Pointers 4

One of the key features of the C++ language is the concept of a *pointer*. We will see later in this chapter that pointers are extremely useful for allocating memory for arrays whose sizes are not known when the code is compiled. We will see in Chap. 5 that they also have use when writing functions that allow us to repeat the same operation on different variables. We conclude this chapter by discussing some features of pointers that have been introduced in recent C++ standards.

4.1 Pointers and the Computer's Memory

Pointers are best introduced by explaining how they relate to the storage of variables in the computer's memory.

4.1.1 Addresses

Let us suppose that an integer variable `total_sum` is declared and assigned the value 10:

```
  int total_sum = 10;
```

The address—that is, location—of this variable in the computer's memory is given by `&total_sum` and can be printed to the console in the usual way (as shown below) although this address will not be meaningful to humans.

J. Pitt-Francis and J. Whiteley, *Guide to Scientific Computing in C++*, Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-73132-2_4

```
std::cout << &total_sum << "\n";
```

When the integer variable `total_sum` is declared, memory is allocated to this variable, and the location of this memory will not vary throughout execution of the code. As such, the expression `&total_sum`, which represents the address of this location, will take a constant value throughout execution of the code.

4.1.2 Pointer Variables

In addition to data types such as integers and floating point numbers that we have encountered earlier in this book, we may also declare *pointer variables* which are variables that store addresses—that is, the location in the computer's memory—of other variables. In the code below, `p_x` is a pointer to a double precision floating point variable, and `p_i` is a pointer to an integer variable. The pointer `p_x` may then be used to store the address of a double precision floating point number, whilst `p_i` may be used to store the address of an integer. The asterisk that prefixes these variables when they are declared indicates that these variables are pointers. In this book, we follow a coding standard where all pointer variables, apart from those introduced later in this chapter that represent arrays, have names that begin with `p_` to denote that they are a pointer variable: a discussion of conventions such of these that are used for variable names, which forms a part of what is known as *coding standards*, is given in Sect. 6.6.

```
double* p_x;
int* p_i;
```

Note that the spacing can vary, so that `int* p_i` and `int *p_i` are equivalent. However, `int* p_i` states more clearly that the type of `p_i` is a pointer to an integer, rather than an integer.

All pointer variables require an asterisk when they are declared. Hence, in the code below, `p_x`, `p_y`, `p_i` are pointers, while `j` is an integer variable.

```
1   double *p_x, *p_y;
2   int *p_i, j;
```

When declaring more than one pointer on a line the asterisk must be repeated as shown in line 1 of the listing above, which means that `int* p_i` in line 2 would be less appropriate as only one variable (`p_i`) is a pointer variable. For this reason, we recommend only one pointer declaration per line.

Now we have explained how to declare a pointer variable, and what these variables represent, we explain how to use them.

4.1.3 Example Use of Pointers

If a variable p_x has been declared as a pointer to a double precision floating point number, then it is clearly important to distinguish between: (i) the location of the memory to which this pointer points at (denoted by p_x); and (ii) the contents of this memory (denoted by *p_x). The asterisk operator in *p_x is called a *pointer de-reference* and can be thought of as the opposite to the & operator introduced in Sect. 4.1.1.

The code below shows how pointers to double precision floating point variables may be combined with double precision floating point variables.

```
double y, z;      // y, z store double precision numbers
double* p_x;      // p_x stores the address of a double
                  // precision floating point number
z = 3.0;
p_x = &z;         // p_x stores the address of z
y = *p_x + 1.0;  // *p_x is the contents of the memory
                 //  p_x, i.e. the value of z
```

4.1.4 Warnings on the Use of Pointers

A variable pointer should not be used until first having been assigned a valid address. For example, the following fragment of code may cause problems that are difficult to locate.

```
1  double* p_x; // p_x can store the address of a double
2               // precision number - haven't said which
3               // address yet
4
5  *p_x = 1.0; // trying to store the value 1.0 in an
6              // unspecified memory location
```

