Value Ranges for Enumerations

Originally, the only valid values for an enumeration were those named in the declaration. However, C++ has expanded the list of valid values that can be assigned to an enumeration variable through the use of a type cast. Each enumeration has a *range*, and you can assign any integer value in the range, even if it's not an enumerator value, by using a type cast to an enumeration variable. For example, suppose that `bits` and `myflag` are defined this way:

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
bits myflag;
```

In this case, the following is valid:

```
myflag = bits(6);    // valid, because 6 is in bits range
```

Here `6` is not one of the enumerations, but it lies in the range the enumerations define.

The range is defined as follows. First, to find the upper limit, you take the largest enumerator value. Then you find the smallest power of two greater than this largest value and subtract one; the result is the upper end of the range. (For example, the largest `bigstep` value, as previously defined, is 101. The smallest power of two greater than this is 128, so the upper end of the range is 127.) Next, to find the lower limit, you find the smallest enumerator value. If it is 0 or greater, the lower limit for the range is 0. If the smallest enumerator is negative, you use the same approach as for finding the upper limit but toss in a minus sign. (For example, if the smallest enumerator is -6, the next power of two [times a minus sign] is -8, and the lower limit is -7.)

The idea is that the compiler can choose how much space to use to hold an enumeration. It might use 1 byte or less for an enumeration with a small range and 4 bytes for an enumeration with type `long` values.

C++11 extends enumerations with a form called the *scoped enumeration*. Chapter 10 discusses this form briefly in the section "Class Scope."

Pointers and the Free Store

The beginning of Chapter 3 mentions three fundamental properties of which a computer program must keep track when it stores data. To save the book the wear and tear of your thumbing back to that chapter, here are those properties again:

- Where the information is stored
- What value is kept there
- What kind of information is stored

You've used one strategy for accomplishing these ends: defining a simple variable. The declaration statement provides the type and a symbolic name for the value. It also causes the program to allocate memory for the value and to keep track of the location internally.

Let's look at a second strategy now, one that becomes particularly important in developing C++ classes. This strategy is based on pointers, which are variables that store addresses of values rather than the values themselves. But before discussing pointers, let's talk about how to explicitly find addresses for ordinary variables. You just apply the address operator, represented by `&`, to a variable to get its location; for example, if `home` is a variable, `&home` is its address. Listing 4.14 demonstrates this operator.

Listing 4.14 **address.cpp**

```
// address.cpp -- using the & operator to find addresses
#include <iostream>
int main()
{
    using namespace std;
    int donuts = 6;
    double cups = 4.5;

    cout << "donuts value = " << donuts;
    cout << " and donuts address = " << &donuts << endl;
// NOTE: you may need to use unsigned (&donuts)
// and unsigned (&cups)
    cout << "cups value = " << cups;
    cout << " and cups address = " << &cups << endl;
    return 0;
}
```

Here is the output from the program in Listing 4.14 on one system:

```
donuts value = 6 and donuts address = 0x0065fd40
cups value = 4.5 and cups address = 0x0065fd44
```

The particular implementation of `cout` shown here uses hexadecimal notation when displaying address values because that is the usual notation used to specify a memory address. (Some implementations use base 10 notation instead.) Our implementation stores `donuts` at a lower memory location than `cups`. The difference between the two addresses is 0x0065fd44 - 0x0065fd40, or 4. This makes sense because `donuts` is type `int`, which uses 4 bytes. Different systems, of course, will give different values for the address. Also some may store `cups` first, then `donuts`, giving a difference of 8 bytes because `cups` is `double`. And some may not even use adjacent locations.

Using ordinary variables, then, treats the value as a named quantity and the location as a derived quantity. Now let's look at the pointer strategy, one that is essential to the C++ programming philosophy of memory management. (See the following sidebar, "Pointers and the C++ Philosophy.")

Pointers and the C++ Philosophy

Object-oriented programming differs from traditional procedural programming in that OOP emphasizes making decisions during runtime instead of during compile time. *Runtime* means while a program is running, and *compile time* means when the compiler is putting a program together. A runtime decision is like, when on vacation, choosing what sights to see depending on the weather and your mood at the moment, whereas a compile-time decision is more like adhering to a preset schedule, regardless of the conditions.

Runtime decisions provide the flexibility to adjust to current circumstances. For example, consider allocating memory for an array. The traditional way is to declare an array. To declare an array in C++, you have to commit yourself to a particular array size. Thus, the array size is set when the program is compiled; it is a compile-time decision. Perhaps you think an array of 20 elements is sufficient 80% of the time but that occasionally the program will need to handle 200 elements. To be safe, you use an array with 200 elements. This results in your program wasting memory most of the time it's used. OOP tries to make a program more flexible by delaying such decisions until runtime. That way, after the program is running, you can tell it you need only 20 elements one time or that you need 205 elements another time.

In short, with OOP you would like to make the array size a runtime decision. To make this approach possible, the language has to allow you to create an array—or the equivalent—while the program runs. The C++ method, as you soon see, involves using the keyword `new` to request the correct amount of memory and using pointers to keep track of where the newly allocated memory is found.

Making runtime decisions is not unique to OOP. But C++ makes writing the code a bit more straightforward than does C.

The new strategy for handling stored data switches things around by treating the location as the named quantity and the value as a derived quantity. A special type of variable—the *pointer*—holds the address of a value. Thus, the name of the pointer represents the location. Applying the `*` operator, called the *indirect value* or the *dereferencing* operator, yields the value at the location. (Yes, this is the same `*` symbol used for multiplication; C++ uses the context to determine whether you mean multiplication or dereferencing.) Suppose, for example, that `manly` is a pointer. In that case, `manly` represents an address, and `*manly` represents the value at that address. The combination `*manly` becomes equivalent to an ordinary type `int` variable. Listing 4.15 demonstrates these ideas. It also shows how to declare a pointer.

Listing 4.15 **pointer.cpp**

```
// pointer.cpp -- our first pointer variable
#include <iostream>
int main()
{
    using namespace std;
    int updates = 6;        // declare a variable
    int * p_updates;        // declare pointer to an int
```

```
    p_updates = &updates;   // assign address of int to pointer

// express values two ways
    cout << "Values: updates = " << updates;
    cout << ", *p_updates = " << *p_updates << endl;

// express address two ways
    cout << "Addresses: &updates = " << &updates;
    cout << ", p_updates = " << p_updates << endl;

// use pointer to change value
    *p_updates = *p_updates + 1;
    cout << "Now updates = " << updates << endl;
    return 0;
}
```

Here is the output from the program in Listing 4.15:

```
Values: updates = 6, *p_updates = 6
Addresses: &updates = 0x0065fd48, p_updates = 0x0065fd48
Now updates = 7
```

As you can see, the `int` variable `updates` and the pointer variable `p_updates` are just two sides of the same coin. The `updates` variable represents the value as primary and uses the `&` operator to get the address, whereas the `p_updates` variable represents the address as primary and uses the `*` operator to get the value (see Figure 4.8). Because `p_updates` points to `updates`, `*p_updates` and `updates` are completely equivalent. You can use `*p_updates` exactly as you would use a type `int` variable. As the program in Listing 4.15 shows, you can even assign values to `*p_updates`. Doing so changes the value of the pointed-to value, `updates`.

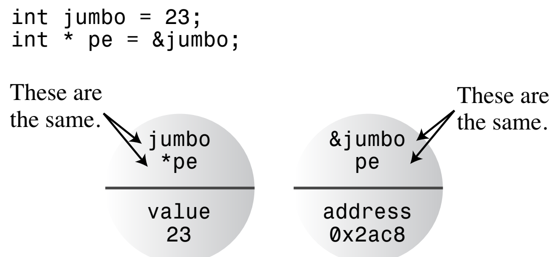

Figure 4.8 Two sides of a coin.

Declaring and Initializing Pointers

Let's examine the process of declaring pointers. A computer needs to keep track of the type of value to which a pointer refers. For example, the address of a `char` typically looks the same as the address of a `double`, but `char` and `double` use different numbers of bytes and different internal formats for storing values. Therefore, a pointer declaration must specify what type of data to which the pointer points.

For example, the preceding example has this declaration:

```
int * p_updates;
```

This states that the combination `* p_updates` is type `int`. Because you use the `*` operator by applying it to a pointer, the `p_updates` variable itself must *be* a pointer. We say that `p_updates` points to type `int`. We also say that the type for `p_updates` is pointer-to-`int` or, more concisely, `int *`. To repeat: `p_updates` is a pointer (an address), and `*p_updates` is an `int` and not a pointer (see Figure 4.9).

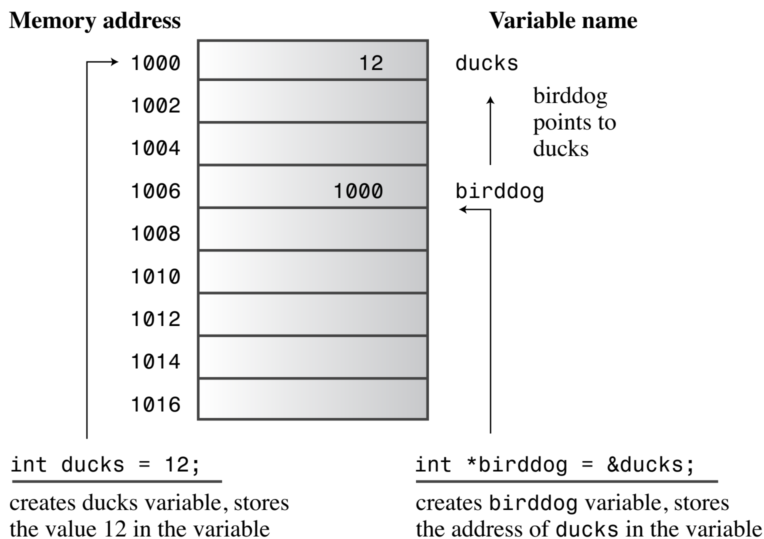

Figure 4.9 Pointers store addresses.

Incidentally, the use of spaces around the `*` operator are optional. Traditionally, C programmers have used this form:

```
int *ptr;
```

This accentuates the idea that the combination `*ptr` is a type `int` value. Many C++ programmers, on the other hand, use this form:

```
int* ptr;
```

This emphasizes the idea that `int*` is a type, pointer-to-`int`. Where you put the spaces makes no difference to the compiler. You could even do this:

```
int*ptr;
```

Be aware, however, that the following declaration creates one pointer (`p1`) and one ordinary `int` (`p2`):

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

You need an `*` for each pointer variable name.

Note

In C++, the combination `int *` is a compound type, pointer-to-`int`.

You use the same syntax to declare pointers to other types:

```
double * tax_ptr; // tax_ptr points to type double
char * str;       // str points to type char
```

Because you declare `tax_ptr` as a pointer-to-`double`, the compiler knows that `*tax_ptr` is a type `double` value. That is, it knows that `*tax_ptr` represents a number stored in floating-point format that occupies (on most systems) 8 bytes. A pointer variable is never simply a pointer. It is always a pointer to a specific type. `tax_ptr` is type pointer-to-`double` (or type `double *`), and `str` is type pointer-to-`char` (or `char *`). Although both are pointers, they are pointers of two different types. Like arrays, pointers are based on other types.

Note that whereas `tax_ptr` and `str` point to data types of two different sizes, the two variables `tax_ptr` and `str` themselves are typically the same size. That is, the address of a `char` is the same size as the address of a `double`, much as 1016 might be the street address for a department store, whereas 1024 could be the street address of a small cottage. The size or value of an address doesn't really tell you anything about the size or kind of variable or building you find at that address. Usually, addresses require 2 or 4 bytes, depending on the computer system. (Some systems might have larger addresses, and a system can use different address sizes for different types.)

You can use a declaration statement to initialize a pointer. In that case, the pointer, not the pointed-to value, is initialized. That is, the following statements set `pt` and not `*pt` to the value `&higgens`:

```
int higgens = 5;
int * pt = &higgens;
```

Listing 4.16 demonstrates how to initialize a pointer to an address.

Listing 4.16 **init_ptr.cpp**

```
// init_ptr.cpp -- initialize a pointer
#include <iostream>
int main()
{
```

```
    using namespace std;
    int higgens = 5;
    int * pt = &higgens;

    cout << "Value of higgens = " << higgens
         << "; Address of higgens = " << &higgens << endl;
    cout << "Value of *pt = " << *pt
         << "; Value of pt = " << pt << endl;
    return 0;
}
```

Here is some sample output from the program in Listing 4.16:

```
Value of higgens = 5; Address of higgens = 0012FED4
Value of *pt = 5; Value of pt = 0012FED4
```

You can see that the program initializes `pt`, not `*pt`, to the address of `higgens`. (Your system most likely will show different values for the addresses and may display them in a different format.)

Pointer Danger

Danger awaits those who incautiously use pointers. One extremely important point is that when you create a pointer in C++, the computer allocates memory to hold an address, but it does not allocate memory to hold the data to which the address points. Creating space for the data involves a separate step. Omitting that step, as in the following, is an invitation to disaster:

```
long * fellow;          // create a pointer-to-long
*fellow = 223323;       // place a value in never-never land
```

Sure, `fellow` is a pointer. But where does it point? The code failed to assign an address to `fellow`. So where is the value `223323` placed? We can't say. Because `fellow` wasn't initialized, it could have any value. Whatever that value is, the program interprets it as the address at which to store `223323`. If `fellow` happens to have the value `1200`, then the computer attempts to place the data at address `1200`, even if that happens to be an address in the middle of your program code. Chances are that wherever `fellow` points, that is not where you want to put the number `223323`. This kind of error can produce some of the most insidious and hard-to-trace bugs.

> **Caution**
>
> Pointer Golden Rule: *Always* initialize a pointer to a definite and appropriate address before you apply the dereferencing operator (`*`) to it.

Pointers and Numbers

Pointers are not integer types, even though computers typically handle addresses as integers. Conceptually, pointers are distinct types from integers. Integers are numbers you can add, subtract, divide, and so on. But a pointer describes a location, and it doesn't make sense, for example, to multiply two locations by each other. In terms of the operations you can perform with them, pointers and integers are different from each other. Consequently, you can't simply assign an integer to a pointer:

```
int * pt;
pt = 0xB8000000;  // type mismatch
```

Here, the left side is a pointer to `int`, so you can assign it an address, but the right side is just an integer. You might know that 0xB8000000 is the combined segment-offset address of video memory on your aging computer system, but nothing in the statement tells the program that this number is an address. C prior to C99 lets you make assignments like this. But C++ more stringently enforces type agreement, and the compiler will give you an error message saying you have a type mismatch. If you want to use a numeric value as an address, you should use a type cast to convert the number to the appropriate address type:

```
int * pt;
pt = (int *) 0xB8000000; // types now match
```

Now both sides of the assignment statement represent addresses of integers, so the assignment is valid. Note that just because it is the address of a type `int` value doesn't mean that `pt` itself is type `int`. For example, one might have a platform for which type `int` is a 2-byte value and the addresses are 4-byte values.

Pointers have some other interesting properties that we'll discuss as they become relevant. Meanwhile, let's look at how pointers can be used to manage runtime allocation of memory space.

Allocating Memory with `new`

Now that you have a feel for how pointers work, let's see how they can implement the important technique of allocating memory as a program runs. So far, you've initialized pointers to the addresses of variables; the variables are *named* memory allocated during compile time, and each pointer merely provides an alias for memory you could access directly by name anyway. The true worth of pointers comes into play when you allocate *unnamed* memory during runtime to hold values. In this case, pointers become the only access to that memory. In C, you can allocate memory with the library function `malloc()`. You can still do so in C++, but C++ also has a better way: the `new` operator.

Let's try out this new technique by creating unnamed runtime storage for a type `int` value and accessing the value with a pointer. The key is the C++ `new` operator. You tell `new` for what data type you want memory; `new` finds a block of the correct size and

returns the address of the block. You assign this address to a pointer, and you're in business. Here's an example of the technique:

```
int * pn = new int;
```

The `new int` part tells the program you want some new storage suitable for holding an `int`. The `new` operator uses the type to figure out how many bytes are needed. Then it finds the memory and returns the address. Next, you assign the address to `pn`, which is declared to be of type pointer-to-`int`. Now `pn` is the address and `*pn` is the value stored there. Compare this with assigning the address of a variable to a pointer:

```
int higgens;
int * pt = &higgens;
```

In both cases (`pn` and `pt`), you assign the address of an `int` to a pointer. In the second case, you can also access the `int` by name: `higgens`. In the first case, your only access is via the pointer. That raises a question: Because the memory to which `pn` points lacks a name, what do you call it? We say that `pn` points to a *data object*. This is not "object" in the sense of "object-oriented programming"; it's just "object" in the sense of "thing." The term "data object" is more general than the term "variable" because it means any block of memory allocated for a data item. Thus, a variable is also a data object, but the memory to which `pn` points is not a variable. The pointer method for handling data objects may seem more awkward at first, but it offers greater control over how your program manages memory.

The general form for obtaining and assigning memory for a single data object, which can be a structure as well as a fundamental type, is this:

```
typeName * pointer_name = new typeName;
```

You use the data type twice: once to specify the kind of memory requested and once to declare a suitable pointer. Of course, if you've already declared a pointer of the correct type, you can use it rather than declare a new one. Listing 4.17 illustrates using `new` with two different types.

Listing 4.17 **use_new.cpp**

```
// use_new.cpp -- using the new operator
#include <iostream>
int main()
{
    using namespace std;
    int nights = 1001;
    int * pt = new int;         // allocate space for an int
    *pt = 1001;                 // store a value there

    cout << "nights value = ";
    cout << nights << ": location " << &nights << endl;
    cout << "int ";
    cout << "value = " << *pt << ": location = " << pt << endl;
```

```
    double * pd = new double;    // allocate space for a double
    *pd = 10000001.0;            // store a double there

    cout << "double ";
    cout << "value = " << *pd << ": location = " << pd << endl;
    cout << "location of pointer pd: " << &pd << endl;
    cout << "size of pt = " << sizeof(pt);
    cout << ": size of *pt = " << sizeof(*pt) << endl;
    cout << "size of pd = " << sizeof pd;
    cout << ": size of *pd = " << sizeof(*pd) << endl;
    return 0;
}
```

Here is the output from the program in Listing 4.17:

```
nights value = 1001: location 0028F7F8
int value = 1001: location = 00033A98
double value = 1e+007: location = 000339B8
location of pointer pd: 0028F7FC
size of pt = 4: size of *pt = 4
size of pd = 4: size of *pd = 8
```

Of course, the exact values for the memory locations differ from system to system.

Program Notes

The program in Listing 4.17 uses `new` to allocate memory for the type `int` and type `double` data objects. This occurs while the program is running. The pointers `pt` and `pd` point to these two data objects. Without them, you cannot access those memory locations. With them, you can use `*pt` and `*pd` just as you would use variables. You assign values to `*pt` and `*pd` to assign values to the new data objects. Similarly, you print `*pt` and `*pd` to display those values.

The program in Listing 4.17 also demonstrates one of the reasons you have to declare the type a pointer points to. An address in itself reveals only the beginning address of the object stored, not its type or the number of bytes used. Look at the addresses of the two values. They are just numbers with no type or size information. Also note that the size of a pointer-to-`int` is the same as the size of a pointer-to-`double`. Both are just addresses. But because `use_new.cpp` declares the pointer types, the program knows that `*pd` is a `double` value of 8 bytes, whereas `*pt` is an `int` value of 4 bytes. When `use_new.cpp` prints the value of `*pd`, `cout` can tell how many bytes to read and how to interpret them.

Another point to note is that typically `new` uses a different block of memory than do the ordinary variable definitions that we have been using. Both the variables `nights` and `pd` have their values stored in a memory region called the *stack*, whereas the memory allocated by `new` is in a region called the *heap* or *free store*. Chapter 9 discusses this a bit further.

Out of Memory?

It's possible that a computer might not have sufficient memory available to satisfy a `new` request. When that is the case, `new` normally responds by throwing an exception, an error-handling technique discussed in Chapter 15, "Friends, Exceptions, and More." In older implementations `new` returns the value `0`. In C++, a pointer with the value `0` is called the *null pointer.* C++ guarantees that the null pointer never points to valid data, so it is often used to indicate failure for operators or functions that otherwise return usable pointers. The `if` statement, discussed in Chapter 6, helps you deal with this possibility. For now, the important point is that C++ provides the tools to detect and respond to allocation failures.

Freeing Memory with `delete`

Using `new` to request memory when you need it is just the more glamorous half of the C++ memory-management package. The other half is the `delete` operator, which enables you to return memory to the memory pool when you are finished with it. That is an important step toward making the most effective use of memory. Memory that you return, or *free*, can then be reused by other parts of the program. You use `delete` by following it with a pointer to a block of memory originally allocated with `new`:

```
int * ps = new int; // allocate memory with new
. . .               // use the memory
delete ps;          // free memory with delete when done
```

This removes the memory to which `ps` points; it doesn't remove the pointer `ps` itself. You can reuse `ps`, for example, to point to another `new` allocation. You should always balance a use of `new` with a use of `delete`; otherwise, you can wind up with a *memory leak*—that is, memory that has been allocated but can no longer be used. If a memory leak grows too large, it can bring a program seeking more memory to a halt.

You should not attempt to free a block of memory that you have previously freed. The C++ Standard says the result of such an attempt is undefined, meaning that the consequences could be anything. Also you cannot use `delete` to free memory created by declaring ordinary variables:

```
int * ps = new int;   // ok
delete ps;            // ok
delete ps;            // not ok now
int jugs = 5;         // ok
int * pi = &jugs;     // ok
delete pi;            // not allowed, memory not allocated by new
```

Caution

You should use `delete` only to free memory allocated with `new`. However, it is safe to apply `delete` to a null pointer.

Note that the critical requirement for using `delete` is to use it with memory allocated by `new`. This doesn't mean you have to use the same pointer you used with `new`; instead, you have to use the same address:

```
int * ps = new int;   // allocate memory
int * pq = ps;        // set second pointer to same block
delete pq;            // delete with second pointer
```

Ordinarily, you won't create two pointers to the same block of memory because that raises the possibility that you will mistakenly try to delete the same block twice. But as you'll soon see, using a second pointer does make sense when you work with a function that returns a pointer.

Using `new` to Create Dynamic Arrays

If all a program needs is a single value, you might as well declare a simple variable because that is simpler, if less impressive, than using `new` and a pointer to manage a single small data object. More typically, you use `new` with larger chunks of data, such as arrays, strings, and structures. This is where `new` is useful. Suppose, for example, you're writing a program that might or might not need an array, depending on information given to the program while it is running. If you create an array by declaring it, the space is allocated when the program is compiled. Whether or not the program finally uses the array, the array is there, using up memory. Allocating the array during compile time is called *static binding*, meaning that the array is built in to the program at compile time. But with `new`, you can create an array during runtime if you need it and skip creating the array if you don't need it. Or you can select an array size after the program is running. This is called *dynamic binding*, meaning that the array is created while the program is running. Such an array is called a *dynamic array*. With static binding, you must specify the array size when you write the program. With dynamic binding, the program can decide on an array size while the program runs.

For now, we'll look at two basic matters concerning dynamic arrays: how to use C++'s `new` operator to create an array and how to use a pointer to access array elements.

Creating a Dynamic Array with `new`

It's easy to create a dynamic array in C++; you tell `new` the type of array element and number of elements you want. The syntax requires that you follow the type name with the number of elements, in brackets. For example, if you need an array of 10 `int`s, you use this:

```
int * psome = new int [10]; // get a block of 10 ints
```

The `new` operator returns the address of the first element of the block. In this example, that value is assigned to the pointer `psome`.

As always, you should balance the call to `new` with a call to `delete` when the program finishes using that block of memory. However, using `new` with brackets to create an array requires using an alternative form of `delete` when freeing the array:

```
delete [] psome;                  // free a dynamic array
```

The presence of the brackets tells the program that it should free the whole array, not just the element pointed to by the pointer. Note that the brackets are between `delete` and the pointer. If you use `new` without brackets, you should use `delete` without brackets. If you use `new` with brackets, you should use `delete` with brackets. Earlier versions of C++ might not recognize the bracket notation. For the ANSI/ISO Standard, however, the effect of mismatching `new` and `delete` forms is undefined, meaning that you can't rely on some particular behavior. Here's an example:

```
int * pt = new int;
short * ps = new short [500];
delete [] pt;  // effect is undefined, don't do it
delete ps;     // effect is undefined, don't do it
```

In short, you should observe these rules when you use `new` and `delete`:

- Don't use `delete` to free memory that `new` didn't allocate.
- Don't use `delete` to free the same block of memory twice in succession.
- Use `delete []` if you used `new []` to allocate an array.
- Use `delete` (no brackets) if you used `new` to allocate a single entity.
- It's safe to apply `delete` to the null pointer (nothing happens).

Now let's return to the dynamic array. Note that `psome` is a pointer to a single `int`, the first element of the block. It's your responsibility to keep track of how many elements are in the block. That is, because the compiler doesn't keep track of the fact that `psome` points to the first of 10 integers, you have to write your program so that it keeps track of the number of elements.

Actually, the program does keep track of the amount of memory allocated so that it can be correctly freed at a later time when you use the `delete []` operator. But that information isn't publicly available; you can't use the `sizeof` operator, for example, to find the number of bytes in a dynamically allocated array.

The general form for allocating and assigning memory for an array is this:

```
type_name * pointer_name = new type_name [num_elements];
```

Invoking the `new` operator secures a block of memory large enough to hold *`num_elements`* elements of type *`type_name`*, with *`pointer_name`* pointing to the first element. As you're about to see, you can use *`pointer_name`* in many of the same ways you can use an array name.

Using a Dynamic Array

After you create a dynamic array, how do you use it? First, think about the problem conceptually. The following statement creates a pointer, `psome`, that points to the first element of a block of 10 `int` values:

```
int * psome = new int [10]; // get a block of 10 ints
```

Think of it as a finger pointing to that element. Suppose an int occupies 4 bytes. Then, by moving your finger 4 bytes in the correct direction, you can point to the second element. Altogether, there are 10 elements, which is the range over which you can move your finger. Thus, the new statement supplies you with all the information you need to identify every element in the block.

Now think about the problem practically. How do you access one of these elements? The first element is no problem. Because psome points to the first element of the array, *psome is the value of the first element. That leaves nine more elements to access. The simplest way to access the elements may surprise you if you haven't worked with C: Just use the pointer as if it were an array name. That is, you can use psome[0] instead of *psome for the first element, psome[1] for the second element, and so on. It turns out to be very simple to use a pointer to access a dynamic array, even if it may not immediately be obvious why the method works. The reason you can do this is that C and C++ handle arrays internally by using pointers anyway. This near equivalence of arrays and pointers is one of the beauties of C and C++. (It's also sometimes a problem, but that's another story.) You'll learn more about this equivalence in a moment. First, Listing 4.18 shows how you can use new to create a dynamic array and then use array notation to access the elements. It also points out a fundamental difference between a pointer and a true array name.

Listing 4.18 **arraynew.cpp**

```
// arraynew.cpp -- using the new operator for arrays
#include <iostream>
int main()
{
    using namespace std;
    double * p3 = new double [3]; // space for 3 doubles
    p3[0] = 0.2;                  // treat p3 like an array name
    p3[1] = 0.5;
    p3[2] = 0.8;
    cout << "p3[1] is " << p3[1] << ".\n";
    p3 = p3 + 1;                  // increment the pointer
    cout << "Now p3[0] is " << p3[0] << " and ";
    cout << "p3[1] is " << p3[1] << ".\n";
    p3 = p3 - 1;                  // point back to beginning
    delete [] p3;                 // free the memory
    return 0;
}
```

Here is the output from the program in Listing 4.18:

```
p3[1] is 0.5.
Now p3[0] is 0.5 and p3[1] is 0.8.
```

As you can see, `arraynew.cpp` uses the pointer `p3` as if it were the name of an array, with `p3[0]` as the first element, and so on. The fundamental difference between an array name and a pointer appears in the following line:

```
p3 = p3 + 1; // okay for pointers, wrong for array names
```

You can't change the value of an array name. But a pointer is a variable, hence you can change its value. Note the effect of adding 1 to `p3`. The expression `p3[0]` now refers to the former second element of the array. Thus, adding 1 to `p3` causes it to point to the second element instead of the first. Subtracting one takes the pointer back to its original value so that the program can provide `delete []` with the correct address.

The actual addresses of consecutive `int`s typically differ by 2 or 4 bytes, so the fact that adding 1 to `p3` gives the address of the next element suggests that there is something special about pointer arithmetic. There is.

Pointers, Arrays, and Pointer Arithmetic

The near equivalence of pointers and array names stems from *pointer arithmetic* and how C++ handles arrays internally. First, let's check out the arithmetic. Adding one to an integer variable increases its value by one, but adding one to a pointer variable increases its value by the number of bytes of the type to which it points. Adding one to a pointer to `double` adds 8 to the numeric value on systems with 8-byte `double`, whereas adding one to a pointer-to-`short` adds two to the pointer value if `short` is 2 bytes. Listing 4.19 demonstrates this amazing point. It also shows a second important point: C++ interprets the array name as an address.

Listing 4.19 **addpntrs.cpp**

```
// addpntrs.cpp -- pointer addition
#include <iostream>
int main()
{
    using namespace std;
    double wages[3] = {10000.0, 20000.0, 30000.0};
    short stacks[3] = {3, 2, 1};

// Here are two ways to get the address of an array
    double * pw = wages;     // name of an array = address
    short * ps = &stacks[0]; // or use address operator
// with array element
    cout << "pw = " << pw << ", *pw = " << *pw << endl;
    pw = pw + 1;
    cout << "add 1 to the pw pointer:\n";
    cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";
```

```
    cout << "ps = " << ps << ", *ps = " << *ps << endl;
    ps = ps + 1;
    cout << "add 1 to the ps pointer:\n";
    cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";

    cout << "access two elements with array notation\n";
    cout << "stacks[0] = " << stacks[0]
         << ", stacks[1] = " << stacks[1] << endl;
    cout << "access two elements with pointer notation\n";
    cout << "*stacks = " << *stacks
         << ", *(stacks + 1) =  " << *(stacks + 1) << endl;

    cout << sizeof(wages) << " = size of wages array\n";
    cout << sizeof(pw) << " = size of pw pointer\n";
    return 0;
}
```

Here is the output from the program in Listing 4.19:

```
pw = 0x28ccf0, *pw = 10000
add 1 to the pw pointer:
pw = 0x28ccf8, *pw = 20000

ps = 0x28ccea, *ps = 3
add 1 to the ps pointer:
ps = 0x28ccec, *ps = 2

access two elements with array notation
stacks[0] = 3, stacks[1] = 2
access two elements with pointer notation
*stacks = 3, *(stacks + 1) =  2
24 = size of wages array
4 = size of pw pointer
```

Program Notes

In most contexts, C++ interprets the name of an array as the address of its first element. Thus, the following statement makes `pw` a pointer to type `double` and then initializes `pw` to `wages`, which is the address of the first element of the `wages` array:

```
double * pw = wages;
```

For `wages`, as with any array, we have the following equality:

```
wages = &wages[0] = address of first element of array
```

Just to show that this is no jive, the program explicitly uses the address operator in the expression `&stacks[0]` to initialize the `ps` pointer to the first element of the stacks array.

Next, the program inspects the values of pw and *pw. The first is an address, and the second is the value at that address. Because pw points to the first element, the value displayed for *pw is that of the first element, 10000. Then the program adds one to pw. As promised, this adds eight to the numeric address value because double on this system is 8 bytes. This makes pw equal to the address of the second element. Thus, *pw is now 20000, the value of the second element (see Figure 4.10). (The address values in the figure are adjusted to make the figure clearer.)

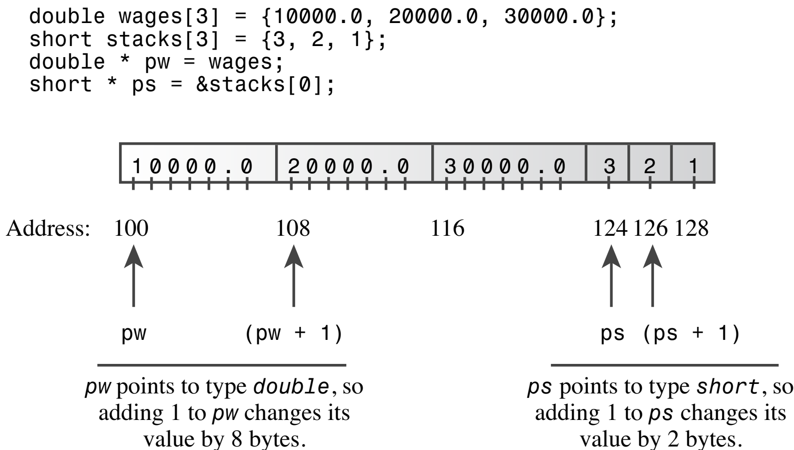

Figure 4.10 Pointer addition.

After this, the program goes through similar steps for ps. This time, because ps points to type short and because short is 2 bytes, adding 1 to the pointer increases its value by 2 (0x28ccea + 2 = 0x28ccec in hexadecimal). Again, the result is to make the pointer point to the next element of the array.

> **Note**
>
> Adding one to a pointer variable increases its value by the number of bytes of the type to which it points.

Now consider the array expression stacks[1]. The C++ compiler treats this expression exactly as if you wrote it as *(stacks + 1). The second expression means calculate the address of the second element of the array and then find the value stored there. The end result is precisely what stacks[1] means. (Operator precedence requires that you use the parentheses. Without them, 1 would be added to *stacks instead of to stacks.)

The program output demonstrates that *(stacks + 1) and stacks[1] are the same. Similarly, *(stacks + 2) is the same as stacks[2]. In general, wherever you use array notation, C++ makes the following conversion:

```
arrayname[i] becomes *(arrayname + i)
```

And if you use a pointer instead of an array name, C++ makes the same conversion:

```
pointername[i] becomes *(pointername + i)
```

Thus, in many respects you can use pointer names and array names in the same way. You can use the array brackets notation with either. You can apply the dereferencing operator (*) to either. In most expressions, each represents an address. One difference is that you can change the value of a pointer, whereas an array name is a constant:

```
pointername = pointername + 1; // valid
arrayname = arrayname + 1;     // not allowed
```

A second difference is that applying the `sizeof` operator to an array name yields the size of the array, but applying `sizeof` to a pointer yields the size of the pointer, even if the pointer points to the array. For example, in Listing 4.19, both `pw` and `wages` refer to the same array. But applying the `sizeof` operator to them produces the following results:

24 = size of `wages` array << displaying `sizeof wages`

4 = size of `pw` pointer << displaying `sizeof pw`

This is one case in which C++ doesn't interpret the array name as an address.

The Address of an Array

Taking the address of an array is another case in which the name of an array is not interpreted as its address. But wait, isn't the name of an array interpreted as the address of the array? Not quite—the name of the array is interpreted as the address of the first element of an array, whereas applying the address operator yields the address of the whole array:

```
short tell[10];        // tell an array of 20 bytes
cout << tell << endl;  // displays &tell[0]
cout << &tell << endl; // displays address of whole array
```

Numerically, these two addresses are the same, but conceptually `&tell[0]`, and hence `tell`, is the address of a 2-byte block of memory, whereas `&tell` is the address of a 20-byte block of memory. So the expression `tell + 1` adds `2` to the address value, whereas `&tell + 1` adds `20` to the address value. Another way of expressing this is to say that `tell` is type pointer-to-`short`, or `short *`, and `&tell` is type pointer-to-array-of-20-`short`s, or `short (*)[20]`.

Now you might be wondering about the genesis of that last type description. First, here is how you could declare and initialize a pointer of that type:

```
short (*pas)[20] = &tell;  // pas points to array of 20 shorts
```

If you omit the parentheses, precedence rules would first associate `[20]` with `pas`, making `pas` an array of 20 pointers-to-`short`, so the parentheses are necessary. Next, if you wish to describe the type of a variable, you can use the declaration of that variable as a guide and remove the variable name. Thus, the type of `pas` is `short (*)[20]`. Also note that because `pas` is set to `&tell`, `*pas` is equivalent to `tell`, so `(*pas)[0]` would be the first element of the `tell` array.

In short, using new to create an array and using a pointer to access the different elements is a simple matter. You just treat the pointer as an array name. Understanding why this works, however, is an interesting challenge. If you actually want to understand arrays and pointers, you should review their mutual relationships carefully.

Summarizing Pointer Points

You've been exposed to quite a bit of pointer knowledge lately, so let's summarize what's been revealed about pointers and arrays to date.

Declaring Pointers

To declare a pointer to a particular type, use this form:

```
typeName * pointerName;
```

Here are some examples:

```
double * pn;      // pn can point to a double value
char * pc;        // pc can point to a char value
```

Here pn and pc are pointers, and double * and char * are the C++ notations for the types pointer-to-double and pointer-to-char.

Assigning Values to Pointers

You should assign a memory address to a pointer. You can apply the & operator to a variable name to get an address of named memory, and the new operator returns the address of unnamed memory.

Here are some examples:

```
double * pn;          // pn can point to a double value
double * pa;          // so can pa
char * pc;            // pc can point to a char value
double bubble = 3.2;
pn = &bubble;         // assign address of bubble to pn
pc = new char;        // assign address of newly allocated char memory to pc
pa = new double[30];  // assign address of 1st element of array of 30 double to pa
```

Dereferencing Pointers

Dereferencing a pointer means referring to the pointed-to value. You apply the dereferencing, or indirect value, operator (*) to a pointer to dereference it. Thus, if pn is a pointer to bubble, as in the preceding example, then *pn is the pointed-to value, or 3.2, in this case.

Here are some examples:

```
cout << *pn; // print the value of bubble
*pc = 'S';   // place 'S' into the memory location whose address is pc
```

Array notation is a second way to dereference a pointer; for instance, pn[0] is the same as *pn. You should never dereference a pointer that has not been initialized to a proper address.

Distinguishing Between a Pointer and the Pointed-to Value

Remember, if pt is a pointer-to-int, *pt is not a pointer-to-int; instead, *pt is the complete equivalent to a type int variable. It is pt that is the pointer.

Here are some examples:

```
int * pt = new int;      // assigns an address to the pointer pt
*pt = 5;                 // stores the value 5 at that address
```

Array Names

In most contexts, C++ treats the name of an array as equivalent to the address of the first element of an array.

Here is an example:

```
int tacos[10];           // now tacos is the same as &tacos[0]
```

One exception is when you use the name of an array with the sizeof operator. In that case, sizeof returns the size of the entire array, in bytes.

Pointer Arithmetic

C++ allows you to add an integer to a pointer. The result of adding one equals the original address value plus a value equal to the number of bytes in the pointed-to object. You can also subtract an integer from a pointer to take the difference between two pointers. The last operation, which yields an integer, is meaningful only if the two pointers point into the same array (pointing to one position past the end is allowed, too); it then yields the separation between the two elements.

Here are some examples:

```
int tacos[10] = {5,2,8,4,1,2,2,4,6,8};
int * pt = tacos;        // suppose pf and tacos are the address 3000
pt = pt + 1;             // now pt is 3004 if a int is 4 bytes
int *pe = &tacos[9];     // pe is 3036 if an int is 4 bytes
pe = pe - 1;             // now pe is 3032, the address of tacos[8]
int diff = pe - pt;      // diff is 7, the separation between
                         // tacos[8] and tacos[1]
```

Dynamic Binding and Static Binding for Arrays

You can use an array declaration to create an array with static binding—that is, an array whose size is set during the compilation process:

```
int tacos[10]; // static binding, size fixed at compile time
```

You use the new [] operator to create an array with dynamic binding (a dynamic array)—that is, an array that is allocated and whose size can be set during runtime. You free the memory with delete [] when you are done:

```
int size;
cin >> size;
int * pz = new int [size];   // dynamic binding, size set at run time
...
delete [] pz;                // free memory when finished
```

Array Notation and Pointer Notation

Using bracket array notation is equivalent to dereferencing a pointer:

```
tacos[0] means *tacos means the value at address tacos
tacos[3] means *(tacos + 3) means the value at address tacos + 3
```

This is true for both array names and pointer variables, so you can use either pointer notation or array notation with pointers and array names.

Here are some examples:

```
int * pt = new int [10];   // pt points to block of 10 ints
*pt = 5;                   // set element number 0 to 5
pt[0] = 6;                 // reset element number 0 to 6
pt[9] = 44;                // set tenth element (element number 9) to 44
int coats[10];
*(coats + 4) = 12;         // set coats[4] to 12
```

Pointers and Strings

The special relationship between arrays and pointers extends to C-style strings. Consider the following code:

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

The name of an array is the address of its first element, so flower in the cout statement is the address of the char element containing the character r. The cout object assumes that the address of a char is the address of a string, so it prints the character at that address and then continues printing characters until it runs into the null character (\0). In short, if you give cout the address of a character, it prints everything from that character to the first null character that follows it.

The crucial element here is not that flower is an array name but that flower acts as the address of a char. This implies that you can use a pointer-to-char variable as an argument to cout also because it, too, is the address of a char. Of course, that pointer should point to the beginning of a string. We'll check that out in a moment.

But what about the final part of the preceding cout statement? If flower is actually the address of the first character of a string, what is the expression "s are red\n"? To be consistent with cout's handling of string output, this quoted string should also be an

address. And it is, for in C++ a quoted string, like an array name, serves as the address of its first element. The preceding code doesn't really send a whole string to `cout`; it just sends the string address. This means strings in an array, quoted string constants, and strings described by pointers are all handled equivalently. Each is really passed along as an address. That's certainly less work than passing each and every character in a string.

> **Note**
>
> With `cout` and with most C++ expressions, the name of an array of `char`, a pointer-to-`char`, and a quoted string constant are all interpreted as the address of the first character of a string.

Listing 4.20 illustrates the use of the different forms of strings. It uses two functions from the string library. The `strlen()` function, which you've used before, returns the length of a string. The `strcpy()` function copies a string from one location to another. Both have function prototypes in the `cstring` header file (or `string.h`, on less up-to-date implementations). The program also uses comments to showcase some pointer misuses that you should try to avoid.

Listing 4.20 **ptrstr.cpp**

```
// ptrstr.cpp -- using pointers to strings
#include <iostream>
#include <cstring>              // declare strlen(), strcpy()
int main()
{
    using namespace std;
    char animal[20] = "bear";   // animal holds bear
    const char * bird = "wren"; // bird holds address of string
    char * ps;                  // uninitialized

    cout << animal << " and ";  // display bear
    cout << bird << "\n";       // display wren
    // cout << ps << "\n";      //may display garbage, may cause a crash

    cout << "Enter a kind of animal: ";
    cin >> animal;              // ok if input < 20 chars
    // cin >> ps; Too horrible a blunder to try; ps doesn't
    //            point to allocated space

    ps = animal;                // set ps to point to string
    cout << ps << "!\n";       // ok, same as using animal
    cout << "Before using strcpy():\n";
    cout << animal << " at " << (int *) animal << endl;
    cout << ps << " at " << (int *) ps << endl;

    ps = new char[strlen(animal) + 1];  // get new storage
    strcpy(ps, animal);         // copy string to new storage
```

```
    cout << "After using strcpy():\n";
    cout << animal << " at " << (int *) animal << endl;
    cout << ps << " at " << (int *) ps << endl;
    delete [] ps;
    return 0;
}
```

Here is a sample run of the program in Listing 4.20:

```
bear and wren
Enter a kind of animal: fox
fox!
Before using strcpy():
fox at 0x0065fd30
fox at 0x0065fd30
After using strcpy():
fox at 0x0065fd30
fox at 0x004301c8
```

Program Notes

The program in Listing 4.20 creates one `char` array (`animal`) and two pointers-to-`char` variables (`bird` and `ps`). The program begins by initializing the `animal` array to the `"bear"` string, just as you've initialized arrays before. Then, the program does something new. It initializes a pointer-to-`char` to a string:

```
const char * bird = "wren"; // bird holds address of string
```

Remember, `"wren"` actually represents the address of the string, so this statement assigns the address of `"wren"` to the `bird` pointer. (Typically, a compiler sets aside an area in memory to hold all the quoted strings used in the program source code, associating each stored string with its address.) This means you can use the pointer `bird` just as you would use the string `"wren"`, as in this example:

```
cout << "A concerned " << bird << " speaks\n";
```

String literals are constants, which is why the code uses the `const` keyword in the declaration. Using `const` in this fashion means you can use `bird` to access the string but not to change it. Chapter 7 takes up the topic of `const` pointers in greater detail. Finally, the pointer `ps` remains uninitialized, so it doesn't point to any string. (As you know, that is usually a bad idea, and this example is no exception.)

Next, the program illustrates that you can use the array name `animal` and the pointer `bird` equivalently with `cout`. Both, after all, are the addresses of strings, and `cout` displays the two strings (`"bear"` and `"wren"`) stored at those addresses. If you activate the code that makes the error of attempting to display `ps`, you might get a blank line, you might get garbage displayed, and you might get a program crash. Creating an uninitialized pointer is a bit like distributing a blank signed check: You lack control over how it will be used.

For input, the situation is a bit different. It's safe to use the array `animal` for input as long as the input is short enough to fit into the array. It would not be proper to use `bird` for input, however:

- Some compilers treat string literals as read-only constants, leading to a runtime error if you try to write new data over them. That string literals be constants is the mandated behavior in C++, but not all compilers have made that change from older behavior yet.
- Some compilers use just one copy of a string literal to represent all occurrences of that literal in a program.

Let's amplify the second point. C++ doesn't guarantee that string literals are stored uniquely. That is, if you use a string literal `"wren"` several times in a program, the compiler might store several copies of the string or just one copy. If it does the latter, then setting `bird` to point to one `"wren"` makes it point to the only copy of that string. Reading a value into one string could affect what you thought was an independent string elsewhere. In any case, because the `bird` pointer is declared as `const`, the compiler prevents any attempt to change the contents of the location pointed to by `bird`.

Worse yet is trying to read information into the location to which `ps` points. Because `ps` is not initialized, you don't know where the information will wind up. It might even overwrite information that is already in memory. Fortunately, it's easy to avoid these problems: You just use a sufficiently large `char` array to receive input and don't use string constants to receive input or uninitialized pointers to receive input. (Or you can sidestep all these issues and use `std::string` objects instead of arrays.)

> **Caution**
>
> When you read a string into a program-style string, you should always use the address of previously allocated memory. This address can be in the form of an array name or of a pointer that has been initialized using `new`.

Next, notice what the following code accomplishes:

```
ps = animal;                // set ps to point to string
...
cout << animal << " at " << (int *) animal << endl;
cout << ps << " at " << (int *) ps << endl;
```

It produces the following output:

```
fox at 0x0065fd30
fox at 0x0065fd30
```

Normally, if you give `cout` a pointer, it prints an address. But if the pointer is type `char *`, `cout` displays the pointed-to string. If you want to see the address of the string, you have to type cast the pointer to another pointer type, such as `int *`, which this code does.

So ps displays as the string "fox", but (int *) ps displays as the address where the string is found. Note that assigning animal to ps does not copy the string; it copies the address. This results in two pointers (animal and ps) to the same memory location and string.

To get a copy of a string, you need to do more. First, you need to allocate memory to hold the string. You can do this by declaring a second array or by using new. The second approach enables you to custom fit the storage to the string:

```
ps = new char[strlen(animal) + 1]; // get new storage
```

The string "fox" doesn't completely fill the animal array, so this approach wastes space. This bit of code uses strlen() to find the length of the string; it adds one to get the length, including the null character. Then the program uses new to allocate just enough space to hold the string.

Next, you need a way to copy a string from the animal array to the newly allocated space. It doesn't work to assign animal to ps because that just changes the address stored in ps and thus loses the only way the program had to access the newly allocated memory. Instead, you need to use the strcpy() library function:

```
strcpy(ps, animal);                // copy string to new storage
```

The strcpy() function takes two arguments. The first is the destination address, and the second is the address of the string to be copied. It's up to you to make certain that the destination really is allocated and has sufficient space to hold the copy. That's accomplished here by using strlen() to find the correct size and using new to get free memory.

Note that by using strcpy() and new, you get two separate copies of "fox":

```
fox at 0x0065fd30
fox at 0x004301c8
```

Also note that new located the new storage at a memory location quite distant from that of the array animal.

Often you encounter the need to place a string into an array. You use the = operator when you initialize an array; otherwise, you use strcpy() or strncpy(). You've seen the strcpy() function; it works like this:

```
char food[20] = "carrots"; // initialization
strcpy(food, "flan");      // otherwise
```

Note that something like the following can cause problems because the food array is smaller than the string:

```
strcpy(food, "a picnic basket filled with many goodies");
```

In this case, the function copies the rest of the string into the memory bytes immediately following the array, which can overwrite other memory the program is using. To avoid that problem, you should use strncpy() instead. It takes a third argument: the maximum number of characters to be copied. Be aware, however, that if this function runs out

of space before it reaches the end of the string, it doesn't add the null character. Thus, you should use the function like this:

```
strncpy(food, "a picnic basket filled with many goodies", 19);
food[19] = '\0';
```

This copies up to 19 characters into the array and then sets the last element to the null character. If the string is shorter than 19 characters, `strncpy()` adds a null character earlier to mark the true end of the string.

> **Caution**
>
> Use `strcpy()` or `strncpy()`, not the assignment operator, to assign a string to an array.

Now that you've seen some aspects of using C-style strings and the `cstring` library, you can appreciate the comparative simplicity of using the C++ `string` type. You (normally) don't have to worry about a string overflowing an array, and you can use the assignment operator instead of `strcpy()` or `strncpy()`.

Using `new` to Create Dynamic Structures

You've seen how it can be advantageous to create arrays during runtime rather than at compile time. The same holds true for structures. You need to allocate space for only as many structures as a program needs during a particular run. Again, the `new` operator is the tool to use. With it, you can create dynamic structures. Again, *dynamic* means the memory is allocated during runtime, not at compile time. Incidentally, because classes are much like structures, you are able to use the techniques you'll learn in this section for structures with classes, too.

Using `new` with structures has two parts: creating the structure and accessing its members. To create a structure, you use the structure type with `new`. For example, to create an unnamed structure of the `inflatable` type and assign its address to a suitable pointer, you can use the following:

```
inflatable * ps = new inflatable;
```

This assigns to `ps` the address of a chunk of free memory large enough to hold a structure of the `inflatable` type. Note that the syntax is exactly the same as it is for C++'s built-in types.

The tricky part is accessing members. When you create a dynamic structure, you can't use the dot membership operator with the structure name because the structure has no name. All you have is its address. C++ provides an operator just for this situation: the arrow membership operator (`->`). This operator, formed by typing a hyphen and then a greater-than symbol, does for pointers to structures what the dot operator does for structure names. For example, if `ps` points to a type `inflatable` structure, then `ps->price` is the `price` member of the pointed-to structure (see Figure 4.11).

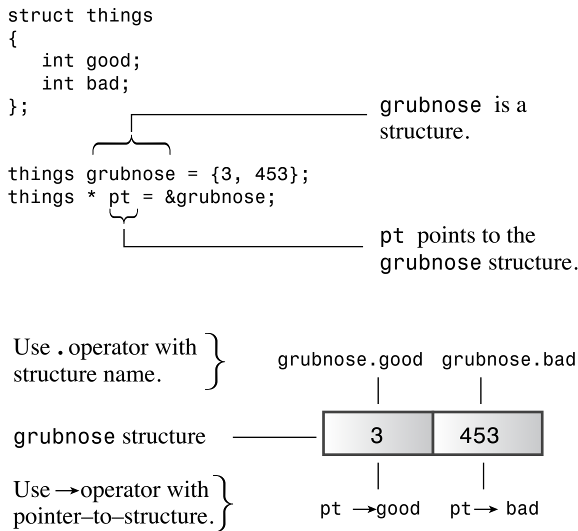

Figure 4.11 Identifying structure members.

Tip

Sometimes new C++ users become confused about when to use the dot operator and when to use the arrow operator to specify a structure member. The rule is simple: If the structure identifier is the name of a structure, use the dot operator. If the identifier is a pointer to the structure, use the arrow operator.

A second, uglier approach to accessing structure members is to realize that if `ps` is a pointer to a structure, then `*ps` represents the pointed-to value—the structure itself. Then, because `*ps` is a structure, `(*ps).price` is the `price` member of the structure. C++'s operator precedence rules require that you use parentheses in this construction.

Listing 4.21 uses `new` to create an unnamed structure and demonstrates both pointer notations for accessing structure members.

Listing 4.21 **newstrct.cpp**

```
// newstrct.cpp -- using new with a structure
#include <iostream>
struct inflatable   // structure definition
{
    char name[20];
    float volume;
    double price;
};
int main()
{
    using namespace std;
```

```
    inflatable * ps = new inflatable; // allot memory for structure
    cout << "Enter name of inflatable item: ";
    cin.get(ps->name, 20);            // method 1 for member access
    cout << "Enter volume in cubic feet: ";
    cin >> (*ps).volume;              // method 2 for member access
    cout << "Enter price: $";
    cin >> ps->price;
    cout << "Name: " << (*ps).name << endl;              // method 2
    cout << "Volume: " << ps->volume << " cubic feet\n"; // method 1
    cout << "Price: $" << ps->price << endl;             // method 1
    delete ps;                        // free memory used by structure
    return 0;
}
```

Here is a sample run of the program in Listing 4.21:

```
Enter name of inflatable item: Fabulous Frodo
Enter volume in cubic feet: 1.4
Enter price: $27.99
Name: Fabulous Frodo
Volume: 1.4 cubic feet
Price: $27.99
```

An Example of Using `new` and `delete`

Let's look at an example that uses `new` and `delete` to manage storing string input from the keyboard. Listing 4.22 defines a function `getname()` that returns a pointer to an input string. This function reads the input into a large temporary array and then uses `new []` with an appropriate size to create a chunk of memory sized to fit to the input string. Then the function returns the pointer to the block. This approach could conserve a lot of memory for programs that read in a large number of strings. (In real life, where many of us live, it would be easier to use the `string` class, which has the use of `new` and `delete` built in to its design.)

Suppose your program has to read 1,000 strings and that the largest string might be 79 characters long, but most of the strings are much shorter than that. If you used `char` arrays to hold the strings, you'd need 1,000 arrays of 80 characters each. That's 80,000 bytes, and much of that block of memory would wind up being unused. Alternatively, you could create an array of 1,000 pointers to `char` and then use `new` to allocate only the amount of memory needed for each string. That could save tens of thousands of bytes. Instead of having to use a large array for every string, you fit the memory to the input. Even better, you could also use `new` to find space to store only as many pointers as needed. Well, that's a little too ambitious for right now. Even using an array of 1,000 pointers is a little too ambitious for right now, but Listing 4.22 illustrates some of the technique. Also just to illustrate how `delete` works, the program uses it to free memory for reuse.

Listing 4.22 **delete.cpp**

