figure 19-1: Parallel arrays are sometimes used to hold collections of related but dissimilar data in languages that don't support classes.

This method works, but it's prone to errors since there's nothing that directly associates the first name with the last name and the Social Security number other than an index. You could easily imagine that a missing instruction here or there, and I would become "Stephen Eddins" or any other random combination of first and last names.

Fortunately for us, C++ provides a better way.

The Class

A first name or a Social Security number doesn't make any sense except in the context of the person to whom they belong — data like that must have a context created by its association with other, related data. What we would like is to be able to create a structure, say `Person`, that contains all of the relevant properties that make up a person (in this case, first name, last name, and social security number).

C++ uses a structure known as the *class* that has the following format:

```
class Person
{
  public:
    char szFirstName[128];
    char szLastName[128];
    int  nSocialSecurityNumber;
};
```

A class definition starts with the keyword `class` followed by the name of the class and an open brace.

The naming rules for class names are the same as for variable names: The first letter must be one of the letters 'a' through 'z' or 'A' through 'Z' or underscore. Every subsequent character in the name must be one of these or the digits '0' through '9'. By convention, class names always start with an uppercase letter. Class names normally consist of multiple words jammed together, with each word starting with an uppercase letter.

The first keyword within the open brace in the early examples will always be `public`. I'll describe the alternatives to `public` in Chapter 24, but just accept it as part of the declaration for now.

You can also use the keyword `struct` instead of `class`. A `struct` is identical to a `class` in every respect except that the `public` is assumed in a `struct`. For historical reasons, the term `class` is more popular in C++, while the term `struct` is used more often in C programs.

Following the public keyword are the declarations for the entries it takes to describe the class. The `Person` class contains two arrays for the first and last names and a third entry to hold the Social Security number.

The entries within a class are known as *members* or *properties* of the class.

The Object

Declaring a class in C++ is like defining a new variable type. You can create a new instance of a class as follows:

```
Person me;
```

An instance of a class is called an *object*.

People get confused about the difference between a class and an object; sometimes people even use the terms interchangeably. Actually, the difference is easy to explain with an example. `Dog` is a class. My dog, Lollie, is an instance of a dog. My other dog, Jack, is a separate, independent instance of a dog. `Dog` is a class; `lollie` and `jack` are objects.

You can access the members of an object by including their name after the name of the object followed by a dot, as in the following:

```
Person me;
me.nSocialSecurityNumber = 456789012;
cin >> me.szLastName;
```

Here `me` is an object of class `Person`. The element `me.nSocialSecurityNumber` is a member or property of the `me` object. The type of `me` is `Person`. The type of `me.nSocialSecurityNumber` is `int`, and its value is set to 456-78-9012. The type of `me.szLastName` is `char[]` (pronounced "array of char").

A class object can be initialized when it is created as follows:

```
Person me = {"Stephen", "Davis", 456789012};
```

Assignment is the only operation defined for user-defined classes by default. Its use is shown here:

```
Person copyOfMe;
copyOfMe = me;   // copy each member of me to copyOfMe
```

The default assignment operator copies the members of the object on the right to the members on the left. The objects on the right and left of the assignment operator must be exactly the same type.

You can define what the other operators might mean when applied to an object of a class that you define. That is considered advanced strokes, however, and is beyond the scope of this book.

Arrays of Objects

You can declare and initialize arrays of objects as follows:

```
Person people[5] = {{   "Adam", "Laskowski", 123456789},
                    { "Kinsey",     "Davis", 234567890},
                    {  "Janet",    "Eddins", 345678901},
                    {"Stephen",     "Davis", 456789012},
                    {"Tiffany",    "Amrich", 567890123}};
```

The layout of `people` in memory is shown in Figure 19-2. Compare this with the parallel array equivalent in Figure 19-1.

In this example, each one of the elements of the array `people` is an object. Thus, `people[0]` is the first object in the array. My information appears as `people[3]`. You can access the members of an individual member of an array of objects using the same "dot-member" syntax as that used for simple objects:

```
// change my social security number
people[3].nSocialSecurityNumber = 456789012;
```

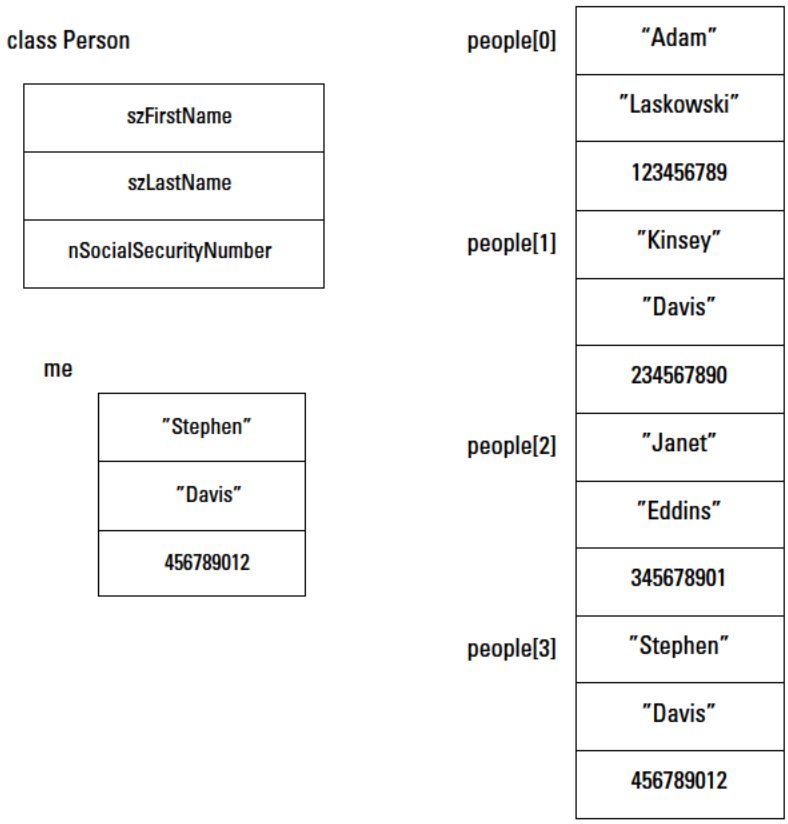


Figure 19-2: The arrangement in memory of an array of 5 `Person` objects.

The type of `people` is `Person[]`, which is read "array of `Person`" (sometimes programmers use the plural of the class name as in "array of `Persons`"). The type of `people[3]` is `Person`.

Looking at an Example

I've gone far enough without an example program to demonstrate how class objects appear in a program. The following InputPerson program inputs the data for an array of people. It then sorts the array by Social Security number and outputs the sorted list.

The sorting algorithm I used is known as a *Bubble Sort*. It isn't particularly efficient, but it's very simple to code. I explain how it works in a sidebar, but don't get wrapped up in the details of the Bubble Sort. Focus instead on how the program inputs the critical elements of a `Person` into a single element of an array that it can then manipulate as a single entity.


```
// InputPerson - create objects of class Person and
//               display their data
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// Person - stores the name and social security number
class Person
{
  public:
    char szFirstName[128];
    char szLastName[128];
    int  nSocialSecurityNumber;
};

// getPerson - read a Person object from the keyboard
//             and return a copy to the caller
Person getPerson()
{
    Person person;

    cout << "\nEnter another Person\n"
         << "First name: ";
    cin  >> person.szFirstName;

    cout << "Last name: ";
    cin  >> person.szLastName;

    cout << "Social Security number: ";
    cin  >> person.nSocialSecurityNumber;

    return person;
}

// getPeople - read an array of Person objects;
//             return the number read
int getPeople(Person people[], int nMaxSize)
{
    // keep going until operator says he's done or
    // until we're out of space
    int index;
    for(index = 0; index < nMaxSize; index++)
    {
        char cAnswer;
        cout << "Enter another name? (Y or N):";
        cin  >> cAnswer;

        if (cAnswer != 'Y' && cAnswer != 'y')
        {
```

```
            break;
        }

        people[index] = getPerson();
    }
    return index;
}

// displayPerson - display a person on the default display
void displayPerson(Person person)
{
    cout << "First name: " << person.szFirstName << endl;
    cout << "Last name : " << person.szLastName  << endl;
    cout << "Social Security number : "
         << person.nSocialSecurityNumber << endl;
}

// displayPeople - display an array of Person objects
void displayPeople(Person people[], int nCount)
{
    for(int index = 0; index < nCount; index++)
    {
        displayPerson(people[index]);
    }
}

// sortPeople - sort an array of nCount Person objects
//              by Social Security Number
//              (this uses a binary sort)
void sortPeople(Person people[], int nCount)
{
    // keep going until the list is in order
    int nSwaps = 1;
    while(nSwaps != 0)
    {
        // we can tell if the list is in order by
        // the number of records we have to swap
        nSwaps = 0;

        // iterate through the list...
        for(int n = 0; n < (nCount - 1); n++)
        {
            // ...if the current entry is greater than
            // the following entry...
            if (people[n].nSocialSecurityNumber >
                people[n+1].nSocialSecurityNumber)
            {
                // ...then swap them...
                Person temp  = people[n+1];
                people[n+1] = people[n];
```

```
                people[n]   = temp;

                // ...and count it.
                nSwaps++;
            }
        }
    }
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // allocate room for some names
    Person people[128];

    // prompt the user for input
    cout << "Read name/social security information\n";
    int nCount = getPeople(people, 128);

    // sort the list
    sortPeople(people, nCount);

    // now display the results
    cout << "\nHere is the list sorted by "
         << "social security number" << endl;
    displayPeople(people, nCount);

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The program starts by declaring class `Person` with data members for the first name, last name, and Social Security number. Contrary to good programming practice, this program uses fixed-length arrays for the name strings. (If I were writing this for a commercial package, I would use variable length arrays, or I would include a test to make sure that input from the keyboard did not overflow the buffer. See Chapter 17 if you don't know what I'm talking about.)

The first function, `getPerson()`, prompts the user for the data necessary to describe a single `Person` object. It then returns a copy of that `Person` to the caller.

The second function, `getPeople()`, invokes the `getPerson()` function repeatedly to retrieve the data for a number of individuals. It stores the `Person` objects retrieved into the array `people`. This function accepts as an argument the maximum size of the `people` array and returns to the caller the actual number of elements stored there.

The `displayPerson()` and `displayPeople()` functions are the output analogs to the `getPerson()` and `getPeople()` functions. `displayPerson()` outputs the information for a single individual, whereas `displayPeople()` calls that function on each element defined in the people array.

The `sortPeople()` function sorts the elements of the people array in order of increasing Social Security number. This function is described in the "Bubble Sort" sidebar. Don't worry too much about how this function works. You're way ahead of the game if you can follow the rest of the program.

The output from a test run of this program appears as follows:

```
Read name/social security information
Enter another name? (Y or N):y

Enter another Person
First name: Adam
Last name: Laskowski
Social Security number: 123456789
Enter another name? (Y or N):y

Enter another Person
First name: Stephen
Last name: Davis
Social Security number: 456789012
Enter another name? (Y or N):y

Enter another Person
First name: Janet
Last name: Eddins
Social Security number: 345678901
Enter another name? (Y or N):n