In the code above, we haven't specified the location of the double precision floating point variable that p_x points at. It may therefore be pointing at *any* location in the computer's memory. Changing the contents of an unspecified location in a computer's memory—as is done in line 5 of the code above—clearly has the potential to cause problems that may be hard to locate. This problem may be avoided by the use of the new keyword as shown below to allocate a valid memory address to p_x, and the `delete` keyword which releases this memory to be used by other parts of the program when this memory is no longer required.

```cpp
double* p_x;       // p_x stores the address of a double
                   // precision floating point number

p_x = new double; // assigns an address to p_x
*p_x = 1.0;        // stores 1.0 in memory with
                   // address p_x
delete p_x;        // releases memory for re-use
```

A further reason to use pointers with care is shown in the code below. The first time `y` is printed (in line 5) it takes the value 3: the second time `y` is printed (in line 7) it takes the value 1 even though `y` is not explicitly altered in the code between these two lines. This is because the line between the `std::cout` statements, line 6, has altered the value of `y`, possibly unintentionally, by using the pointer variable `p_x` (which contains the address of `y`) to change the value of `y`.

```cpp
1   double y;
2   double* p_x;
3   y = 3.0;
4   p_x = &y;
5   std::cout << "y = " << y << "\n";
6   *p_x = 1.0;  // This changes the value of y
7   std::cout << "y = " << y << "\n";
```

A situation where the contents of the same variable may be accessed using different names, such as in the code above, is known as *aliasing*. In C++, this is most likely to happen when pointers are involved, either when two pointers alias the same address in memory, or when a pointer references the contents of another variable. When one or more pointers allow the same variable to be accessed using different names, the aliasing is known as *pointer aliasing*.

4.2 Dynamic Allocation of Memory for Arrays

One of the main uses of pointers is the dynamic allocation of memory for storing arrays. In Sect. 1.4.5, we explained how arrays could be declared when the size of the array was known in advance. However, we do not always know the sizes of the arrays in a program when we compile the code. In Sect. 3.5, for example, we demonstrated how to allow the user of a code to specify the number of nodes in a finite difference grid when executing the code. If the coordinates of the nodes in this mesh were to be stored in an array we would not know, when compiling the code, what size to make this array. Under these circumstances, using the method of declaring arrays given in

Sect. 1.4.5, we have to compile the code with some estimate of the size of this array. If we overestimate the size of this array, we are being wasteful of computational memory with the potential effect of preventing the execution of the code on a system with insufficient memory. If we underestimate the size of this array, the program will almost certainly crash. In either case, we will then have to recompile the code with a new estimate of the array size. The use of pointers to dynamically allocate memory for arrays avoids these problems, as we do not need to know the array size at compile time.

A further use of pointers for dynamically allocating memory is for the efficient storage of irregularly sized arrays, for example a lower triangular matrix. If a lower triangular matrix is stored in an array as described in Sect. 1.4.5, we will have to allocate the same number of columns to each row of the matrix. As we know that roughly half these entries are zero, we are being wasteful of computational memory. Dynamic allocation of memory allows us to allocate memory more prudently.

Memory can be allocated using the `new` operator, and deallocated using the `delete` operator.

4.2.1 Vectors

To use pointers to create a one-dimensional array of double precision floating point numbers of length `10` called `x`, we use the following section of code.

```
double* x;
x = new double [10];
```

The elements of the array may then be accessed in exactly the same way as if the array had been created by using the type of declaration introduced in Sect. 1.4.5. In the dynamic allocation of memory for the array using the pointer `x` above, `x` stores the address of the first element of the array. This can be seen by printing out both the pointer `x` and the address of the first element of the array, as shown below.

```
std::cout << x << "\n";
std::cout << &x[0] << "\n"; //prints the same value
```

The memory allocated to `x` may be, and should be, deallocated by using the statement below when this array is no longer required.

```
delete[] x;
```

Always be sure to free any memory allocated when it is no longer required—a code can very quickly use all available memory otherwise. In later chapters of this book, when we develop a class of vectors, we will see that one advantage of writing a class of vectors is that memory allocated to a vector is automatically freed when appropriate.

