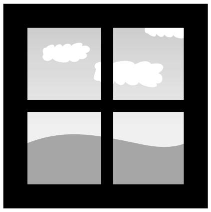

SESSION 13

A Few C++ Pointers

Session Checklist

- ✔ Addressing variables in memory
- ✔ Introducing the pointer data type
- ✔ Recognizing the inherent dangers of pointers
- ✔ Passing pointers to functions
- ✔ Allocating objects off the free store, which is commonly known as the heap

30 Min. To Go

Part II introduced the C++ operators. Operations such as addition, multiplication, bitwise AND, and logical OR were performed on intrinsic variable types such as `int` and `float`. There is another intrinsic variable type that we have yet to cover — pointers.

To anyone familiar with other programming languages, C++ seems like a conventional language, so far. Many languages don't include the logical operators presented and C++ presents its own unique semantics, but there are no concepts present in C++ that other languages do not offer. The introduction of pointers to the language is the initial departure of C++ from other, more conventional languages.

Note

Pointers were actually introduced in the predecessor to C++, the C programming language. Everything described in this chapter works the same way in C.

Pointers are a mixed blessing. While pointers provide capabilities unique to C++, they can be syntactically difficult and they provide many opportunities to screw up.

This chapter introduces the pointer variable type. It begins with some concept definitions, flows through pointer syntax, and then introduces some of the reasons for the pointer mania that grips C++ programmers.

What's Your Address?

Just as in the saying "everyone has to be somewhere," every variable is stored somewhere in the computer's memory. Memory is divided into individual bytes, with each byte carrying its own address numbered 0, 1, 2, and so forth.

Note

By convention, memory addresses are expressed in hexadecimal.

Different variable types require a different number of bytes of storage. Table 13-1 shows the memory that each variable type consumes in Visual C++ 6 and GNU C++ on a Pentium processor.

Table 13-1
Memory Requirements for Different Variable Types

Variable Type	Memory Consumed [bytes]
`int`	4
`long`	4
`float`	4
`double`	8

Consider the following Layout test program, which demonstrates the layout of variables in memory. (Ignore the new `&` operator — suffice it to say that `&n` returns the address of the variable `n`.)

```
// Layout - this program tries to give the
//          reader an idea of the layout of
//          local memory in her compiler
#include <stdio.h>
#include <iostream.h>

int main(int nArgc, char* pszArgs[])
{
    int    m1;
    int    n;
    long   l;
    float  f;
    double d;
    int    m2;

    // set output to hex mode
    cout.setf(ios::hex);

    // output the address of each variable
    // in order to get an idea of the size
    // of each variable
    cout << “--- = 0x” << (long)&m1 << “\n”;
    cout << “&n  = 0x” << (long)&n  << “\n”;
    cout << “&l  = 0x” << (long)&l  << “\n”;
    cout << “&f  = 0x” << (long)&f  << “\n”;
    cout << “&d  = 0x” << (long)&d  << “\n”;
    cout << “--- = 0x” << (long)&m2 << “\n”;

    return 0;
}
```

The output from this program is shown in Listing 13-1. From this, you can see that the variable n was stored at location 0x65fdf0.

Note

Don’t worry if the values you see when running this program are different. The relationship between the locations is what matters.

From the comparison of locations we can also infer that the size of n is 4 bytes (0x65fdf4 – 0x65fdf0), the size of the long l is also 4 (0x65fdf0 – 0x65fdec), and so forth.

This demonstration only makes sense if you assume that variables are laid out immediately next to each other, which is the case in GNU C++. This is only the case in Visual C++ if a certain project setting is specified correctly.

Listing 13-1
Results of executing the Layout program

```
--- = 0x65fdf4
&n  = 0x65fdf0
&l  = 0x65fdec
&f  = 0x65fde8
&d  = 0x65fde0
--- = 0x65fddc
Press any key to continue
```

There is nothing in the definition of C++ that dictates that the layout of variables is anything like that shown in Listing 13-1. As far as we are concerned, it is purely an accident that GNU C++ and Visual C++ chose the same variable layout.

Introduction to Pointer Variables

Let's begin with a new definition and a couple of new operators. A *pointer variable* is a variable that contains an address, usually the address of another variable. Add to this definition the new operators shown in Table 13-2.

Table 13-2
Pointer Operators

Operator	Meaning
& (unary)	the address of
* (unary)	(in an expression) the thing pointed at by (in a declaration) pointer to

The use of these new features is best described by example:

```
void fn()
{
   int nInt;
   int* pnInt;

   pnInt  = &nInt;    // pnInt now points to nInt
   *pnInt = 10;       // stores 10 into int location
                      // pointed at by pnInt
}
```

The function `fn()` begins with the declaration of `nInt`. The next statement declares the variable `pnInt` to be a variable of type “pointer to an `int`.”