Here is the list sorted by social security number.
First name: Adam
Last name : Laskowski
Social Security number : 123456789
First name: Janet
Last name : Eddins
Social Security number : 345678901
First name: Stephen
Last name : Davis
Social Security number : 456789012
Press any key to continue . . .
```

You've seen most of the non–object-oriented features of C++. The next chapter introduces you to the Code::Blocks debugger, which wraps up the sections dedicated to what I call functional programming. After that, I jump into object-oriented programming in Part V.

Bubble Sort

Most of this book is dedicated to the syntax of C++. However, in addition to the details of the language, you will also need to learn common programming algorithms in order to become a proficient programmer. The Bubble Sort is one of those algorithms that every programmer should master.

There are a number of common algorithms for sorting fields. Each has its own advantages. In general, the simpler algorithms take longer to execute, whereas the really fast algorithms are more difficult to program. The Bubble Sort is very easy to program but isn't particularly fast. This is not a problem for small data sets; arrays up to several thousand entries in length can be sorted in very much less than a second on modern high-speed processors. For small to moderate amounts of data, the simplicity of the Bubble Sort far outweighs any performance penalty.

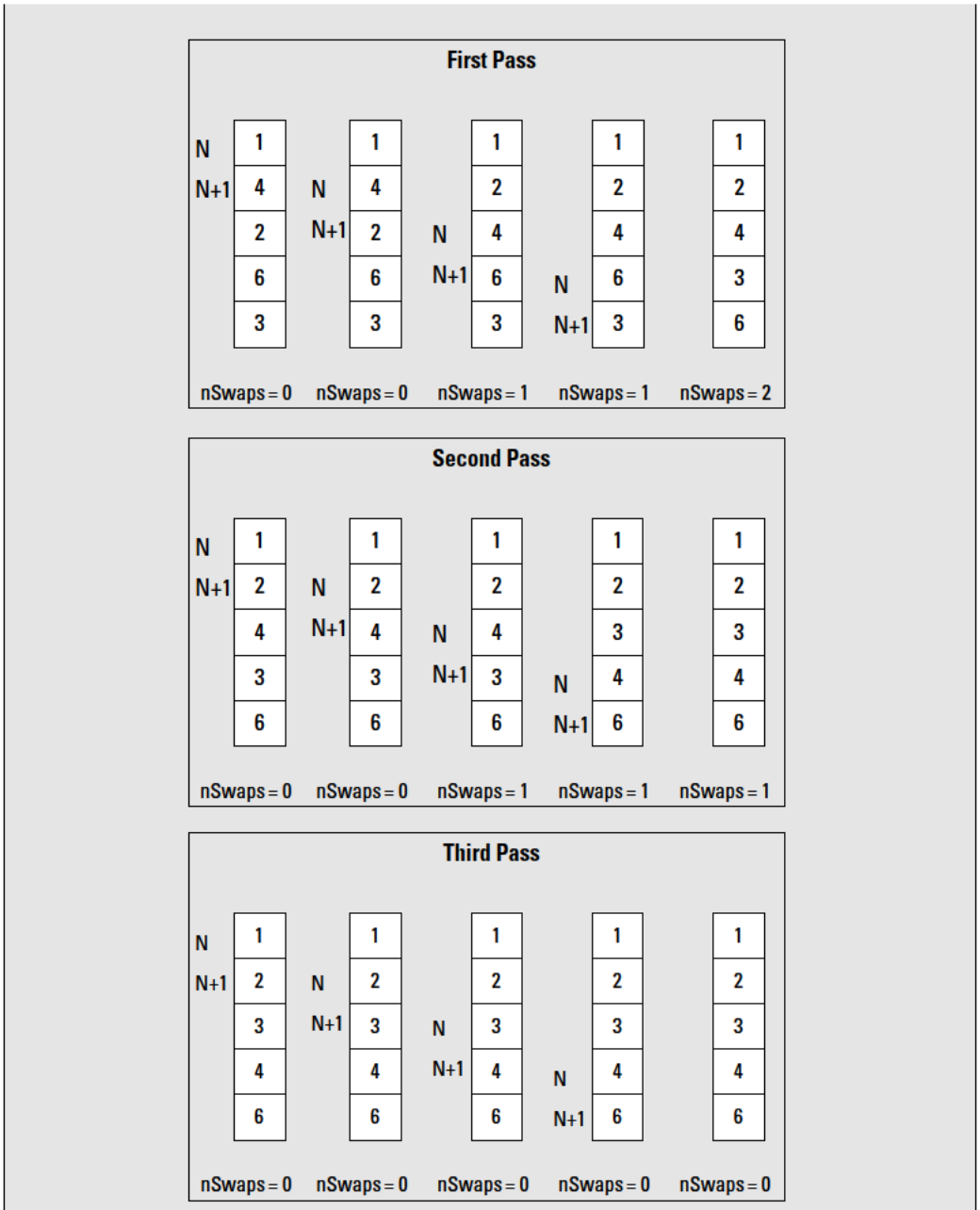
In the Bubble Sort, the program makes multiple passes through the data set. On each pass, it compares each element with the next element in the list. If element N is less than N+1, then these two are in the proper order so the Bubble Sort takes no action. However, if element N is greater than N+1, then the Bubble Sort swaps the two elements and then moves on to the next element. In practice, this looks like the following:

```
// if the current entry is greater than
// the following entry...
if (people[n].nSocialSecurityNumber >
    people[n+1].nSocialSecurityNumber)
{
    // ...then swap them...
    Person temp = people[n+1];
    people[n+1] = people[n];
    people[n]   = temp;

    // ...and count it.
    nSwaps++;
}
```

At the end of the first pass through the entire array, the largest element will have moved to the end of the list, but the rest of the array will still not be in order. However, repeated passes through the array cause each element to "bubble" up to its proper place in the array. The Bubble Sort sets the number of elements that were swapped on each pass by zeroing the counter `nSwaps` before iterating through the list and incrementing the number of elements swapped on each pass. The algorithm doesn't really care how many swaps were executed; if any swaps were executed, then the array was not in order. However, once the Bubble Sort can make it all the way through the list without executing any swaps, then it knows that the array is in order.

The figure demonstrates how the Bubble Sort sorts an array of five integers. During the first pass through the list, two swaps are executed. On the second pass, the algorithm executes only a single swap. The resulting list is in order, but the algorithm doesn't know this until it makes its way all the way through the array without making any swaps, as shown in the third pass. At this point, the Bubble Sort is finished.



Chapter 20

Debugging Your Programs, Part 3

In This Chapter

- Debugging using the built-in debugger
- Building your application with debugger information
- Setting a breakpoint
- Single-stepping your program
- Fixing a sample problem

I introduced a few techniques for finding errors at the end of Parts II (Chapter 8) and III (Chapter 13). Now that you are nearing the end of Part IV, I want to touch on debugging techniques one final time.

In this chapter, I introduce you to the debugging tools built into the Code::Blocks development environment (similar tools exist for most other environments). Learning to use the debugger will give you clear insight into what your program is doing (and what it's not doing, at times).

A New Approach to Debugging

Chapters 8 and 13 demonstrated how to find problems by adding output statements in key positions. Outputting key variables lets you see what intermediate values your program is calculating and what path it's taking through your C++ code.

However, the output technique has several distinct disadvantages. The first is that it's difficult to know what to display. In a small program, such as most of the programs in this book, you can display almost everything — there just aren't that many variables to slug through. However, in a major league program, there may be many hundreds of variables, especially if you include all of the elements in the arrays. Knowing which variables to display can be problematic.

A second problem is the time it takes to rebuild the program. Once again, this isn't a problem with small programs. Code::Blocks can rebuild a small program in just a few seconds. In these cases, adding or changing output statements doesn't take more than a few minutes. However, I have been on projects where rebuilding the entire program took many hours. In a big program, adding new output statements as you zero in on a bug can take a long time.

Finally, it's very difficult to debug a pointer problem using the output approach. If a pointer is invalid, any attempt to use it will cause the program to abort, and knowing a valid pointer from an invalid one simply by displaying its value on `cout` is almost impossible.

The solution

What you need is a way to stop the program in the middle of its execution and query the value of key variables. That's exactly what the debugger does.

The debugger is actually a utility built into the Code::Blocks environment. Every environment has some type of debugger, and they all offer the same basic features though the specific commands may be different. The debugger allows the programmer to control the execution of her program. She can execute one step in the program at a time, she can stop the program at any point, and she can examine the value of variables.

Unlike the C++ language, which is standardized, every debugger has its own command set. Fortunately, once you've learned how to use the Code::Blocks debugger, you won't have any trouble learning to use the debugger that comes with your favorite C++ environment.

The programmer controls the debugger through commands entered from the keyboard within the Code::Blocks environment exactly as she would use the edit commands to modify the C++ source code or build commands to create the executable program. The debug commands are available from both menu items and hot keys.

The best way to learn how to use the Code::Blocks debugger is to use it to find a couple of nasty problems in a buggy version of one of the programs you've already seen.

Entomology for Dummies

The following version of the Concatenate program (which you'll find on the enclosed CD-ROM as ConcatenateError1) represents my first attempt at the ConcatenateHeap program from Chapter 18.

This version has at least two serious bugs, both of which are in the `concatenateString()` function.

```
//
//  ConcatenateError1 - similar to ConcatenatePtr except
//                      this version has several bugs in it
//                      that can be easily found with the
//                      debugger
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;


// concatenateString - concatenate two strings together
//                      into an array allocated off of the
//                      heap
char* concatenateString(const char* pszSrc1,
                        const char* pszSrc2)
{
    // allocate an array of sufficient length
    int nTargetSize = strlen(pszSrc1)+strlen(pszSrc2)+1;
    char* pszTarget = new char[nTargetSize];

    // first copy the first string into the target
    while(*pszSrc1 != '\0')
    {
        *pszTarget++ = *pszSrc1++;
    }

    // now copy the contents of the second string onto
    // the end of the first
    while(*pszSrc2 != '\0')
    {
        *pszTarget++ = *pszSrc2++;
    }

    // return the resulting string to the caller
    return pszTarget;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // Prompt user
    cout << "This program accepts two strings\n"
         << "from the keyboard and outputs them\n"
         << "concatenated together.\n" << endl;

    // input two strings
```

```
    cout << "Enter first string: ";
    char szString1[256];
    cin.getline(szString1, 256);

    cout << "Enter the second string: ";
    char szString2[256];
    cin.getline(szString2, 256);

    // now concatenate one onto the end of the other
    cout << "Concatentate first string onto the second"
         << endl;
    char* pszT = concatenateString(szString1, szString2);

    // and display the result
    cout << "Result: <"
         << pszT
         << ">" << endl;

    // return the memory to the heap
    delete pszT;
    pszT = NULL;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The following shows the results of executing the program:

```
This program accepts two strings
from the keyboard and outputs them
concatenated together.

Enter first string: this is a string
Enter the second string: THIS IS ALSO A STRING
Concatentate first string onto the second
Result: <OF_ƒdT D>
Press any key to continue . . .
```

Clearly, the result is not correct, so something must be wrong. Rather than start inserting output statements, I will use the debugger to find the problems this time.

I suggest that you follow along with me and take the same steps I do in the following section. You can start with the ConcatenateError1 program from the CD-ROM.

Starting the debugger

I can tell the debugger that I want to execute the program up to a certain line or view a particular variable. In order to do that, however, the debugger has to know exactly where each C++ line of code is stored and where each variable is kept. It does this by attaching extra information onto the executable — actually, quite a bit of extra information. Because this information can get really lengthy and because I don't need it for the release version that I ship to the public, including debug information is optional.

I decided whether to include debug information in the executable when I created the project. Figure 20-1 shows the next to last dialog box presented by the Project Wizard, the Console Application dialog box. The default is to generate debug information as shown here. The Release configuration is the version of the executable without the extra debug information. I cannot use the debugger if I do not create a Debug configuration version.

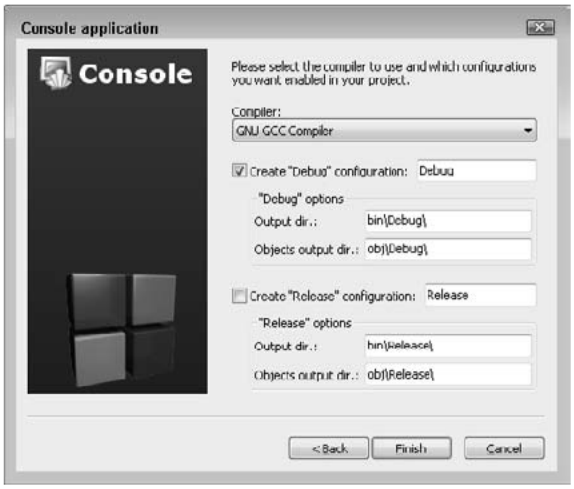


Figure 20-1: The Console Application dialog box of the Project Wizard allows you to select whether to build a debug version of the executable or not.

I can turn debugger information on at any time by selecting Settings➪Compiler and Debugger and then making sure that the Produce Debugging Symbols [-g] check box is checked in the Compiler Flags subwindow of the Compiler Settings window. I have to rebuild the executable by selecting Build➪Rebuild for the change to have any effect.

So assume that I did tell Code::Blocks to include debug information in the executable.

I am reasonably certain that the problem is in the `concatenateString()` function itself. So I decide that I want to stop executing the program at the call to `concatenateString()`. To do this, I need to do what's called *setting a breakpoint*.

A *breakpoint* is a command to the debugger that says stop execution of the program if you get to this spot. There are at least four ways to set a breakpoint, all of which are equivalent:

- Click with the cursor just to the right of the line number on line 60 (see Figure 20-2).
- Right-click on line 60 and select Toggle Breakpoint from the menu that appears (it's the first option).
- Put the cursor on line 60 and select F5 (Toggle Breakpoint).
- Put the cursor on line 60 and select Debug➪Toggle Breakpoint.

Multiple methods exist for entering almost every other debugger command that I describe in this chapter, but in the interest of brevity, I describe only one. You can experiment to find the others.

