Chapter 3

Pointers and References

Powerful object-oriented mechanisms, such as run-time selection of overridden methods, are in large part only possible because of the indirection provided through the use of pointer values. It is often said that the language Java has no pointers. This is true in a superficial sense, in that the Java programmer seldom thinks about explicit pointer values. But lurking just below the surface of the language, almost everything in Java is represented internally by pointer values. The language C++ in general hides much less from the programmers view, and thus the use of pointers in C++ programs is explicit and direct. C++ provides a variety of different mechanisms for manipulating pointer values, and an explanation of these will be the topic of this chapter. The use of pointers and object-oriented programming will be explained in Chapters 4 and 5.

3.1 Java Really Does Have Pointers

We begin by explaining the assertion that Java really does have pointer values lurking just below the surface of the language. Remember the box class definition described in Section 2.7:

```java
class box -  //1pt  Java box
    public int value;
```

```java
box a = new box();
a.value = 7;  //1pt  set variable a
```

The declaration of a variable, such as the variable a, results in space being set aside and labeled with the name a. This space is only large enough to hold a pointer value. The value
we normally think of as being the quantity held by a is not allocated until the new operation is performed. This situation can be pictured as follows:

```
  a
  \---
     new box()
```

A second variable can be declared and assigned the value held by a:

```cpp
box b;
b = a;
```

Internally, the locations a and b are distinct, but the memory values they reference are the same. This can be pictured as follows:

```
  a
  \---
     --

  b
  \---
```

It is because internally they reference the same value that changes to either a or b will be reflected in the other variable.

Rather than being implicitly hidden from the programmer, pointers in C++ are explicit and must be directly manipulated in code.

### 3.2 Pointers on Pointers

A *pointer* is simply a variable that maintains as value the address of another location in memory. Because memory addresses have a fixed limit, the amount of storage necessary to hold a pointer can be determined at compile time, even if the size or extent of the object to which it will point is not known.

```
p
  \---
     *p
```

**Note**

Avoid confusing *the pointer itself* and *the object the pointer references*.

The pointer itself will usually reside in a local variable, and the value the pointer refers to will often be a dynamically allocated heap value. However, neither of these are guaranteed. We will discuss the distinction between heap and stack values in more detail in Chapter 4.
3.2. POINTERS ON POINTERS

Pointer values are used whenever there is a need for one or more levels of indirection at run time. Among the reasons for using pointer values are the following:

- A single pointer variable must reference a variety of different values over the course of execution, or
- A pointer will reference only a single value, but the particular value it will reference cannot be known at run time, or
- The amount of memory necessary to hold a value cannot be determined at compile time, and must be allocated a run-time.

A null pointer is a value that does not reference any other memory location, and should not be considered to point to any valid object. A pointer variable can always be assigned the numeric value zero in order to make it null, although the actual internal value used to represent the null pointer can differ from one platform to another. A pointer can also be tested for equality to the value zero, in order to determine whether or not it represents a null pointer. A pointer that is not equivalent to a null pointer on such a test is said to be non-null.

There are four principle mechanisms used to access the values denoted by a pointer, which depend in part on the type of value the pointer references:

- A pointer can be explicitly dereferenced using the unary * operator. If p is a variable holding a pointer to a value of some type, then *p is the value addressed by the pointer. We will discuss pointers to simple values in Section 3.4.

- A pointer to a structure, or class, can combine pointer dereferencing and member field extraction using the pointer operator. If p is a pointer to a value of a class type that contains a member field x, then p->x is the same as (*p).x. We will discuss the use of pointers and structures in Section 3.5.

- A pointer variable can be subscripted. This is useful only if the pointer addresses an array of objects. The subscript index is used to determine the element accessed by the expression. The connection between pointers and arrays is discussed in Section 3.6.

- An integer value can be added to or subtracted from a pointer in order to yield a new pointer. It is assumed (but not verified) that the pointer references an array of values. Again, this will be explored when we discuss pointers and arrays in Section 3.6.

A pointer can be distinguished from a reference, which is an internal pointer, closer to the way a Java value is represented internally. References will be discussed in Section 3.7.