Note

Pointer variables are declared similar to normal variables except for the addition of the unary `*` character. This “splat” character can appear anywhere between the base type name, in this case `int`, and the variable name; however, it is becoming increasingly common to add the splat to the end of the variable type.

Note

Just as the names of integer variables start with an n, pointer variables begin with a p. Thus, `pnX` is a pointer to an integer variable `X`. Again, this is just a convention to help keep things straight — C++ doesn’t care what names you use.

In an expression, the unary operator `&` means “the address of.” Thus, we would read the first assignment as “store the address of `nInt` in `pnInt`.”

To make this more concrete, let’s assume that the memory for function `fn()` starts at location 0x100. In addition, let’s assume that `nInt` is at address 0x102 and that `pnInt` is at 0x106. The lay out here is simpler than the actual results from the `Layout` program, but the concepts are identical.

The first assignment is shown in Figure 13-1. Here you can see that the value of `&nInt` (0x102) is stored in `pnInt`.

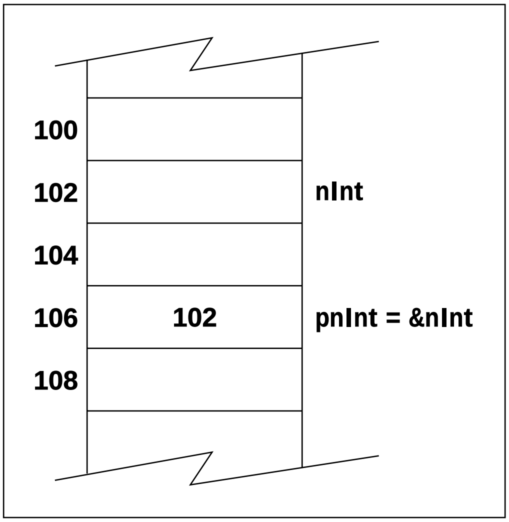

Figure 13-1
Storing the address of nInt in pnInt.

The second assignment in the small program snippet says "store 10 in the location pointed at by pnInt." Figure 13-2 demonstrates this. The value 10 is stored in the address contained in pnInt, which is 0x102 (the address of nInt).

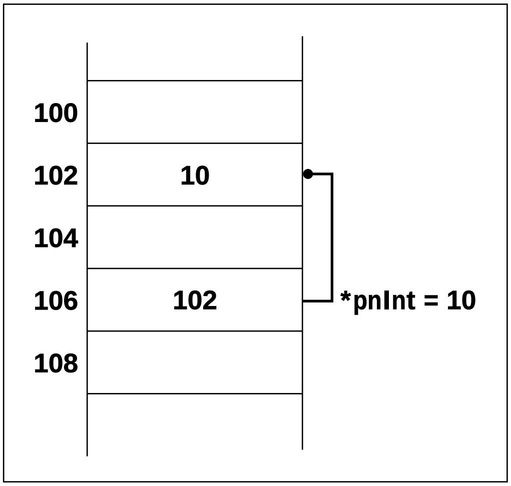

Figure 13-2
Storing 10 in the address pointed at by pnInt.

20 Min.
To Go

Types of Pointers

Remember that every expression has a type as well as a value. The type of the &nInt expression is a pointer to an integer, which is written as int*. A comparison of this expression type to the declaration of pnInt shows that the types match exactly:

```
pnInt = &nInt;  // both sides of the assignment are of type int*
```

Similarly, because pnInt is of type int*, the type of *pnInt is int.

```
*pnInt = 10;    // both sides of the assignment are of type int
```

Expressed in plain English, the type of the thing pointed to by pnInt is int.

Note

Despite the fact that a pointer variable has different types, such as int* and double*, the pointer is an intrinsic type. Irrespective of what it might point to, a pointer on a Pentium class machine takes 4 bytes.

Matching types is extremely important. Consider what might happen if the following were allowed:

```
int   n1;
int* pnInt;
pnInt = &n1;
*pnInt = 100.0;
```

The second assignment attempts to store the 8-byte double value 100.0 in the 4-byte space allocated for n1. The result is that variables nearby are wiped out. This is demonstrated graphically in the following LayoutError program shown in Listing 13-2. The output from this program appears at the end of the listing .