A small stop sign appears just to the right of the line number, as shown in Figure 20-2.

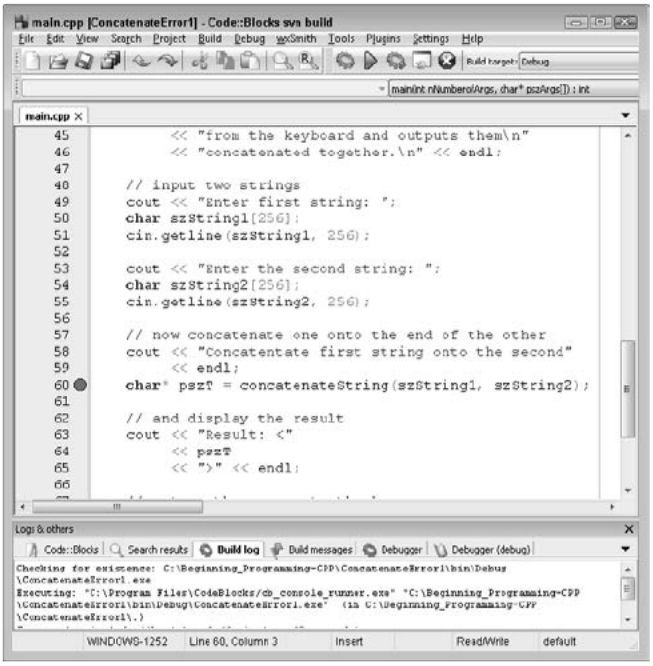


Figure 20-2: A small, red stop sign indicates that a breakpoint has been set at the specified location.

To start the program, I select Debug➪Start. At first the program seems to execute like normal. It first prompts me for the first string. It follows that by prompting me for a second string. As soon as I enter that string, however, the program appears to stop, and a small, yellow arrow appears inside the stop sign on the source code display. This is shown in Figure 20-3. This little, yellow arrow is the *current location indicator*. This points to the next C++ line to be executed.

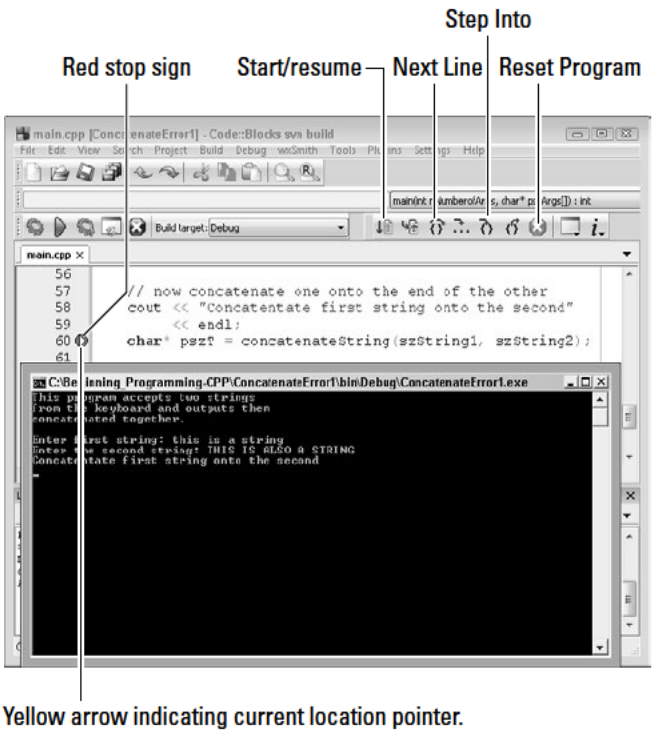


Figure 20-3: The program stops executing, and a small, yellow arrow appears at the next line to be executed when the program encounters a breakpoint.

You will also notice from Figure 20-3 that another toolbar appears. The Debugger toolbar includes the most common debug commands, including most of the commands that I demonstrate in this chapter. (I have added callouts for the commands I will describe later in this chapter.)

Navigating through a program with the debugger

Okay, so I've managed to stop the execution of my program in the middle with the debugger. What can I do now?

I'll start by executing the `concatenateString()` function one statement at a time. I could set a new breakpoint at the first instruction in the function, but setting a new breakpoint on every line is tedious. Fortunately, the Code::Blocks debugger offers a more convenient choice: the *Step Into* command.

On the Debug toolbar, this is the fifth command from the left. However, if you get confused, this menu has Tool Tips — just point at the command in the toolbar and leave the arrow motionless. After a few seconds, the name of the command will pop up. Or you can select Debug➪Step Into from the main menu.

The Step Into command executes a single C++ statement; in this case, the command steps into the function call. Execution stops immediately before the first executable statement in `concatenateString()`. Next, I select Debug➪Debugging Windows➪Watches to display the window shown in Figure 20-4. From this window, I can see that the two arguments to the function, `pszSrc1` and `pszSrc2`, appear to be correct.

The values of `nTargetSize` and `pszTarget` have no meaning at this point since they have yet to be initialized.

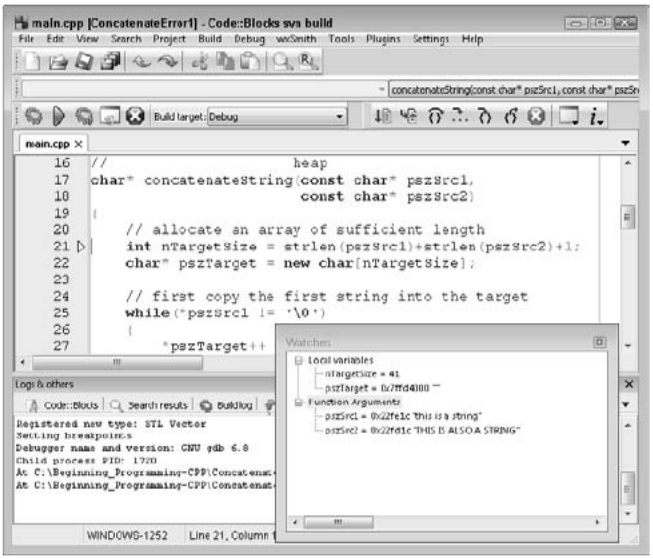


Figure 20-4: The Watches window shows both the arguments to the functions and any locally defined variables.

I could select Step Into again to move forward, but this will step me into the `strlen()` functions. Since these are C++ library routines, I'm willing to accept that these are working fine.

The other option is known as Next Line. Next Line steps to the next line of C++ code in the current function, treating function calls just like any other C++ command.

Together, Step Into and Next Line are known as *single-step* commands. For commands other than function calls, the two commands are equivalent. Many debuggers use the term Step Over rather than Next Line to highlight the distinction from Step Into.

I select Next Line from the Debug toolbar. Notice how the Current location pointer moves from line 21 to line 22, as shown in Figure 20-5. In addition, the `nTargetSize` variable is highlighted red in the Watch window to indicate that its value has changed. The value of `nTargetSize` is now 38, the correct length of the sum of the two strings.

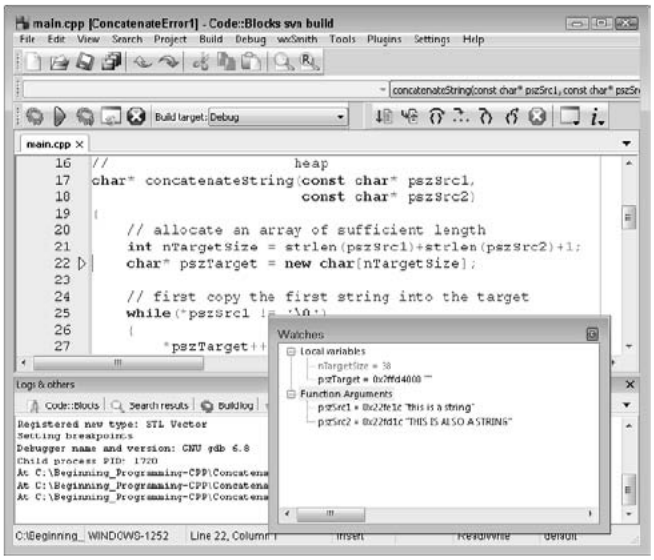


Figure 20-5: Selecting Next Line moves the current location pointer to line 22 and initializes `nTargetSize`.

You need to be absolutely clear about what just happened. All you see is that the screen blinks and the current location pointer moves down one line. What actually happened is that the debugger set a temporary breakpoint at line 22 and then restarted the program at line 21. The program executed the two calls to `strlen()` and then performed the addition, storing the results in `nTargetSize`. You may have seen only the one line of code get executed, but in fact many lines of C++ code were executed within the `strlen()` functions (executed twice, actually).

So far, so good, so I select Next Line a few more times until I enter the `while` loop.

This `while` loop is structured a little differently than what you've seen before. Here, I increment the pointer as part of the assignment itself, rather than in the increment clause of a `for` loop, as follows:

```
while(*pszSrc1 != '\0')
{
     *pszTarget++ = *pszSrc1++; // Line 27
}
```

Line 27 of the program says, "store the value of the `char` pointed at by `pszSrc1` into the `char` location pointed at by `pszTarget` and then increment `pszSrc1` and `pszTarget`."

Figure 20-6 shows the debug display after I execute the loop a few times. Notice after each execution that, since their value is modified, both `pszSrc1` and `pszTarget` are highlighted in the Watches window.

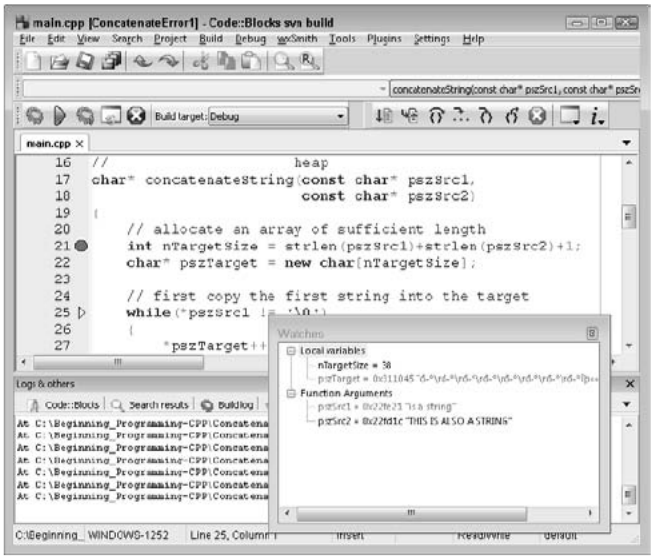


Figure 20-6: The `while` loop increments `pszSrc1` and `pszTarget` on each pass.

Also notice that the string pointed at by `pszSrc1` seems to be shrinking. This is because as `pszSrc1` is incremented, it is effectively moving down the string until eventually it will point to nothing more than the terminating `null`. That's when control will leave the `while` loop and continue on to the next loop.

But wait! The string pointed at by `pszTarget` is not growing. Remember that the intent is to copy the contents of `pszSrc1` into `pszTarget`. What's happening?

After a moment's reflection, the answer is obvious: I'm also changing the value of `pszTarget` and leaving the characters I've copied behind. That's what was wrong with my function in the first place. I need to keep a copy of the original pointer unmodified to return to the caller!

Now that I know the problem (or, at least, a problem — there may be more) I stop the debugger by clicking Stop Debugger on the Debug toolbar. The Console Application dialog box disappears immediately, and the Code::Blocks display returns to that used for editing.

Fixing the (first) bug

To solve the problem that I noted, I only need to save the value returned by `new` and return it rather than the modified `pszTarget` pointer from the function. I include only the modified `concatenateString()` function here (the rest of the program is unchanged — the entire program is included on the enclosed CD-ROM as ConcatenateError2):

```
char* concatenateString(const char* pszSrc1,
                        const char* pszSrc2)
{
    // allocate an array of sufficient length
    int nTargetSize = strlen(pszSrc1)+strlen(pszSrc2)+1;
    char* pszTarget = new char[nTargetSize];
    char* pszT = pszTarget;  // save a pointer to return

    // first copy the first string into the target
    while(*pszSrc1 != '\0')
    {
        *pszTarget++ = *pszSrc1++;
    }

    // now copy the contents of the second string onto
    // the end of the first
    while(*pszSrc2 != '\0')
    {
        *pszTarget++ = *pszSrc2++;
    }

    // return the original pointer to the caller
    return pszT;
}
```

Here, I save the pointer returned by new into both `pszTarget`, which I intend to increment, and `pszT`, which will stay unmodified. The function returns the latter, unmodified pointer to the caller.