The manipulation of pointers can be tricky, and is one of the most common sources of programming errors. For this reason it is important to fully understand the use and manipulation of pointers.
3.3 The address-of operator

Pointers can be used to reference either newly allocated memory, created using the new operator, or to reference other memory values. In the latter case the ampersand is used as the address-of operator. It determines the address in memory for a variable argument. This address can then be assigned to a pointer variable. The following illustrates the use of this function with the routine scanf, which reads text from the standard input, then parses the text and converts it into a variety of different values.\(^1\) For real and integer conversions the scanf function requires a pointer to the location where the value will be stored.

```c
int i; // 1pt location for final value
int *p; // 1pt pointer variable
p = &i; // 1pt set p to point to i
scanf("%d", p); // 1pt scan number into i
```

Often the address operator is applied directly in the argument, which avoids the need to explicitly declare a pointer variable. The example above could be written more directly as follows:

```c
int i; // 1pt location for final value
scanf("%d", &i); // 1pt scan number into i
```

Like all operators in C++, the address operator can be overloaded to provide a new meaning when used with an object value. This issue will be considered in detail in Chapter 7.

3.4 Pointers to Simple Values

When a pointer is referencing a primitive data type, such as an integer or a real, there are only two major operations that can be applied to the pointer value. The first is to compare the pointer value to another pointer. The second operation is to dereference the pointer value. The dereference results in a memory location, which can then be used as the target for an assignment:

```c
int i = 7;
int j = 11;
```

\(^1\)Although scanf is an excellent illustration of the use of pointers as arguments, the use of the function itself is now discouraged. The function is part of the legacy inherited from the earlier language C, and the functionality it provided is now available in a more robust form by the stream input library described in Chapter 10.
3.4. POINTERS TO SIMPLE VALUES

```c
int *p = &i; //-1pt set p to point to i

*p = *p + 3; //-1pt i now has the value 10
```

Pointers should be compared only for equality, to test whether two pointers refer to the same memory value. Ordering tests are legal, but are generally not what the programmer intends. One pointer is considered less than another if the value it references is found in memory at a smaller address. Since the exact memory locations is inherently platform specific, such a test has limited utility.

There is a difference between modifying a pointer value, and modifying the value that a pointer refers to. The above example changes the value that p references, namely, the variable i. The following changes p itself so that it will point to the variable j:

```c
p = &j; //-1pt change p to point to j
```

With all pointer values it is important to remember that the pointer itself and the value it references are separate, and may be managed independently of each other. In particular, the programmer should be careful to ensure that whenever a pointer is used the value it references will still exist. Executing the function Set followed by the function Use in the next example is an almost certain recipe for disaster, as the value the pointer refers to will have been overwritten by the time it is used:

```c
int *p; //-1pt global pointer variable

void Set ()
-
int i; //-1pt local variable
i = 7; //-1pt give i a value
p = &i; //-1pt set p to point to it
```

```c
void Use ()
-
double d;
d = 3.0;
d += *p; //-1pt use the value p points to
```

**Rule**

Pointers should only be compared for equality

**Warning**

Nothing prevents a pointer from referencing a deleted value
3.4.1 Pointers to Pointer

A pointer to a value that is itself a pointer is declared using multiple levels of star symbols. A common example is the declaration of the argument values supplied to the main procedure under Unix and some other operating systems. The main procedure is provided with two arguments. The first is an integer value that contains the number of elements in the second argument. The second argument is an array of string values. Given the close relationship between arrays and pointers (see Section 3.6) and the fact that primitive strings are represented as pointers to character (see Chapter 8), this second argument is often declared as a pointer to a pointer to a character:\(^2\)

\[
\text{int main (int argc, char ** argv)}
\]

\[
...
\]

\[
\text{cout \ll \"name of program\ll \ll \star\star \text{argv} \ll \backslash n;}
\]

\[
\text{return 0;}
\]

The first character in the name of the program can be printed, as shown, by finding the value of the pointer argv references, then finding the value of the first character this element points to.