Listing 13-2
LayoutError program with output shown

```
// LayoutError - demonstrate the results of
// messing up a pointer usage
#include <stdio.h>
#include <iostream.h>

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

Continued

Listing 13-2 *Continued*

```
{
    int    upper = 0;
    int    n     = 0;
    int    lower = 0;

    // output the values of the three variables before...
    cout << “upper = “ << upper << “\n”;
    cout << “n     = “ << n     << “\n”;
    cout << “lower = “ << lower << “\n”;

    // now store a double in the space
    // allocated for an int
    cout << “\nPerforming assignment of double\n”;
    double* pD = (double*)&n;
    *pD = 13.0;

    // display the results
    cout << “upper = “ << upper << “\n”;
    cout << “n     = “ << n     << “\n”;
    cout << “lower = “ << lower << “\n”;

    return 0;
}
```

Output:

```
upper = 0
n     = 0
lower = 0

Performing assignment of double
upper = 1076494336
n     = 0
lower = 0
Press any key to continue
```

The first three lines in `main()` declare three integers in the normal fashion. The assumption here is that these three variables are laid out next to each other.

The next three executable lines output the value of the three variables. Not surprisingly, all three variables display as 0. The assignment `*pD = 13.0;` stores the double value 13.0 into the integer variable n. The three output statements display the values of all three variables after the assignment.

After assigning the double value 13.0 to the integer variable n, n itself is not modified; however, the nearby variable upper is filled with a garbage value.

Note

The following casts a pointer from one type to another:

```
double* pD = (double*)&n;
```

Here &n **is of type** int* **whereas the variable** pD **is of type** double***. The cast** (double*) **changes the type of the value** &n **to match that of** pD **in the same way that:**

```
double d = (double)n;
```

casts the int **value contained in** n **into a double.**

Passing Pointers to Functions

One of the uses of pointer variables is in passing arguments to functions. To understand why this is important, you need to understand how arguments are passed to a function.

Pass by value

You may have noticed that it is not normally possible to change the value of a variable passed to a function from within the function. Consider this code segment:

```
void fn(int nArg)
{
    nArg = 10;
    // value of nArg at this point is 10
}

void parent(void)
{
    int n1 = 0;
    fn(n1);
    // value of n1 at this point is 0
}
```

The parent() function initializes the integer variable n1 to zero. The value of n1 is then passed to fn(). On entering the function, nArg is equal to 10, the value passed. fn() changes the value of nArg before returning to parent(). Perhaps surprisingly, on return to parent(), the value of n1 is still 0.

The reason is that C++ doesn't pass a variable to a function. Instead, C++ passes the value contained in the variable at the time of the call. That is, the expression is evaluated, even if it is just a variable name, and the result is passed. Passing the value of a variable to a function is known as *pass by value.*

Tip

It is easy for a speaker to get lazy and say something similar to "pass the variable x to the function fn()." This really means, "pass the value of the expression x."

Passing pointer values

Like any other intrinsic type, a pointer may be passed as an argument to a function:

```
void fn(int* pnArg)
{
   *pnArg= 10;
}

void parent(void)
{
   int n = 0;
fn(&i);        // this passes the address of i
                  // now the value of n is 10
}
```

In this case, the address of n is passed to the function fn() rather than the value of n. The significance of this difference is apparent when you consider the assignment in fn().

Let's return to our earlier example values. Suppose n is located at address 0x102. Rather than the value 10, the call fn(&n) passes the "value" 0x106. Within fn(), the assignment *pnArg = 10 stores the value 10 in the int variable located at location 0x102, thereby overwriting the value 0. On return to parent(), the value of n is 10 because n is just another name for 0x102.

Passing by reference

C++ provides a shorthand method for passing the variable that doesn't involve the hassle of dealing with pointers yourself. In the following example, the variable `n` is passed by reference. In *pass by reference* the parent function passes a reference to the variable rather than the value. Reference is another word for address.

```
void fn(int& nArg)
{
    nArg = 10;
}

void parent(void)
{
    int n = 0;
fn(n)
                    // here the value of n is 10
}
```

In this case, a reference to `n` is passed to `fn()` rather than the value. The `fn()` function stores the value 10 into `int` location referenced by `nArg`.

Note

Notice that "reference" is not an actual type. Thus, the function's full name is `fn(int)` and not `fn(int&)`.

10 Min.
To Go

Heap Memory

Just as it is possible to pass a pointer to a function, it is also possible for a function to return a pointer. A function that returns the address of a `double` would be declared as follows:

```
double* fn(void);
```

However, one must be very careful when returning a pointer. To understand the dangers, you must know something about variable scope.

Scope

C++ variables have a property in addition to their value and type known as scope. *Scope* is the range over which a variable is defined. Consider this code snippet:

```
// the following variable is accessible to
// all functions and defined as long as the
// program is running(global scope)
int nGlobal;

// the following variable nChild is accessible
// only to the function and is defined only
// as long as C++ is executing child() or a
// function which child() calls (function scope)
void child(void)
{
    int nChild;
}