I rebuild the application, and then I repeat my earlier steps to single-step through the first loop within `concatenateString()`. Figure 20-7 shows the display after executing the loop seven times.

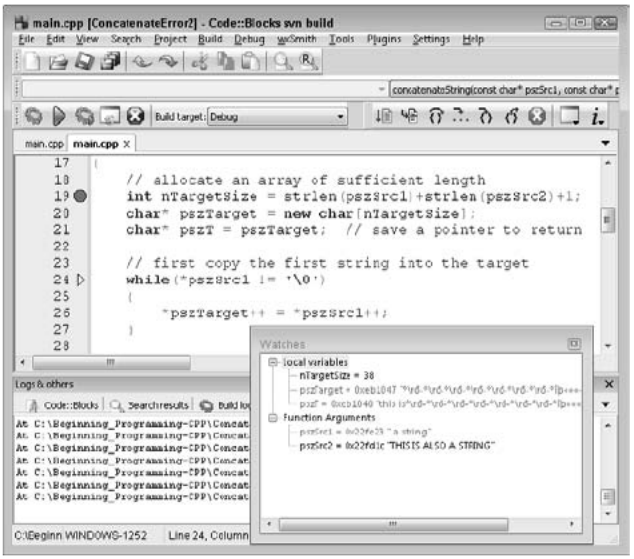


Figure 20-7: The Watches window of the updated `concatenateString()` function shows the string being built in the array pointed at by `pszT`.

Notice how `pszT` points to an array containing the first seven characters of the source string `this is`. Also notice that the value of `pszTarget` is 7 larger than `pszT`.

But also notice all the garbage characters in the `pszT` string that appear after `this is`. Code::Blocks displays extra garbage because the target string has no terminating null. It doesn't need one yet, since I haven't completed constructing it.

Finding and fixing the second bug

The two source strings aren't all that long, so I use the Next Line command to single-step through the entire loop. Figure 20-8 shows the Debug window after executing the second loop for the last time. Here, `pszT` points to the completed target string with both source strings concatenated together. Without a terminating `null`, however, the string still displays garbage after the final character.

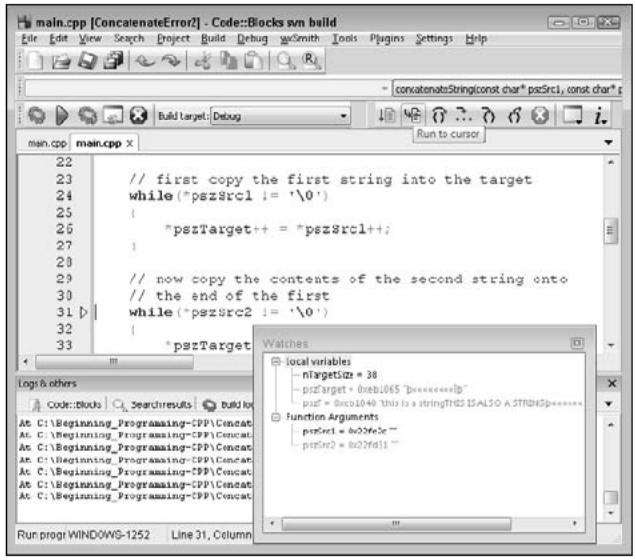


Figure 20-8: The Debug window after executing the second loop for the last time.

Because I'm now done with the function, I select Debug➪Continue from the Code::Blocks menu. This causes the debugger to resume the program where it left off and to continue to the next breakpoint or to the end of the program, whichever comes first.

Sure enough, the displayed concatenated array includes the same garbage that I saw in the debugger:

```
This program accepts two strings
from the keyboard and outputs them
concatenated together.

Enter first string: this is a string
Enter the second string: THIS IS ALSO A STRING
Concatentate first string onto the second
Result: <this is a stringTHIS IS ALSO A STRING       >
Press any key to continue . . .
```

If I didn't include a terminating `null`, then what caused the string returned by `concatenateString()` to terminate at all? Why didn't the string continue on for pages? The short answer is, "Nothing." It could be that C++ had to display many thousands of characters before eventually hitting a character containing a `null`. In practice, this rarely happens, however. Zero is by far the most common value in memory. You generally don't have to look too far before you find a byte containing a zero that terminates the string.

All I need to do to fix this problem is add a terminating `null` after the final `while` loop:

```
char* concatenateString(const char* pszSrc1,
                        const char* pszSrc2)
{
    // allocate an array of sufficient length
    int nTargetSize = strlen(pszSrc1)+strlen(pszSrc2)+1;
    char* pszTarget = new char[nTargetSize];
    char* pszT = pszTarget;  // save a pointer to return

    // first copy the first string into the target
    while(*pszSrc1 != '\0')
    {
        *pszTarget++ = *pszSrc1++;
    }

    // now copy the contents of the second string onto
    // the end of the first
    while(*pszSrc2 != '\0')
    {
        *pszTarget++ = *pszSrc2++;
    }

    // add a terminating NULL
    *pszTarget = '\0';

    // return the unmodified pointer to the caller
    return pszT;
}
```

Executing this version in the debugger creates the display shown in Figure 20-9. Notice that once the terminating `null` has been added, the string pointed at by `pszT` magically "cleans up," losing all the garbage that strings on after the data that I put there.

Let me be clear: Those garbage characters are still there. It's just that the terminating `null` causes C++ to not display them.The output from the program is the predictable string that you've come to love and admire:

```
This program accepts two strings
from the keyboard and outputs them
concatenated together.

Enter first string: this is a string
Enter the second string: THIS IS ALSO A STRING
Concatentate first string onto the second
Result: <this is a stringTHIS IS ALSO A STRING>
Press any key to continue . . .
```

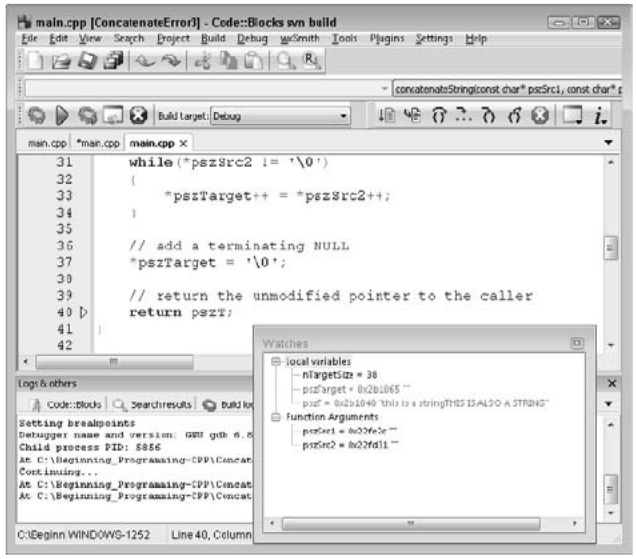
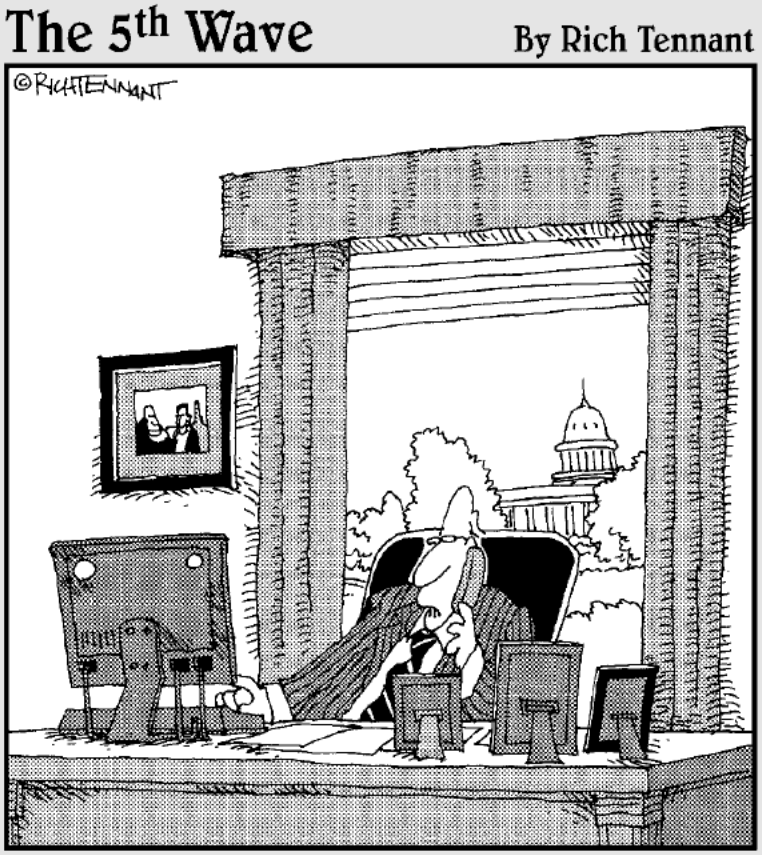


Figure 20-9: Adding the terminating `null` removes all of the garbage characters at the end of the concatenated string.

It's possible to find problems in small programs by adding output statements at key locations. However, the debugger is a much more elegant and powerful tool for finding problems. Single-stepping your way through a program in the debugger gives you a real feel for what the computer is doing with your source code. You develop an understanding for how the computer works that I don't think you can get any other way. The debugger that comes with Code::Blocks is about as easy to use as any that I've seen. I recommend that you use it early and often.

Part V
Object-Oriented Programming



"Yes, I know how to query information from the program, but what if I just want to leak it instead?"

In this part . . .

Parts I through IV describe C++ as just another functional language, not very different from its predecessor, C. This part introduces you to the concepts behind object-oriented programming. These concepts revolutionized the programming world when they became widely adopted in the late 1980s. This is the part that describes what makes C++ the truly powerful language that it is.

Chapter 21

What Is Object-Oriented Programming?

In This Chapter

- Abstracting away the details
- Contrasting the object-oriented approach with the functional approach
- Classifying things

Examples of objects abound in everyday life. Right in front of me is a chair, a table, a computer, and a red Starbucks mug. I have no trouble grouping these objects into taxonomies based upon their properties. For example, the mug is a container, it's also a thermal insulator, so I can use it to hold hot or cold things, and it has mass, so that I can use it as a paperweight or to throw at the dog. Object-oriented programming applies this view of the world to that of programming. To explain what I mean, let me start with a story.

Abstraction and Microwave Ovens

Sometimes when my son and I are watching football, I whip up a batch of nachos. Nothing fancy, mind you — I dump some chips on a plate, throw on refried beans, cheese, and a batch of jalapenos, and nuke the lot in the microwave oven for five minutes. To use the oven, I open the door, place the nachos inside, punch some buttons on the front, and hit start. After a few minutes, the bell rings to tell me they're done. If I do something wrong, the oven beeps at me and doesn't start. Sometimes it displays an error message on the little display.

This doesn't sound very profound, and it isn't really until you consider all the things that I don't do to use my microwave oven:

- I limit myself to the front panel of the microwave. I don't look inside the case. I don't look at the listings of the code that tells the processor unit what to do. I don't study the wiring diagram that's pasted on the inside wall of the case.
- I don't rewrite or change anything inside the microwave to get it to work. The microwave oven that I use to make nachos is the exact same microwave that I used earlier to heat up chili dogs (nothing but health food at my house). And it will be the same microwave I use to heat up my Malt-O-Meal tomorrow (assuming it doesn't break).
- I don't think about what might be going on inside my microwave oven in order to use it. Even if I designed microwaves for a living, I'm not likely to think about how it works when I make nachos before the big game.

These are not profound observations. Humans can think about only so much at any one time. We tend to reduce the number of things that we have to deal with by abstracting away all the little details. This allows us to work at the level of detail appropriate to the problem we're trying to solve.

Note: In object-oriented (OO) terms, this level of detail is known as the *level of abstraction*.

When I'm working on nachos, I view my microwave oven as a black box. I don't concern myself with what's going on inside that box unless, of course, it breaks. Then I might take the top off and see if I can figure out what's wrong with it; then I am working at a different level of abstraction. I still don't take the tops off the chips on the circuit board or try to take apart the individual components. (I'm not *that* crazy.)

As long as the microwave is heating food, I limit myself to the interface that it exposes to the outside world: the keypad and LCD display. It is very important that from this interface there is nothing that I can do that will cause the microwave to:

- Enter an inconsistent state and crash (causing me to have to reboot my microwave)
- Worse, turn my nachos into a blackened, flaming mass
- Worse yet, catch on fire and burn down the house

Functional nachos

Suppose I were to ask my son to write an algorithm for making nachos using the same basic approach used for changing tires in Chapter 1. He would

probably write something like, "Open a can of beans, grate some cheese, cut the jalapenos," and so on. For the part about heating the nachos, he would write something similar to, "Cook in the oven until cheese is melted."