3.4.2 Pointers and const

**Warning**

* A constant pointer is different from a pointer to const

The placement of the modifier const indicates whether it is the pointer itself or the value it points to that is constant:

\[
\text{int i = 7;}
\]

\[
\text{const int * p = \&i; //\-1pt pointer to a constant}
\]

\[
\text{int * const q = \&i; //\-1pt constant pointer}
\]

\[
*p = 8; //\-1pt not allowed, p points to a const
\]

\[
*q = 8; //\-1pt allowed, q is pointing to non const
\]

\[
p = q; //\-1pt allowed, p itself is not constant
\]

\[
q = p; //\-1pt not allowed q is constant
\]

To create a pointer that cannot be changed, and that references a value that itself cannot be changed, two const modifiers must be used:

\(^2\)It is just as often declared as an array of pointers to character values. An example in this form was shown in Section 2.8. The relationship between pointers and arrays will be explored in Section 3.6.
3.4. POINTERS TO SIMPLE VALUES

const int * const r = & i;

3.4.3 void * pointers

The data type void * (called either pointer to void or void-star pointer) is a special declaration type in C++ that can be used to hold any type of pointer value. It is used in much the same fashion as the data type Object in Java. That is, a pointer to void is a universal type, a type that can reference anything. Any pointer type can be converted into a void pointer:

```c
    double d;
    double * dp = & d;

    void * p = dp;
```

Just as with Object values in Java, converting from a void-star pointer back into the original pointer type requires a cast:

```c
    double * dp2;
    dp2 = (double *) p; //pt convert p back into pointer to double
```

If the security of the conversion cannot be easily verified, the safer `dynamic_cast` operator should be used (see Section 6.3). (Note that conversions of pointers is one situation where it is not possible to use the `type(value)` syntax for casts).

3.4.4 Pointers to Functions

It is possible to form a pointer whose value is a function. The syntax used for describing such a value is, however, rather unintuitive. The following declares a function that takes two integer values as argument and returns a floating point result. This is followed by the declaration of a global variable that illustrates the form used to declare function pointers:

```c
    double fdiv (int i, int j) return i / (double) j;
```

```c
    double (*fptr) (int, int); //pt declare variable fptr
    fptr = & fdiv; //pt assign value
```

The declaration of `fptr` asserts that it is a pointer to a function that requires two integer arguments and returns a floating point value. The variable is then set by the assignment statement to point to the function `fdiv`. To call such a value the pointer dereference need not be written, although no harm results from being precise:

```c
    double (i, j) = fdiv (i, j); // call function
```
double x = fptr(7, 14); /*-1pt call fptr directly
double x = (*fptr)(7, 14); /*-1pt dereference fptr and call

One common use for function pointers is in conjunction with the system library routine qsort. The function in this case is a comparison function, which must take exactly two arguments that are declared as void star pointers, and return an integer result. (Since qsort was created before the introduction of the boolean datatype to the C++ language, it uses the older convention that nonzero integer values are true, and integer zero represents false). Assume, for example, that we have an array of 100 double precision values. These could be sorted as follows:

```c
double values[100];

int comp (void * a, void * b)
{
    double * d1 = (double *) a;
    double * d2 = (double *) b;
    return (*d1) < (*d2);
}

qsort (values, 100, sizeof(double), &comp);
```

The built-in function `sizeof`, used in this example, takes as argument a type description, and yields the size requirements (in byte units) for the type.

The qsort facility predates the Standard Template Library, and for newer code it is preferable to use the sort generic library routine from the STL (see Chapter 9).

It is also possible to form pointers to member functions. However, both the syntax and the interpretation are even more obscure than pointers to functions, and as the need for this facility is rather rare, it will not be discussed here.

## 3.5 Pointers to Structures

**Note** The arrow operator is a combination of dereference and field access. When pointers refer to classes or structures, a special syntax can be used to combine pointer dereferencing and member or data field access. The arrow operator, formed as a two-character combination of – and >, has the same effect as dereferencing a pointer and using the field access operator on the result:

```c
struct link {
    int value;
    link * next; /*-1pt pointer to next link in chain
```
3.6. POINTERS TO ARRAYS

";

link finalElement; //-1pt declare a single default element
link * firstLink = & finalElement; // -1pt set pointer to initially refer to this

(*firstLink).value = 7; // -1pt these two statements
firstLink->value = 7; // -1pt have the same effect

Assume we have a series of link values combined into a linked list structure. The following
prints out the values of the list, and illustrates the use of pointer operations:

for (link * p = aList; p != &finalElement; p = p->next)
    cout << *p << "\n";

3.6 Pointers to Arrays

In C++ there is a close association between a pointer and an array. In fact, one could argue
that the C++ language naturally assumes that all pointers always reference arrays. This is
because there are a number of operations that can be used with pointers that only make
sense if the pointer references an array. However, as we will shortly see, a danger is that
there is no run-time test to ensure that these expectations are satisfied.