// the following variable nParent has function
// scope
void parent(void)
{
    int nParent = 0;
    fn();

    int nLater = 0;
    nParent = nLater;
}

int main(int nArgs, char* pArgs[])
{
    parent();
}
```

Execution begins with `main()`. The function `main()` immediately invokes `parent()`. The first thing that the processor sees in `parent()` is the declaration of `nParent`. At that point, `nParent` goes into scope — that is, `nParent` is defined and available for the remainder of the function `parent()`.

The second statement in parent() is the call to child(). Once again, the function child() declares a local variable, this time nChild. The variable nChild is within the scope of child(). Technically, nParent is not within the scope of child()because child() doesn't have access to nParent; however, the variable nParent continues to exist.

When child() exits, the variable nChild goes out of scope. Not only is nChild no longer accessible but it no longer even exists. (The memory occupied by nChild is returned to the general pool to be used for other things.)

As parent() continues executing, the variable nLater goes into scope at the declaration. At the point that parent() returns to main(), both nParent and nLater go out of scope.

The programmer may declare a variable outside of any function. A *global variable* is a variable declared outside of any function. Such a variable remains in scope for the duration of the program.

Because nGlobal is declared globally in this example, it is available to all three functions and remains available for the life of the program.

The scope problem

The following code segment compiles without error but does not work:

```
double* child(void)
{
    double dLocalVariable;
    return &dLocalVariable;
}

void parent(void)
{
    double* pdLocal;
    pdLocal = child();
    *pdLocal = 1.0;
}
```

The problem is that dLocalVariable is defined only within the scope of the function fn(). Thus, by the time that the memory address of dLocalVariable is returned from child() it refers to a variable that no longer exists. The memory that dLocalVariable formerly occupied is probably being used for something else.

This is a very common error because it can creep up in a number of different ways. Unfortunately, this error does not cause the program to instantly stop. In fact, the program may work perfectly well most of the time — as long as the memory formerly occupied by dLocalVariable is not reused immediately, the program continues to work. Such intermittent problems are the most difficult to solve.

The heap solution

The scope problem occurs because C++ returns the locally defined memory before the programmer is ready. What is needed is a block of memory controlled by the programmer. The programmer can allocate the memory and put it back when the programmer wants and not because C++ thinks it a good idea. Such a block of memory is called the heap.

The *heap* is a segment of memory that is explicitly controlled by the program.

Heap memory is allocated using the new command followed by the type of object to allocate. For example, the following allocates a double variable off the heap.

```
double* child(void)
{
    double* pdLocalVariable = new double;
    return pdLocalVariable;
}
```

Even though the variable pdLocalVariable goes out of scope when the function child() returns, the memory to which pdLocalVariable refers does not.

A memory location returned by new does not go out of scope until it is explicitly returned to the heap using the delete command:

```
void parent(void)
{
    // child() returns the address of a block
    // of heap memory
    double* pdMyDouble = child();

    // store a value there
    *pdMyDouble = 1.1;
    // ...

    // now return the memory to the heap
    delete pdMyDouble;
    pdMyDouble = 0;

    // ...
}
```

Done!

Here the pointer returned by `child()` is used to store a double value. After the function is finished with the memory location, it is returned to the heap. Setting the pointer to 0 after the `delete` is not necessary, but it is a good idea. That way, if the programmer mistakenly attempts to store something in `*pdMyDouble` after the `delete`, the program will crash immediately.

Tip

A program that crashes immediately on encountering an error is much easier to fix than one that is intermittent in its behavior.

Pointers to Functions

In addition to pointers to intrinsic types, it is possible to declare pointers to functions. For example, the following declares a pointer to a function that takes an integer argument and returns a double:

```
double (*pFN)(int);
```

There are a number of uses for a variable such as this; the twists and turns of pointers to functions, however, are beyond the scope of this book.

REVIEW

Pointer variables are a powerful, if dangerous, mechanism for accessing objects by their memory address. This is probably the single most important language feature and it is probably the feature most responsible for the dominance of C and later C++ over other computer languages.

- Pointer variables are declared by adding a '*', known as the asterisk or splat character, to the variable type. The splat character may appear anywhere between the variable name and the root type; however, it makes more sense to append the splat to the end of the variable type.
- The address of operator & returns the address of an object while the splat operator * returns the object pointed at by an address or a pointer type variable.
- Variable types such as `int*` are their own variable types and are not equivalent to `int`. The address of operator & converts a type such as `int` into a pointer type such as `int*`. The splat operator * converts a pointer type such as `int*` into a base type such as `int`. One pointer type can be converted to another, but at considerable risk.

Subsequent lessons review some of the additional ways that pointers can be used to extend the C++ programmer's bag of tricks.

QUIZ YOURSELF

1. If a variable `x` contains the value 10 and is stored at location 0x100 what is the value of `x`? What is the value of `&x`? (See "Introduction to Pointer Variables.")
2. If x is an `int`, what is the type of `&x`? (See "Types of Pointers.")
3. Why would you pass a pointer to a function? (See "Passing Pointer Values.")
4. What is heap memory and how do you gain access to it? (See "Heap Memory.")