That description is straightforward and complete, but it's not how a functional programmer would code a program to make nachos. Functional programmers live in a world devoid of objects such as microwave ovens. They tend to worry about flowcharts with their myriad functional paths. In a functional solution, the flow of control would pass from my finger through the microwave's front panel and on into the interior of the thing. Soon, the flow would be wiggling through complex logic paths concerned with how long to charge up some capacitor and whether it's time to sound the "come and get it" tone.

In a world like this, it's hard to think in terms of levels of abstraction. There are no objects, no abstractions behind which to hide inherent complexity.

Object-oriented nachos

In an object-oriented approach to making nachos, I would start by identifying the types of objects in the problem: chips, beans, cheese, and an oven. These are the nouns that I have to work with. That done, I would identify the verbs relevant to each object. Next, I would solve the problem using nothing but the nouns and verbs identified before. Finally, then, and only then, I would implement each of these objects in software.

I identified the nouns and verbs relevant to tire changing for you in Chapter 1. You were left with the job of implementing the solution using the nouns and verbs I gave you.

While I am writing object-level code, I am said to be working (and thinking) at the level of abstraction of the basic objects. I need to think about making a useful oven, but I don't have to think about the process of making nachos yet. After all, the designers of my microwave didn't think about the specific problem of my making a snack. Rather, they set about the problem of designing and building a useful microwave oven.

After I have successfully coded and tested the objects I need, I can ratchet up to the next level of abstraction. I can start thinking at the nacho-making level, rather than at the microwave-making level. At this point, I can pretty much translate my son's instructions directly into C++ code.

Classification and Microwave Ovens

Critical to the concept of abstraction is that of classification. If I were to ask my son, "What's a microwave oven?" he would probably say, "It's an oven that. . . ." If I then ask, "What's an oven?" he might reply, "It's a kitchen appliance that. . . ." I could keep asking this question, ratcheting myself up the abstraction ladder until I ended up with, "It's a thing," which is another way of saying, "It's an object."

My son understands that our particular microwave is an instance of the type of things called microwave ovens. In addition, he sees microwave ovens as just a special kind of oven, which is, in turn, a special type of kitchen appliance, and so on.

The technical way of saying this is that our oven is an *instance* of the class *microwave*. The class *microwave* is a *subclass* of the class *oven,* and the class *oven* is a *superclass* of the class *microwave*.

Humans classify. Everything about our world is ordered into taxonomies. We do this to reduce the number of things that we have to remember. Consider, for example, the first time that you saw a hybrid car. The advertisement called it a "revolutionary automobile, unlike any car you've ever seen," but you and I know that this just isn't so. Sure, its propulsion system is different from conventional cars, but it's still a car and as such does the same things that all cars do: convey you and your kin from one place to another. It has a steering wheel, seats, a motor, brakes, and so on. I bet I could even drive one without help.

I don't have to clutter my limited storage with all the things that a hybrid card has in common with other cars. All I have to remember is that "a hybrid car is a car that. . . ." and tack on those few things that are unique to a hybrid. Cars are a subclass of wheeled vehicles, of which there are other members, such as trucks and pickups. Maybe wheeled vehicles are a subclass of vehicles, which includes boats and planes. And on and on and on.

Why Build Objects This Way?

It may seem easier to design and build a microwave oven specifically for this one problem, rather than to build a separate, more generic oven object. Suppose, for example, that I were to build a microwave to cook nachos and nachos only. I wouldn't need to put a front panel on it, other than a START button. I always cook nachos the same amount of time. I could dispense with all that DEFROST and TEMP COOK nonsense. The microwave could be tiny. It would need to hold only one fat, little plate. The cubic feet of space would be completely wasted on nachos.

For that matter, suppose I just dispense with the concept of "microwave oven" altogether. All I really need is the guts of the oven. Then in the recipe, I can put the instructions to make it work: "Put nachos in the box. Connect the red wire to the black wire. Notice a slight hum. Don't stand too close if you intend to have children." Stuff like that.

Nevertheless, the functional approach does have some problems:

- **Too complex.** You don't want the details of oven building mixed in with the details of nacho building. If you can't define the objects and pull them out of the morass of details to deal with separately, you must deal with all the complexities of the problem at the same time.
- **Not flexible.** If you need to replace the microwave oven with some other type of oven, you should be able to do so as long as the interface to the new oven is about the same as the old one. Without a simple and clearly delineated interface, it becomes impossible to cleanly remove an object type and replace it with another.
- **Not reusable.** Ovens are used to make many different dishes. You don't want to create a new oven each time you encounter a new recipe. Having solved a problem once, it would be nice to reuse the solution in future programs.

It does cost more to write a generic object. It would be cheaper to build a microwave made specifically for nachos. You could dispense with expensive timers, buttons, and the like that aren't needed to make nachos. After you have used a generic object in more than one application, however, the costs of a slightly more expensive class more than outweigh the repeated costs of building cheaper, less flexible classes for every new application.

Self-Contained Classes

Now, it's time to reflect on what you've learned. In an object-oriented approach to programming:

- The programmer identifies the classes necessary to solve the problem. (I knew right off that I was going to need an oven to make decent nachos.)
- The programmer creates self-contained classes that fit the requirements of the problem and doesn't worry about the details of the overall application.
- The programmer writes the application using the classes just created without thinking about how they work internally.

An integral part of this programming model is that each class is responsible for itself. A class should be in a defined state at all times. It should not be possible to crash the program by calling a class with illegal data or with an illegal sequence of correct data.

Many of the features of C++ that are shown in subsequent chapters deal with giving the class the capability to protect itself from errant programs just waiting to trip it up.

Chapter 22

Structured Play: Making Classes Do Things

In This Chapter

- Adding member functions to a class
- Defining the member function
- Invoking the member function
- Accessing one member from another member
- Overloading member functions

Classes were introduced to the C language as a convenient way to group unalike but related data elements — for example, the Social Security number and name of the same person. That's the way I introduce them in Chapter 19. C++ expanded the concept of classes to give them the ability to mimic objects in the real world. That's the essence of the difference between C and C++.

In the previous chapter, I review at a high level the concept of object-oriented programming. In this chapter, I make it more concrete by examining the active features of a class that allow them to better mimic the object-oriented world we live in.

Activating Our Objects

C++ uses classes to simulate real-world objects. However, the classes in Chapter 19 are lacking in that regard because classes do things. (The classes in Chapter 19 don't have any verbs associated with them — they don't do anything.) Consider for example, a savings account. It is necessary for a `Savings` class to save the owner's name, probably her Social Security number, certainly her account number and balance. But this isn't sufficient.

Objects in the real world do things. Ovens cook. Savings accounts accumulate interest. CDs charge a substantial penalty for early withdrawal. Stuff like that.

Consider the problem of handling deposits in a `Savings` account class. Functional programs do things via functions. Thus, a function program might create a separate function that takes as its argument a pointer to a `Savings` account object that it wants to update followed by the amount to deposit.

Never mind for now exactly how to pass a pointer to a `Savings` account object. You'll see more about that in the next chapter.

But that's not the way that savings accounts work in the real world. When I drive up to the bank window and tell them I want to make a deposit to my savings account, the teller doesn't hand me a ledger into which I note the deposit and write the new balance. She doesn't do it herself either. Instead, she types in the amount of the deposit at some terminal and then places that amount in the till. The machine spits out a deposit slip with the new balance on it that she hands me, and it's all done. Neither of us touches the bank's books directly.

This may seem like a silly exercise but consider why the bank doesn't do things "the functional way." Ignore for a minute the temptation I might have to add a few extra zeros to the end of my deposit before adding it up. The bank doesn't do things this way for the same reason that I don't energize my microwave oven by connecting and disconnecting wires inside the box — the bank wants to maintain tight controls on what happens to its balances.

If something screws up and my savings account balance gets incremented by a million dollars or so ("My gosh, how did that happen?"), the bank has a vested interest in being able to figure out exactly what happened and make sure that it doesn't happen again. A simple arithmetic error made by me or the teller is not sufficient justification for a mistake like that. The bank has a legal and fiduciary responsibility for maintaining its accounts in good order. It can't do that if every person who sallies up to the teller window has direct access to the books.

This care extends to programmers as well. You can rest easy at night knowing that not every programmer gets direct access to the bank balances either. Only the most trusted of programmers get to write the code that increments and decrements bank balances.

I use the term "trusted" here in two senses. First, the bank trusts these individuals not to intentionally steal. However, the bank also trusts these programmers to take all of the necessary process steps to fully vet and test the `deposit()` and `withdraw()` functions to make sure that they are bug-free and implement the bank's rules accurately.

To make the `Savings` class mimic a real-world savings account, it needs active properties of its own, like `deposit()` and `withdrawal()` (and `chargePenalty()` for who knows why, in my case). Only in this way can a `Savings` class be held responsible for its state.

Creating a Member Function

A function that is part of a class definition is known as a *member function*. The data within the class is known as *data members*. Member functions are the verbs of the class, whereas data members are the nouns.

Member functions are also known as *methods* because that's what they were called in the original object-oriented language, Smalltalk. The term methods had meaning to Smalltalk, but it has no special meaning in C++, except that it's easier to say and sounds more impressive in a conversation. I'll try not to bore you with this trivia, but you will hear the term method bandied about at object-oriented parties, so you might as well get used to it. I'll try to stick with the term member functions, but even I slip into technical jargon from time to time.

Note: Functions that you have seen so far that are not members of a class don't have a special name. I refer to them as *non-member functions* when I need to differentiate them from their member cousins.

There are three aspects to adding a member function to a class: defining the function, naming the function, and calling the function. Sounds pretty obvious when you say it that way.

Defining a member function

The following class demonstrates how to define two key member functions, `deposit()` and `withdraw()`, in a class `Savings` account:

```
// Savings - a simple savings account class
class Savings
{
  public:
    int    nAccountNumber;
    double dBalance;

    // deposit - deposit an amount to the balance;
    //           deposits must be positive number; return
    //           the resulting balance or zero on error
    double deposit(double dAmount)
```

```
    {
        // no negative deposits - that's a withdrawal
        if (dAmount < 0)
        {
            return 0.0;
        }

        // okay - add to the balance and return the total
        dBalance += dAmount;
        return dBalance;
    }

    // withdraw - execute a withdrawal if sufficient funds
    //            are available
    double withdraw(double dAmount)
    {
        if (dBalance < dAmount)
        {
            return 0.0;
        }

        dBalance -= dAmount;
        return dBalance;
    }
};
```

A real savings account class would have a lot of other information like the customer's name. Adding that extra stuff doesn't help explain the concepts, however, so I've left it off to keep the listings as short as possible.

You can see that the definition of the `deposit()` and `withdraw()` member functions look just like those of any other function except that they appear within the definition of the class itself. There are some other subtle differences that I address later in this chapter.

It is possible to define a member function outside of the class, as you will see a little later in this chapter.

Naming class members

A member function is a lot like a member of a family. The full name of the deposit function is `Savings::deposit(double)` just like my name is Stephen Davis. My mother doesn't call me that unless I'm in trouble. Normally, members of my family just call me by my first name, Stephen.

Similarly, from within the `Savings` class, the deposit function is known simply as `deposit(double)`.

The class name at the beginning indicates that this is a reference to the `deposit()` function that is a member of the class `Savings`. The `::` is simply a separator between the class name and the member name. The name of the class is part of the extended name of the member function just like Stephen Davis is my extended name. (See Chapter 11 if you don't remember about extended names.)

Classes are normally named using nouns that describe concepts like `Savings` or `SavingsAccount`. Member functions are normally named with the associated verbs like `deposit()` or `withdraw()`. Other than that, member functions follow the same naming convention as other functions. Data members are normally named using nouns that describe specific properties like `szName` or `nSocialSecurityNumber`.

You can define a different `deposit()` function that has nothing to do with the `Savings` class — there are Stephens out there who have nothing to do with my family. (I mean this literally: I know several Stephens who want *nothing* to do with my family.) For example, `Checking::deposit(double)` or `River::deposit()` are easily distinguishable from `Savings::deposit(double)`.

A non-member function can appear with a null class name. For example, if there were a deposit function that was not a member of any class, its name would be `::deposit()` or simply `deposit()`.

Calling a member function

Before I show you how to invoke a member function, let me quickly refresh you on how to access a data member of an object. Given the earlier definition of the `Savings` class, you could write the following:

```
void fn()
{
    Savings s;

    s.nAccountNumber = 0;
    s.dBalance = 0.0;
}
```

The function `fn()` creates a `Savings` object `s` and then zeros the data members `nAccountNumber` and `dBalance` of that object.