An expression of type ‘array of ...’, when used by itself as a value, is converted into a
pointer type. An array of integers, for example, is converted into a pointer to integer. The
base type for the pointer is the element type, not the array type:

int values[100];

int * p = values; // -1pt legal, as values is converted into a pointer

As with arrays, pointer values can be subscripted:


No attempt is made to ensure that the index value is in range for the underlying array,
or even that the pointer does indeed reference an array. Accessing or modifying a value that
is out of range is a common, and subtle, source of programming errors:

p[310] = 7; // -1pt index value too large
p[-4] = 12; // -1pt index value too small

\begin{footnotesize}
\begin{center}
\begin{tabular}{l}
\textbf{NOTE} & \\
Pointers can be subscripted just like arrays
\end{tabular}
\end{center}
\end{footnotesize}\begin{footnotesize}
\begin{center}
\begin{tabular}{l}
\textbf{WARNING} & \\
Neither pointer nor array index values are checked to ensure they are in range
\end{tabular}
\end{center}
\end{footnotesize}
Rarely, very rarely, there is a legitimate reason for out of bound index values. More often it is simply a programming error. Occasionally it is a sign of malicious intent. The famous internet worm in 1988 operated by indexing a string array with out of range values, writing new assembly instructions to the known positions, then executing the new instructions.

It is also possible to perform arithmetic on pointer values. Adding or subtracting an integer from a pointer changes the pointer to reference the next or previous elements in sequence. That is, adding 1 to a pointer advances it to the next element, adding two skips an element, and so on. Again, no attempt is made to ensure the legitimacy of these operations.

A common use of pointer arithmetic is in loops. For example, strings in C++ are simple pointers to character arrays, terminated by a character with value zero. (We will discuss strings in more detail in Chapter 8). The following code fragment finds and prints all the vowels in a text string:

```cpp
char * text = "some text";

// p++ advances pointer to next location
for (char * p = text; *p \= ';' p++)
    if (isVowel(*p))
        cout \= "vowel value is \l" \= *p \= "\n";
```

Although it is legal, it is rare for pointers to be incremented or decremented by quantities other than one. However, adding integer values to a pointer to yield another pointer is relatively common.

### 3.7 References

A reference in C++ is an alias, an alternative way to name an existing object. Although most often references are implemented internally as pointers, this need not always be the case (the compiler can optimize away the underlying pointer in some circumstances). References and pointers differ in three important regards:

- A reference can never be null; it must always refer to a legitimate object.
- Once established, a reference can never be changed to make it point to a different object.
- A reference does not require any explicit mechanism to dereference the memory address and access the actual data value.

A reference is declared using the ampersand, as in the following:

```cpp
int i = 7;
```
3.7. REFERENCES

```java
int & j = i; // -1pt j is an alias for i
j++; // -1pt i is now 8
i += 3; // -1pt i is now 11, as is j
```

Operators do not operate on a reference, per se, rather they operate on the value that
the reference designates. For example, the increment operator above is not actually applied
to j, but rather to the value that j refers to, namely i.

A reference can be used as the target of an assignment. Some functions will return a
reference as a result for precisely this reason. This is illustrated by the following somewhat
contrived example:

```java
int values[100];
int & index(int i) - return values[i + 2]"
index(27) = 12; // -1pt changes values[29];
```

References are very similar to the way that Java interprets assignment of object values.
Consider the following snippet of Java code:

```java
class box - // -1pt Java box
    public int value = 0;
"

box x = new box();
box y = x;
x.value = 7;
    // -1pt changing x will change y
System.out.println("Value of x.box" + x.value);
    // -1pt reassigning x will not change y
x = new box();
System.out.println("Value of y.box" + y.value);
```

Since x and y reference the same object, a change in x will result in an observed change
in y. A difference between C++ and Java references can be seen in a subsequent assignment.
Changing the variable x itself in Java would have no effect on y, which could continue to refer to
the previous value. Whereas such a change would alter both values in C++. This
difference is perhaps more easily seen in the context of parameter values, which are really
just a form of assignment. This is illustrated in the next section.
3.7.1 Pass by Reference Parameters