```
// delete.cpp -- using the delete operator
#include <iostream>
#include <cstring>      // or string.h
using namespace std;
char * getname(void);   // function prototype
int main()
{
    char * name;        // create pointer but no storage

    name = getname();   // assign address of string to name
    cout << name << " at " << (int *) name << "\n";
    delete [] name;     // memory freed

    name = getname();   // reuse freed memory
    cout << name << " at " << (int *) name << "\n";
    delete [] name;     // memory freed again
    return 0;
}

char * getname()        // return pointer to new string
{
    char temp[80];      // temporary storage
    cout << "Enter last name: ";
    cin >> temp;
    char * pn = new char[strlen(temp) + 1];
    strcpy(pn, temp);   // copy string into smaller space

    return pn;          // temp lost when function ends
}
```

Here is a sample run of the program in Listing 4.22:

```
Enter last name: Fredeldumpkin
Fredeldumpkin at 0x004326b8
Enter last name: Pook
Pook at 0x004301c8
```

Program Notes

Consider the function `getname()` in the program in Listing 4.22. It uses `cin` to place an input word into the `temp` array. Next, it uses `new` to allocate new memory to hold the word. Including the null character, the program needs `strlen(temp) + 1` characters to store the string, so that's the value given to `new`. After the space becomes available, `getname()` uses the standard library function `strcpy()` to copy the string from `temp` to the new block. The function doesn't check to see whether the string fits, but `getname()`

covers that by requesting the right number of bytes with `new`. Finally, the function returns `pn`, the address of the string copy.

In `main()`, the return value (the address) is assigned to the pointer `name`. This pointer is defined in `main()`, but it points to the block of memory allocated in the `getname()` function. The program then prints the string and the address of the string.

Next, after it frees the block pointed to by `name`, `main()` calls `getname()` a second time. C++ doesn't guarantee that newly freed memory is the first to be chosen the next time `new` is used, and in this sample run, it isn't.

Note in this example that `getname()` allocates memory and `main()` frees it. It's usually not a good idea to put `new` and `delete` in separate functions because that makes it easier to forget to use `delete`. But this example does separate `new` from `delete` just to show that it is possible.

To appreciate some of the more subtle aspects of this program, you should know a little more about how C++ handles memory. So let's preview some material that's covered more fully in Chapter 9.

Automatic Storage, Static Storage, and Dynamic Storage

C++ has three ways of managing memory for data, depending on the method used to allocate memory: automatic storage, static storage, and dynamic storage, sometimes called the *free store* or *heap*. Data objects allocated in these three ways differ from each other in how long they remain in existence. We'll take a quick look at each type. (C++11 adds a fourth form called *thread storage* that we'll discuss briefly in Chapter 9.)

Automatic Storage

Ordinary variables defined inside a function use *automatic storage* and are called *automatic variables*. These terms mean that the variables come into existence automatically when the function containing them is invoked, and they expire when the function terminates. For example, the `temp` array in Listing 4.22 exists only while the `getname()` function is active. When program control returns to `main()`, the memory used for `temp` is freed automatically. If `getname()` returned the address of `temp`, the `name` pointer in `main()` would be left pointing to a memory location that would soon be reused. That's one reason you have to use `new` in `getname()`. Actually, automatic values are local to the block that contains them. A *block* is a section of code enclosed between braces. So far, all our blocks have been entire functions. But as you'll see in the next chapter, you can have blocks within a function. If you define a variable inside one of those blocks, it exists only while the program is executing statements inside the block.

Automatic variables typically are stored on a *stack*. This means that when program execution enters a block of code, its variables are added consecutively to the stack in memory and then are freed in reverse order when execution leaves the block. (This is called a *last-in, first-out*, or *LIFO*, process.) So the stack grows and shrinks as execution proceeds.

Static Storage

Static storage is storage that exists throughout the execution of an entire program. There are two ways to make a variable static. One is to define it externally, outside a function. The other is to use the keyword `static` when declaring a variable:

```
static double fee = 56.50;
```

Under K&R C, you can initialize only static arrays and structures, whereas C++ Release 2.0 (and later) and ANSI C allow you to initialize automatic arrays and structures, too. However, as you may have discovered, some C++ implementations do not yet implement initialization for automatic arrays and structures.

Chapter 9 discusses static storage in more detail. The main point you should note now about automatic and static storage is that these methods rigidly define the lifetime of a variable. Either the variable exists for the entire duration of a program (a static variable) or it exists only while a particular function is being executed (an automatic variable).

Dynamic Storage

The `new` and `delete` operators provide a more flexible approach than automatic and static variables. They manage a pool of memory, which C++ refers to as the *free store* or *heap*. This pool is separate from the memory used for static and automatic variables. As Listing 4.22 shows, `new` and `delete` enable you to allocate memory in one function and free it in another. Thus, the lifetime of the data is not tied arbitrarily to the life of the program or the life of a function. Using `new` and `delete` together gives you much more control over how a program uses memory than does using ordinary variables. However, memory management becomes more complex. In a stack, the automatic addition and removal mechanism results in the part of memory in use always being contiguous. But the interplay between `new` and `delete` can leave holes in the free store, making keeping track of where to allocate new memory requests more difficult.

Stacks, Heaps, and Memory Leaks

What happens if you *don't* call `delete` after creating a variable on the free store (or heap) with the `new` operator? The variable or construct dynamically allocated on the free store continues to persist if `delete` is not called, even though the memory that contains the pointer has been freed due to rules of scope and object lifetime. In essence, you have no way to access the construct on the free store because the pointer to the memory that contains it is gone. You have now created a *memory leak*. Memory that has been leaked remains unusable through the life of the program; it's been allocated but can't be deallocated. In extreme (though not uncommon) cases, memory leaks can be so severe that they use up all the memory available to the application, causing it to crash with an out-of-memory error. In addition, these leaks may negatively affect some operating systems or other applications running in the same memory space, causing them, in turn, to fail.

Even the best programmers and software companies create memory leaks. To avoid them, it's best to get into the habit of joining your `new` and `delete` operators immediately, planning for and entering the deletion of your construct as soon as you dynamically allocate it on the free store. C++'s smart pointers (Chapter 16) help automate the task.

> **Note**
>
> Pointers are among the most powerful of C++ tools. They are also the most dangerous because they permit computer-unfriendly actions, such as using an uninitialized pointer to access memory or attempting to free the same memory block twice. Furthermore, until you get used to pointer notation and pointer concepts through practice, pointers can be confusing. Because pointers are an important part of C++ programming, they weave in and out of future discussions in this book. This book discusses pointers several more times. The hope is that each exposure will make you more comfortable with them.

Combinations of Types

This chapter has introduced arrays, structures, and pointers. These can be combined in various ways, so let's review some of the possibilities, starting with a structure:

```
struct antarctica_years_end
{
    int year;
 /* some really interesting data, etc. */
};
```

We can create variables of this type:

```
antarctica_years_end s01, s02, s03;  // s01, s02, s03 are structures
```

We can then access members using the membership operator:

```
s01.year = 1998;
```

We can create a pointer to such a structure:

```
antarctica_years_end * pa = &s02;
```

Provided the pointer has been set to a valid address, we then can use the indirect membership operator to access members:

```
pa->year = 1999;
```

We can create arrays of structures:

```
antarctica_years_end trio[3]; // array of 3 structures
```

We then can use the membership operator to access members of an element:

```
trio[0].year = 2003;  // trio[0] is a structure
```

Here, `trio` is an array, but `trio[0]` is a structure, and `trio[0].year` is a member of that structure. Because an array name is a pointer, we also can use the indirect membership operator:

```
(trio+1)->year = 2004; // same as trio[1].year = 2004;
```

We can create an array of pointers:

```
const antarctica_years_end * arp[3] = {&s01, &s02, &s03};
```