Notice that the following does not make sense:

```
void fn()
{
    Savings s1, s2;

    nAccountNumber = 0;  // doesn't work
    dBalance = 0.0;
}
```

Which `nAccountNumber` and `dBalance` are you talking about? The account number and balance of `s1` or `s2`. Or some other object entirely? A reference to a data member makes sense only in the context of an object.

Invoking a member function is the same. You must first create an object and then you can invoke the member function on that object:

```
void fn()
{
    // create and initialize an object s
    Savings s = {0, 0.0};

    // now make a deposit of $100
    s.deposit(100.0);

    // or a withdrawal
    s.withdraw(50.0);
}
```

The syntax for calling a member function looks like a cross between the syntax for accessing a data member and that used for calling functions. The right side of the dot looks like a conventional function call, but an object appears on the left side of the dot.

This syntax makes sense when you think about it. In the call `s.deposit()`, `s` is the savings object to which the `deposit()` is to be made. You can't make a deposit without knowing to which account. Calling a member function without an object makes no more sense than referencing a data member without an object.

Accessing other members from within a member function

I can see it now: You repeat to yourself, "You can't access a member without reference to an object. You can't access a member without reference to an object. You can't. . . ." And then, wham, it hits you. `Savings::deposit()` appears to do exactly that:

```
double deposit(double dAmount)
{
    // no negative deposits - that's a withdrawal
    if (dAmount < 0)
    {
        return 0.0;
    }

    // okay - add to the balance and return the total
    dBalance += dAmount;
    return dBalance;
}
```

The `Savings::deposit()` function references `dBalance` without an explicit reference to any object. It's like that TV show: "How Do They Do It?"

So, okay, which is it? Can you or can you not reference a member without an object? Believe me, the answer is no. When you reference one member from within another member of the same class without explicitly referring to an object, the reference is implicitly against the "current object."

What is the current object? Go back and look at the example in greater detail. I am pulling out just the key elements of the example here for brevity's sake:

```
class Savings
{
  public:
    int    nAccountNumber;
    double dBalance;

    double deposit(double dAmount)
    {
        dBalance += dAmount;
        return dBalance;
    }
};

void fn()
{
    // create and initialize two objects
    Savings s1 = {0, 0.0};
    Savings s2 = {1, 0.0};

    // now make a deposit of $100 to one account
    s1.deposit(100.0);

    // and then the other
    s2.deposit(50.0);
}
```

When `deposit()` is invoked with `s1`, the unqualified reference to `dBalance` refers to `s1.dBalance`. At that moment in time, `s1` is the "current object." During the call to `s2.deposit(50.0)`, `s2` becomes the current object. During this call, the unqualified reference to `dBalance` refers to `s2.dBalance`.

The "current object" has a name. It's called `this` as in "this object." Clever, no? Its type is "pointer to an object of the current class." I say more about this in Chapter 23 when I talk about pointers to objects.

Keeping a Member Function after Class

One of the things that I don't like about C++ is that it provides multiple ways of doing most things. Keeping with that penchant for flexibility, C++ allows you to define member functions outside the class as long as they are declared within the class.

The following is an example of the `withdraw()` function written outside the class declaration (once again, I've left out the error checking to make the example as short as possible):

```
// this part normally goes in the Savings.h include file
class Savings
{
  public:
    int    nAccountNumber;
    double dBalance;

    double deposit(double dAmount);
};

// this part appears in a separate Savings.cpp file
double Savings::deposit(double dAmount)
{
    dBalance += dAmount;
    return dBalance;
}
```

Now the definition of `Savings` contains nothing more than the prototype declaration of the member function `deposit()`. The actual definition of the function appears later. Notice, however, that when it does appear, it appears with its full extended name, including the class name — there is no default class name outside of the class definition.

This form is ideal for larger member functions. In these cases, the number of lines of code within the member functions can get so large that it obscures the definition of the class itself. In addition, this form is useful when defining classes in their own C++ source modules. The definition of the class can appear in an include file, `Savings.h`, while the definition of the function appears in a separately compiled `Savings.cpp`.

Overloading Member Functions

You can overload member functions just like you overload any other functions. Remember, however, that the class name is part of the extended name. That means that the following is completely legal:

```
class Student
{
  public:
    double grade();  // return Student's grade
    double grade(double dNewGPA); // set Student's grade
};

class Hill
{
  public:
    double grade(double dSlope); // set the slope
};
    void grade(double);

void fn()
{
    Student s;
    Hill h;

    // set the student's grade
    s.grade(3.0);

    // now query the grade
    double dGPA = s.grade();

    // now grade a hill to 3 degrees slope
    h.grade(3.0);

    // call the non-member function
    grade(3.0);
}
```

When calling a member function, the type of the object is just as important as the number and type of the arguments. The first call to `grade()` invokes the function `Student::grade(double)` to set the student's grade point average. The second call is to `Student::grade()`, which returns the student's grade point average without changing it.

The third call is to a completely unrelated function, `Hill::grade(double)`, that sets the slope on the side of the hill. And the final call is to the non-member function `::grade(double)`.

Chapter 23

Pointers to Objects

In This Chapter

- Adding member functions to a class
- Defining the member function
- Invoking the member function
- Accessing one member from another member
- Overloading member functions

Chapters 17 and 18 focus on various aspects of the care and feeding of pointers. Surely, you think, nothing more can be said on the subject. But I hadn't introduced the concept classes before those chapters. In this chapter, I describe the intersection of pointer variables and object-oriented programming. This chapter deals with the concept of pointers to class objects. I'll describe how to create one, how to use it, and how to delete it once you're finished with it.

Pointers to Objects

A pointer to a programmer-defined type such as a class works essentially the same as a pointer to an intrinsic type:

```
int nInt;
int* pInt = &nInt;

class Savings
{
  public:
    int nAccountNumber;
    double dBalance;
};
Savings s;
Savings* pS = &s;
```

The first pair of declarations defines an integer, `nInt`, and a pointer to an integer, `pInt`. The pointer `pInt` is initialized to point to the integer `nInt`.

Similarly, the second pair of declarations creates a `Savings` object `s`. It then declares a pointer to a `Savings` object, `pS`, and initializes it to the address of `s`.

The type of `pS` is "pointer to `Savings`" which is written `Savings*`.

I feel like the late Billy Mays when I say, "But wait! There's more!" The similarities continue. The following statement assigns the value `1` to the `int` pointed at by `pInt`:

```
*pInt = 1;
```

Similarly, the following assigns values to the account number and balance of the `Savings` object pointed at by `pS`.

```
(*pS).nAccountNumber = 1234;
(*pS).dBalance = 0.0;
```

The parentheses are required because the precedence of `.` is higher than `*`. Without the parentheses, `*pS.nAccountNumber = 1234` would be interpreted as `*(pS.nAccountNumber) = 1234`, which means "store 1234 at the location pointed at by `pS.nAccountNumber`." This generates a compiler error because `nAccountNumber` isn't a pointer (nor is `pS` a `Savings`).

Arrow syntax

The only thing that I can figure is that the authors of the C language couldn't type very well. They wasted no efforts in finding shorthand ways of saying things. Here is another case where they made up a shorthand to save keystrokes, inventing a new operator `->` to stand for `*()`:

```
pS->dBalance = 0.0; // same as (*pS).dBalance = 0.0
```

Even though the two are equivalent, the arrow operator is used almost exclusively because it's easier to read (and type). Don't lose sight of the fact, however, that the two forms are completely equivalent.

Calling all member functions

The syntax for invoking a member function with a pointer is similar to accessing a data member:

```
class Savings
{
  public:
    int nAccountNumber;
    double dBalance;

    double withdraw(double dAmount);
    double deposit(double dAmount);
};

void fn()
{
    Savings s = {1234, 0.0};
    Savings* pS = &s;

    // deposit money into the account pointed at by pS
    pS->deposit(100.0);
}
```

The last statement in this snippet says "invoke the `deposit()` member function on the object pointed at by `pS`."

Passing Objects to Functions

Passing pointers to functions is just one of the many ways to entertain yourself with pointers.

Calling a function with an object value

As you know, C++ passes arguments to functions by value by default. If you don't know that, refer to Chapter 11. Complex, user-defined objects are passed by value as well:

```
class Savings
{
  public:
    int nAccountNumber;
    double dBalance;

    double withdraw(double dAmount);
    double deposit(double dAmount);
```

```
};

void someOtherFunction(Savings s)
{
    s.deposit(100.0);
}

void someFunction()
{
    Savings s = {1234, 0.0};

    someOtherFunction(s);
}
```

Here the function `someFunction()` creates and initializes a `Savings` object `s`. It then passes a copy of that object to `someOtherFunction()`. The fact that it's a copy is important for two reasons:

- Making copies of large objects can be very inefficient, causing your program to run slower.
- Changes made to copies don't have any effect on the original object in the calling function.

In this case, the second problem is much worse than the former. I can stand a little bit of inefficiency since a `Savings` object isn't very big anyway, but the deposit made in `someOtherFunction()` got booked against a copy of the original account. My `Savings` account back in `someFunction()` still has a balance of zero. This is shown graphically in Figure 23-1.

Calling a function with an object pointer

The programmer can pass the address of an object rather than the object itself as demonstrated in the following example:

```
class Savings
{
  public:
    int nAccountNumber;
    double dBalance;

    double withdraw(double dAmount);
    double deposit(double dAmount);
```

```
};

void someOtherFunction(Savings* pS)
{
    pS->deposit(100.0);
}

void someFunction()
{
    Savings s = {1234, 0.0};

    someOtherFunction(&s);
}
```

The type of the argument to `someOtherFunction()` is "pointer to `Savings`." This is reflected in the way that `someFunction()` performs the call, passing not the object `s` but the address of the object, `&s`. This is shown graphically in Figure 23-2.

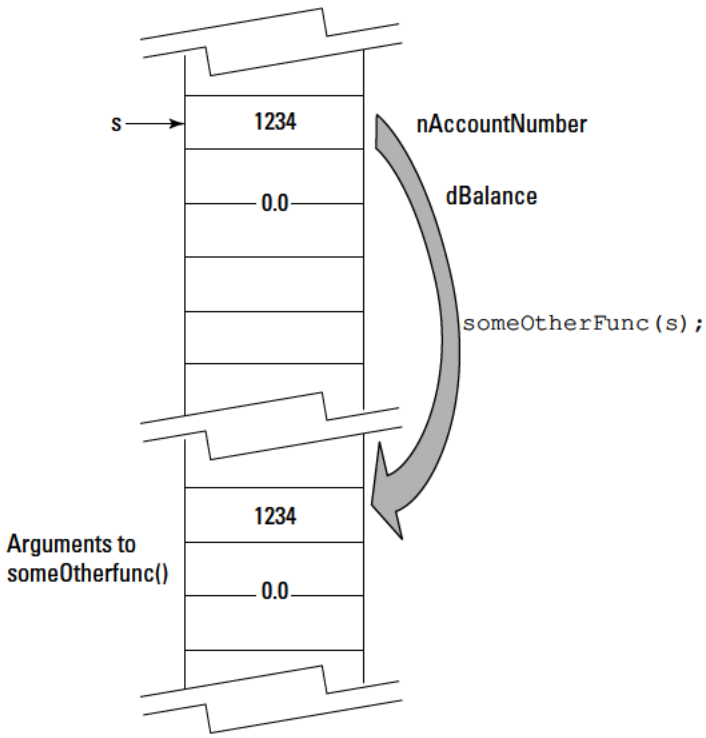


Figure 23-1: By default, C++ passes a copy of the `Student` object `s` to `someOtherFunc-tion()`.

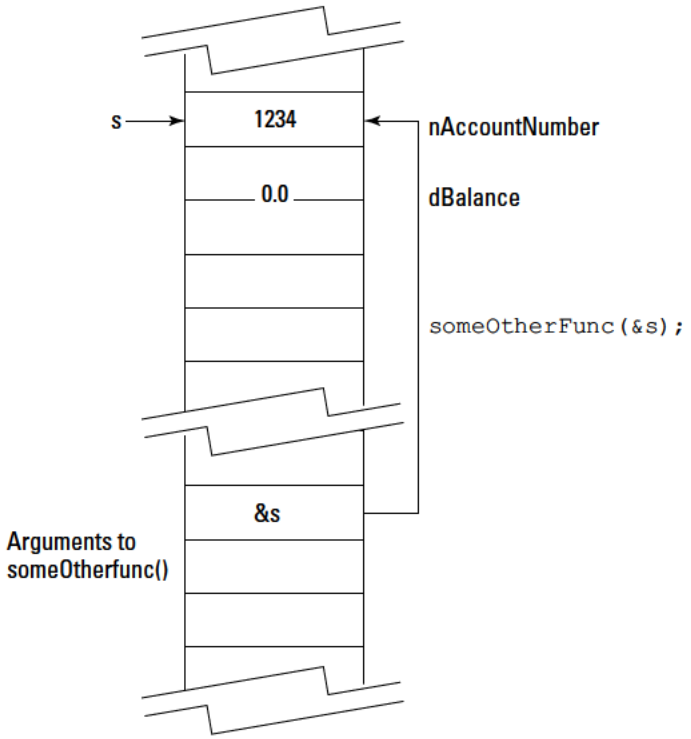


Figure 23-2: By passing the address of the original `Savings` object, the programmer can avoid creating a copy of the original object.