The most common use of references is as a pass by reference parameter. A reference parameter is an alias for the corresponding actual argument value. Modifications to one can have the effect of producing modifications to the other. To illustrate, consider the following simple function:

```c
void passTest (int & i)
{
    i++;    
    i = 7;
}
```

```c
int main ()
{
    int j = 5;
    passTest(j);
    cout << j << 'n';
    return 0;
}
```

If the argument to passTest is declared as pass-by-value, without the ampersand, then the modifications to i within the procedure have no effect on j. But passed by reference, as shown, the value j is an alias for i. Thus, changes to j result in a modification of i.

Primitive types in Java are passed by value, just as in C++. We cannot use the wrapper class `Integer` in this example because an `Integer` value cannot be changed once it is created. However, it is useful to note that in general the parameter passing mechanism used by Java to pass object values is different from either of the possibilities permitted in C++. Consider a superficially similar function in Java:

```java
static void passTest (box i)
{
    i.value++;    
    i = new box(7);
}
```

```java
public static void main (String [ ] args)
{
    box j = new box(5);
    passTest(j);
    System.out.println("J is "+j.value);
}
```
3.7. REFERENCES

In Java i is a reference to j, and thus the change in i will alter j. But reassigning i to a new value has no effect on j. (An accurate description is to say that a Java parameter is an object reference that is passed by value). The following table of the final value for j illustrates that the Java parameter passing mechanism is different from both pass by value and pass by reference in C++:

<table>
<thead>
<tr>
<th>value of j</th>
<th>C++ pass by value, Java primitive</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C++ pass by reference</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Java objects</td>
<td>6</td>
</tr>
</tbody>
</table>

The reader would do well to study this carefully in order to determine how the three different results are produced.

3.7.2 References as Results

References can also be used as a result type for a function. There are two reasons for doing so:

- A reference can be used as the target of an assignment. Therefore, a function call that returns a reference can be used on the left side of an assignment.
- Returning a reference is more efficient than returning a value. Therefore large values can be returned by reference.

A good example of the first case is a subscript operator in a string or vector class. In Chapter 4 we will discuss one such example. There the string class overlays a more primitive array of character values, and the subscript operator simply returns a reference to an individual character:

class string {
    ...
    char & operator [ ] (unsigned int index)
    - return buffer[index]; 
    ...
}

private:
    char * buffer;
    ";

Since the subscript operator returns a reference, individual character positions can be changed using assignment:
string text = "name:";
text[0] = 'f'; //-1pt change name to fame

The second reason to use a reference as a result is because returning a reference is much more efficient than creating and returning a new value. However, care must be exercised. As we will note in Chapter 4, a reference must always refer to an object that is known to exist. Returning a reference to a value that will soon be deleted, such as a local variable, is a certain recipe for eventual disaster. The following example, illustrates the hazard:

double & min (double data[], int n) 
  
  double minVal = data[0];
  for (int i = 1; i < n; i++)
    if (data[i] < minVal)
      minVal = data;
  return minVal; // -1pt error, reference to local
"

Test Your Understanding

1. What is a pointer?
2. What are three reasons why pointers are useful in programs?
3. What is a null pointer?
4. What does it mean to dereference a pointer? How is this operation written in C++?
5. What is the relationship between addresses and pointers? How can the address of a variable be assigned to a different pointer variable?
6. What is the difference between modifying a pointer variable and modifying the value the pointer references. Give an example statement to illustrate each.
7. What is the difference between a pointer to a constant and a constant pointer?
8. What is special about the void * pointer type?
9. What does the built-in function sizeof take as argument? What does it compute?
10. What is an equivalent synonym for the arrow operation p->a?
11. What are some of the ways that pointers and arrays are similar?
12. What is the difference between a pointer and a reference?

13. How is pass by reference different from pass by value? How are each different from the Java parameter passing mechanism?

14. Given the following function definition:

```c
void swap (int & x, int & y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Describe the effect of the following statements:

```c
int data[10];
data[3] = 7;
int i = 3;
swap (i, data[i]);
```

15. What are two reasons why a function might return a reference type as a result?