This addresses both of the problems with passing a copy:

- No matter how large and complicated the object might be, the call passes only a single address.
- Changes made in `someOtherFunction()` are permanent because they refer to the original object and not a copy.

Looking at an example

The following program demonstrates the difference between passing an object by value versus passing the address of an object:

```
//
//  PassObjects - this program demonstrates passing an
//                object by value versus passing the
//                address of the object
//
#include <cstdio>
```

```
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;


// Savings - a simple savings account class
class Savings
{
  public:
    int    nAccountNumber;
    double dBalance;

    // deposit - deposit an amount to the balance;
    //           deposits must be positive number; return
    //           the resulting balance or zero on error
    double deposit(double dAmount)
    {
        // no negative deposits - that's a withdrawal
        if (dAmount < 0)
        {
            return 0.0;
        }

        // okay - add to the balance and return the total
        dBalance += dAmount;
        return dBalance;
    }

    // withdraw - execute a withdrawal if sufficient funds
    //            are available
    double withdraw(double dAmount)
    {
        if (dBalance < dAmount)
        {
            return 0.0;
        }

        dBalance -= dAmount;
        return dBalance;
    }

    // balance - return the balance of the current object
    double balance()
    {
        return dBalance;
    }
};

// someFunction(Savings) - accept object by value
```

```
void someFunction(Savings s)
{
    cout << "In someFunction(Savings)" << endl;

    cout << "Depositing $100" << endl;
    s.deposit(100.0);

    cout << "Balance in someFunction(Savings) is "
         << s.balance() << endl;
}

// someFunction(Savings*) - accept address of object
void someFunction(Savings* pS)
{
    cout << "In someFunction(Savings*)" << endl;

    cout << "Depositing $100" << endl;
    pS->deposit(100.0);

    cout << "Balance in someFunction(Savings) is "
         << pS->balance() << endl;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    Savings s = {0, 0.0};

    // first, pass by value
    someFunction(s);
    cout << "Balance back in main() is "
         << s.balance() << endl;

    // now pass the address
    someFunction(&s);
    cout << "Balance back in main() is "
         << s.balance() << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This program starts by defining a conventional `Savings` class with `deposit()`, `withdrawal()`, and `balance()` member functions (the last one just returns the current balance).

The program then defines two overloaded functions `someFunction()`, one of which accepts as its argument an object of type `Savings` and the second a pointer to an object of type `Savings` (written `Savings*`). Both functions do the same things, first outputting a "Here I am" message and then depositing $100 to the account.

Passing by reference

In an attempt to make things simpler, C++ added a level of complexity by allowing the programmer to declare a function that accepts its argument by reference as follows:

```
// pass by reference
void someFunction(Savings& refS)
{
    refS.deposit(100.0);  // this deposits back into the original
                          // object in fn() even though it looks
                          // copy semantics
}
void fn()
{
    Savings s;
    someFunction(s);      // this passes a reference, not a copy
}
```

This causes C++ to pass the address of `s` to the function `someFunction(Savings)`. Within the function, C++ automatically dereferences the address for you. The effect is exactly the same as if you had passed the address yourself except that C++ handles the pointer grammar. You might think that this makes things simpler. (I suspect the authors of C++ thought it would.) In practice, however, it makes things more complicated since it becomes difficult to tell a value from a reference.

I mention pass by reference not to encourage its use, but because you are likely to see others that aren't as comfortable as you with pointer manipulation using it. I would encourage you to avoid use of references until you are really comfortable with pointers.

The `main()` program creates a `Savings` object `s`, which it first passes to `someFunction(Savings)`. It then passes the address of the `s` object to `someFunction(Savings*)`.

The output from this program appears as follows:

```
In someFunction(Savings)
Depositing $100
Balance in someFunction(Savings) is 100
Balance back in main() is 0
In someFunction(Savings*)
Depositing $100
Balance in someFunction(Savings) is 100
Balance back in main() is 100
Press any key to continue . . .
```

Notice how both functions deposit $100 into a `Savings` account object. However, since `someFunction(Savings)` makes the deposit into a copy, the original `s` object back in `main()` is left unchanged as demonstrated by the zero balance.

By passing the address of `s` to `someFunction(Savings*)`, the program allows that function to modify the original object so the value "stays modified" in `main()` as demonstrated by the fact that the balance is $100 after control returns.

Allocating Objects off the Heap

You can allocate objects off of the heap using the `new` keyword as shown in the following example:

```
Savings* newSavings(int nAccountNum)
{
    Savings* pS = new Savings;
    pS->nAccountNumber = nAccountNum;
    pS->dBalance = 0.0;
    return pS;
}
```

The function allocates a new object of class `Savings` and then initializes it with the account number passed as an argument and a zero balance.

This is useful when you don't know how many objects you are going to need, like in the case of dynamically sized character arrays in Chapter 18. Then, I first counted how many characters I needed room for and then allocated an array of the appropriate size off of the heap.

In this case, I can determine how many `Savings` accounts I need in memory at one time and allocate them dynamically off of the heap.

Of course, there is the little matter of how do you store an unknown quantity of objects. C++ provides several variable-sized data structures in addition to the fixed-sized array as part of the Standard Template Library. A general discussion of the STL is beyond the scope of a beginner book

You must return every object that you allocate off of the heap by passing the unmodified address of that object to the keyword `delete`. Otherwise, your program will slowly run out of memory and die a horrible death.

What is `this` anyway?

In Chapter 22, I mention that an otherwise unqualified reference to a member made from within a member function always refers to the "current object." I even mention that the current object has a name: `this`. You can reference `this` explicitly. I could have written the `Savings` class as follows:

```
class Savings
{
  public:
    int nAccountNumber;
    double dBalance;

    double withdraw(double dAmount)
    {
        this->dBalance -= dAmount;
        return this->dBalance;
    }
    double deposit(double dAmount)
    {
        this->dBalance += dAmount;
        return this->dBalance;
    }
    double balance()
    {
        return this->dBalance;
    }
}
```

In fact, even without explicitly referring to it, you use `this` all the time. If you don't specify an object within a member function, C++ assumes a reference to `this`. Thus, the preceding is what C++ actually "sees" even if you don't mention `this`.

Chapter 24

Do Not Disturb: Protected Members

In This Chapter

- Protecting members of a class
- Why do that?
- Declaring friends of the class

My goal with this part of the book, starting with Chapter 21, has been to model real-world objects in C++ using the class structure. In Chapter 22, I introduce the concept of member functions in order to assign classes' active properties. Returning to the microwave oven example in Chapter 21, assigning active properties allows me to give my Oven class properties like `cook()` and `defrost()`.

However, that's only part of the story. I still haven't put a box around the insides of my classes. I can't very well hold someone responsible if the microwave catches on fire as long as the insides are exposed to anyone who wants to mess with them.

This chapter "puts a box" around the classes by declaring certain members off limits to user functions.

Protecting Members

Members of a class can be flagged as inaccessible from outside the class with the keyword `protected`. This is in direct opposition to the `public` keyword, which designates those members that are accessible to all functions. The public members of a class form the interface to the class (think of the keypad on the front of the microwave oven) while the protected members form the inner workings.

There is a third category called `private`. The only difference between private and protected members is the way they react to inheritance, which I don't present until Chapter 28.

Why you need protected members

Declaring a member `protected` allows a class to put a protective box around the class. This makes the class responsible for its own internal state. If something in the class gets screwed up, the class, rather the author of the class, has nowhere to look except herself. It's not fair, however, to ask the programmer to take responsibility for the state of the class if any ol' function can reach in and muck with it.

In addition, limiting the interface to a class makes the class easier to learn for programmers that use that interface in their programs. In general, I don't really care how my microwave works inside as long as I know how to use the controls. In a similar fashion, I don't generally worry about the inner workings of library classes as long as I understand the arguments to the public member functions.

Finally, limiting the class interface to just some choice public functions reduces the level of coupling between the class and the application code.

Note: *Coupling* refers to how much knowledge the application has of how the class works internally and vice versa. A tightly coupled class has intimate knowledge of the surrounding application and uses that knowledge. A loosely coupled class works only through a simple, generic public interface. A loosely coupled class knows little about its surroundings and hides most of its own internal details as well. Loosely coupled classes are easier to test and debug and easier to replace when the application changes.

I know what you functional types out there are saying: "You don't need some fancy feature to do all that. Just make a rule that says certain members are publicly accessible and others are not." This is true in theory, and I've even been on projects that employed such rules, but in practice it doesn't work. People start out with good intentions, but as long as the language doesn't at least discourage direct access of protected members, these good intentions get crushed under the pressure to get the product out the door.

Making members protected

Adding the keyword `public:` to a class makes subsequent members publicly accessible. Adding the keyword `protected:` makes subsequent members

protected, which means they are accessible only to other members of the same class or functions that are specifically declared *friends* (more on that later in this chapter). They act as toggles — one overrides the other. You can switch back and forth between protected and public as often as you like.

Take, for example, a class `Student` that describes the salient features of a college student. This class has the following public member functions:

- `addGrade(int nHours, double dGrade)` — add a grade to the student.
- `grade()` — return the student's grade point average (GPA).
- `hours()` — return the number of semester hours toward graduation.

The remaining members of `Student` should be declared protected to keep prying expressions out of his business.

The following SimpleStudent program defines such a `Student` class and includes a simple `main()` that exercises the functions:

```
//
//  SimpleStudent - this program demonstrates how the
//                  protected keyword is used to protect
//                  key internal members
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  protected:
    double dGrade;          // the student's GPA
    int    nSemesterHours;

  public:
    // init() - initialize the student to a legal state
    void init()
    {
        dGrade = 0.0;
        nSemesterHours = 0;
    }

    // getGrade() - return the current grade
    double getGrade()
    {
        return dGrade;
```

```
    }

    // getHours() - get the class hours towards graduation
    int getHours()
    {
        return nSemesterHours;
    }

    // addGrade - add a grade to the GPA and total hours
    double addGrade(double dNewGrade, int nHours)
    {
        double dWtdHrs = dGrade * nSemesterHours;
        dWtdHrs += dNewGrade * nHours;
        nSemesterHours += nHours;
        dGrade = dWtdHrs / nSemesterHours;
        return dGrade;
    }
};

int main(int nNumberofArgs, char* pszArgs[])
{
    // create a student and initialize it
    Student s;
    s.init();

    // add the grades for three classes
    s.addGrade(3.0, 3);  // a B
    s.addGrade(4.0, 3);  // an A
    s.addGrade(2.0, 3);  // a C (average should be a B)

    // now print the results
    cout << "Total # hours = " << s.getHours()
         << ", GPA = " << s.getGrade()
         << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This `Student` protects its members `dGrade` and `nSemesterHours`. Outside functions can't surreptitiously set their own GPA high by slipping in the following:

```
void MyFunction(Student* pS)
{
    // set my grade to A+
    pS->dGrade = 3.9;   // generates a compiler error
}
```

This assignment generates a compiler error.

You can start with either the protected or public members; it doesn't matter. In fact, you can switch back and forth as often as you like.

Any function can read a student's GPA through the function `getGrade()`. This is known as an *access* function. However, though external functions can read a value, they cannot change the value via this access function.

An access function is also known as a *getter function* (as in "get the value"). A function that sets the value is also known as a *setter function*.

The `main()` function in this program creates a `Student` object `s`. It cannot initialize `s` to some legal state since the data members are protected. Fortunately, the `Student` class has provided an `init()` function for `main()` to call that initializes the data members to their proper starting state.

After initializing `s`, `main()` calls `addGrade()` to add three different courses and prints out the results using the access member functions. The results appear as follows:

```
Total # hours = 9, GPA = 3
Press any key to continue . . .
```

So what?

So what's the big deal? "Okay," you say, "I see the point about not letting other functions set the GPA to some arbitrary value, but is that it?" No. A finer point lies behind this loose coupling. I chose to implement the algorithms for calculating the GPA as simply as I possibly could. With no more than five minutes' thought, I can imagine at least three different ways I could have chosen to store the grades and semester hours internally, each with their own advantages and disadvantages.

For example, I could save off each grade along with the number of semester hours in an internal array. This would allow the student to review the grades that are going into his GPA.

The point is that the application programmer shouldn't care. As long as the member functions `getGrade()` and `getHours()` calculate the GPA and total number of semester hours accurately, no application is going to care.

Now suppose the school changes the rules for how to calculate the GPA. Suppose, for example, that it declares certain classes to be Pass/Fail, meaning that you get credit toward graduation but the grade in the class doesn't go into the GPA calculation. This may require a total rewrite of the `Student` class. That, in turn, would require modification to any functions that rely

upon the way that the information is stored internally — that is, any functions that have access to the protected members. However, functions that limit themselves to the public members are unaffected by the change.

That is the true advantage of loose coupling: tolerance to change.

Who Needs Friends Anyway?

Occasionally, you need to give a non-member function access to the protected members of a class. You can do this by declaring the function to be a *friend*. Declaring a function to be a friend means you don't have to expose the protected member to everyone by declaring it public.

It's like giving your neighbor a key to check on your house during your vacation. Giving non–family members keys to the house is not normally a good idea, but it beats the alternative of leaving the house unlocked.

The friend declaration appears in the class that contains the protected member. The friend declaration consists of the keyword `friend` followed by a prototype declaration. In the following example, the `initialize()` function is declared as a non-member. However, `initialize()` clearly needs access to all the data members of the class, protected or not:

```
class Student
{
    friend void initialize(Student*);
  protected:
    double dGrade;          // the student's GPA
    int    nSemesterHours;

  public:
    double grade();
    int hours();
    double addGrade(double dNewGrade, int nHours);
};

void initialize(Student* pS)
{
    pS->dGrade = 0.0;
    pS->nSemesterHours = 0;
}
```

A single function can be declared to be a friend of two different classes at the same time. Although this may seem convenient, it tends to bind the two classes together. However, sometimes the classes are bound together by their very nature, as in the following teacher-student example:

```
class Student;   // forward declaration
class Teacher
{
    friend void registration(Teacher*, Student*);
  protected:
    int noStudents;
    Student *pList[128];

  public:
    void assignGrades();
};

class Student
{
    friend void registration(Teacher*, Student*);
  protected:
    Teacher *pTeacher;
    int nSemesterHours;
    double dGrade;
};
```

In this example, the `registration()` function can reach into both the `Student` object to set the `pTeacher` pointer and into the `Teacher` object to add to the teacher's list of students.

Notice how the class `Student` first appears by itself with no body. This is called a *forward declaration* and declares the intention of the programmer to define a class `Student` somewhere within the module. This is a little bit like the prototype declaration for a function. This is generally necessary only when two or more classes reference each other; in this case, `Teacher` contains a reference to `Student` and `Student` to `Teacher`.

Without the forward declaration to `Student`, the declaration within `Teacher` of `Student *pList[100]` generates a compiler error because the compiler doesn't yet know what a `Student` is. Swap the order of the definitions, and the declaration `Teacher *pTeacher` within `Student` generates a compiler error because `Teacher` has not been defined yet.

The forward declaration solves the problem by informing the compiler to be patient — a definition for this new class is coming very soon.

A member of one class can be declared a friend of another class:

```
class Student;

class Teacher
{
    // ...other members...
  public:
    void assignGrade(Student*, int nHours, double dGrade);
};

class Student
{
    friend void Teacher::assignGrade(Student*,
                                     int, double);
    // ...other members...
};
```

An entire class can be declared a friend of another class. This has the effect of making every member function of the class a friend. For example:

```
class Student;

class Teacher
{
  protected:
    int noStudents;
    Student* pList[128];

  public:
    void assignGrade(Student*, int nHours, double dGrade);
};

class Student
{
    friend class Teacher;

    // ...other members...
};
```

Now every member of `Teacher` can access the protected members of `Student` (but not the other way around). Declaring one class to be a friend of another binds the classes together inseparably.

Chapter 25

Getting Objects Off to a Good Start

In This Chapter

- Creating a constructor
- Examining limitations on how constructors are invoked
- Reviewing an example constructor
- Constructing data members
- Introducing the "not constructor" — the destructor

Normally an object is initialized when it is created as in the following:

```
double PI = 3.14159;
```

This is true of class objects as well:

```
class Student
{
  public:
    int nHours;
    double dGrade;
};

Student s = {0, 0.0};
```

However, this is no longer possible when the data elements are declared protected if the function that's creating the objects is not a friend or member of the class (which, in most cases it would not be).

Some other mechanism is required to initialize objects when they are created, and that's where the constructor comes in.

The Constructor

One approach to initializing objects with protected members would be to create an `init()` member function that the application could call when the

object is created. This `init()` function would initialize the object to some legal starting point. In fact, that's exactly what I do in Chapter 24.

This approach would work, but it doesn't exactly fit the "microwave oven" rules of object-oriented programming because it's akin to building a microwave oven that requires you to hit the Reset button before you could do anything with it. It's as if the manufacturer put some big disclaimer in the manual: "DO NOT start any sequence of commands without FIRST depressing the RESET button. Failure to do so may cause the oven to explode and kill everyone in the vicinity or WORSE." (What could be worse than that?)

Now I'm no lawyer, but even I know that putting a disclaimer like that in your manual is not going to save your butt when you end up in court because someone got cut with shrapnel from an exploding microwave, even though you say very clearly to hit reset first.

Fortunately, C++ takes the responsibility for calling the initialization function away from the applications programmer and calls the function automatically whenever an object is created.

You could call this initialization function anything you want as long as there is a rule for everyone to follow. (I'm kind of partial to `init()` myself, but I didn't get a vote.) The rule is that this initialization function is called a *constructor,* and it has the same name as the name of the class.

Outfitted with a constructor, the `Student` class appears as follows:

```
class Student
{
  protected:
    int nSemesterHours;
    double dGrade;

  public:
    Student()
    {
        nSemesterHours = 0;
        dGrade = 0.0;
    }

    // ...other public member functions...
};
void fn()
{
    Student s;  // create an object and invoke the
                // constructor on it
}
```

At the point of the declaration of `s`, C++ embeds a call to `Student::Student()`.

Notice that the constructor is called once for every object created. Thus, the following declaration calls the constructor five times in a row:

```
void fn()
{
    Student s[5];
}
```

It first calls the constructor for `s[0]`, then for `s[1]`, and so forth.

Limitations on constructors

The constructor can only be invoked automatically by C++. You cannot call a constructor like a normal member function. That is, you cannot do something like the following:

```
void fn()
{
    Student s;

    // ...do stuff...

    // now reinitialize s back to its initial state
    s.Student();    // this doesn't work
}
```

The constructor is not just any ol' function.

In addition, the constructor has no return type, not even `void`. The default constructor has no arguments either.

The next chapter shows you how to declare and use a constructor with arguments.

Finally, the constructor must be declared public, or else you will be able to create objects only from within other member functions.

The constructor can call other functions. Thus, your constructor could invoke a publicly available `init()` function that could then be used by anyone to reset the object to its initial state.

Can I see an example?

The following StudentConstructor program looks a lot like the SimpleStudent program from Chapter 24, except that this version includes a constructor that outputs every time it's creating an object. The interesting part to this program is seeing the cases during which the constructor is invoked.

I highly encourage you to single-step this program in the debugger using the Step-Into debugger command from Chapter 20. Use the Step Into debugger command near the declaration of the `Student` objects to step into the constructor automatically.

```
//
//  StudentConstructor - this program demonstrates the use
//                of a default constructor to initialize
//                objects when they are created
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  protected:
    double dGrade;          // the student's GPA
    int    nSemesterHours;

  public:
    // constructor - init the student to a legal state
    Student()
    {
        cout << "Constructing a Student object" << endl;
        dGrade = 0.0;
        nSemesterHours = 0;
    }

    // getGrade() - return the current grade
    double getGrade()
    {
        return dGrade;
    }

    // getHours() - get the class hours towards graduation
    int getHours()
    {
        return nSemesterHours;
    }

    // addGrade - add a grade to the GPA and total hours
    double addGrade(double dNewGrade, int nHours)
    {
```

```
        double dWtdHrs = dGrade * nSemesterHours;
        dWtdHrs += dNewGrade * nHours;
        nSemesterHours += nHours;
        dGrade = dWtdHrs / nSemesterHours;
        return dGrade;
    }
};

int main(int nNumberofArgs, char* pszArgs[])
{
    // create a student and initialize it
    cout << "Creating the Student s" << endl;
    Student s;

    // add the grades for three classes
    s.addGrade(3.0, 3);  // a B
    s.addGrade(4.0, 3);  // an A
    s.addGrade(2.0, 3);  // a C (average should be a B)

    // now print the results
    cout << "Total # hours = " << s.getHours()
         << ", GPA = " << s.getGrade()
         << endl;

    // create an array of Students
    cout << "Create an array of 5 Students" << endl;
    Student sArray[5];

    // now allocate one off of the heap
    cout << "Allocating a Student from the heap" << endl;
    Student *pS = new Student;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The output from this program appears as follows:

```
Creating the Student s
Constructing a Student object
Total # hours = 9, GPA = 3
Create an array of 5 Students
Constructing a Student object
Constructing a Student object
Constructing a Student object
Constructing a Student object
Constructing a Student object
Allocating a Student from the heap
Constructing a Student object
Press any key to continue . . .
```

The `Student` class has been outfitted with a constructor that not only initializes the number of semester hours and grade point average to zero but also outputs a message to the console to announce that a `Student` object is being created.

The `main()` program then simply creates `Student` objects in various ways:

- The first declaration creates a single `Student` object `s` resulting in C++ invoking the constructor.
- The second declaration creates an array of five `Student` objects. C++ calls the constructor five times, once for each object in the array.
- The program allocates a `Student` object from the heap. C++ invokes the constructor again to initialize the object.

Constructing data members

The data members of a class are created at the same time as the object itself. Consider the following simple class `TutorPair` consisting of a `Student` and a `Teacher`:

```
class TutorPair
{
  protected:
    Student s;
    Teacher t;

    int nNumberOfMeetings;

  public:
    TutorPair()
    {
        nNumberOfMeetings = 0;
    }

    // ...other stuff...
};
```

It's not the responsibility of the `TutorPair` class to initialize the member `Student` or the member `Teacher`; these objects should be initialized by constructors in their respective classes. The constructor for `TutorPair` is responsible only for initializing the non-class members of the class.

Thus, when a `TutorPair` is created, C++ does the following (in the order shown):

- It invokes the constructor for the `Student s`.
- It invokes the constructor for the `Teacher t`.
- It enters the constructor for `TutorPair` itself.

The constructors for the data members are invoked in the order that they appear in the class.

The following TutorPairConstructor program demonstrates:

```
//
//  TutorPairConstructor - this program demonstrates
//          how data members are constructed automatically
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  protected:
    double dGrade;          // the student's GPA
    int    nSemesterHours;

  public:
    // constructor - init the student to a legal state
    Student()
    {
        cout << "Constructing a Student object" << endl;
        dGrade = 0.0;
        nSemesterHours = 0;
    }
};

class Teacher
{
  public:
    // constructor - init the student to a legal state
    Teacher()
    {
        cout << "Constructing a Teacher object" << endl;
    }
};

class TutorPair
{
  protected:
    Student s;
```

```
    Teacher t;

    int nNumberOfMeetings;

  public:
    TutorPair()
    {
        cout << "Constructing the TutorPair members"
             << endl;
        nNumberOfMeetings = 0;
    }
};

int main(int nNumberofArgs, char* pszArgs[])
{
    // create a TutorPair and initialize it
    cout << "Creating the TutorPair tp" << endl;
    TutorPair tp;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The `main()` program does nothing more than output a message and then creates an object `tp` of class `TutorPair`. This causes C++ to invoke the constructor for `TutorPair`. However, before the first line of that function is executed, C++ goes through the data members and constructs any objects that it finds there.

The first object C++ sees is the `Student` object `s`. This constructor outputs the first message that you see on the output. The second object that C++ finds is the `Teacher` member `t`. This constructor generates the next line of output.

With all the data members out of the way, C++ passes control to the body of the `TutorPair` constructor that outputs the final line of output:

```
Creating the TutorPair tp
Constructing a Student object
Constructing a Teacher object
Constructing the TutorPair members
Press any key to continue . . .
```