An example code that uses dynamically allocated memory for arrays is shown below. This code creates two arrays, x and y, both of size 10. Elements of x are then assigned manually. Elements of y are then set to be twice the value of the corresponding element of x. Finally, all memory allocated is deleted.

```
#include <iostream>

int main(int argc, char* argv[])
{
   double* x;
   double* y;
   x = new double [10];
   y = new double [10];

   for (int i=0; i<10; i++)
   {
      x[i] = ((double)(i));
      y[i] = 2.0*x[i];
   }

   delete[] x;
   delete[] y;

   return 0;
}
```

4.2.2 Matrices

Memory for matrices may also be allocated dynamically. For example, to create a two-dimensional array of double precision floating point numbers with 5 rows and 3 columns called A we use the following section of code.

Listing 4.1 Dynamic memory allocation for a matrix

```
   int rows = 5, cols = 3;
   double** A;
   A = new double* [rows];
   for (int i=0; i<rows; i++)
   {
      A[i] = new double [cols];
   }
```

The array may then be used in exactly the same way as if it had been created by using the declaration

```
double A[5][3];
```

When allocating memory for the matrix dynamically in the code above, the variable A—which has been declared using line 2 of Listing 4.1—has the following properties after the fragment of code has been executed:

- each A[i] is a pointer, and contains the address of A[i][0]; and
- A contains the address of the pointer A[0].

As such, the variable A is an array of pointers, which explains the two asterisks in line 2 of Listing 4.1. Line 3 of this listing specifies that A is a pointer to an array of pointers to double precision floating point numbers, and that this array is of size rows. The for loop in this listing then specifies that each pointer in the array itself points to an array of double precision floating point numbers of length cols. This has the effect that A[i]—which is a pointer—stores the address of the entry A[i][0], that is, the first entry of row i.

As was the case for vectors, it is important to deallocate memory dynamically allocated for a matrix when it is no longer needed. The memory allocated for the matrix A in Listing 4.1 may be freed using the following code.

```
for (int i=0; i<rows; i++)
{
   delete[] A[i];
}
delete[] A;
```

We cannot emphasise enough how important it is to always delete any memory dynamically allocated, particularly memory allocated inside loops—if not you will soon run out of memory.

4.2.3 Irregularly Sized Matrices

Suppose we want to construct a lower triangular matrix A of integers with 1,000 rows and 1,000 columns. This may clearly be done using the declaration below.

```
int A[1000][1000];
```

However, the declaration above wastes a considerable amount of memory storing the super-diagonal entries of the matrix which we know in advance all take the value 0. We may avoid wasting this memory by allocating the memory for this matrix dynamically, and only allocating memory for the diagonal and sub-diagonal elements. This is demonstrated in the fragment of code below, where in row i of the matrix we declare i+1 nonzero elements: that is, 1 element in row 0, 2 elements in row 1, and so on. Memory can, and should be, deleted in the same way as demonstrated in the previous section when this array is no longer required.

```
int** A;
A = new int* [1000];
for (int i=0; i<1000; i++)
{
   A[i] = new int[i+1];
}
```

Although the fragment of code above does correctly allocate the memory required for a lower triangular matrix it should be used with care: errors would result if, for example, the entry A[9][19] were to be used in a code. When we develop classes later in this book, we will see how the use of classes may avoid problems such as this.

4.3 Tips: Pointers

The concept of pointers is one that inexperienced C++ programmers often struggle with. We strongly urge the reader to attempt the exercises at the end of this chapter to improve their understanding of this topic. In this section, we give tips on the use of pointers. In all other chapters the tips section is the final section before the exercises. This chapter is the exception because some of the caution in the following tips may be mitigated in modern C++. We introduce these advanced topics in Sect. 4.4 which you might ignore on first reading.

4.3.1 Tip 1: Pointer Aliasing

In Sect. 4.1.4, we gave an example where a pointer variable p_x was pointing to the memory location of the double variable y. A change was made to that variable by de-referencing the pointer p_x. This situation might lead to some confusion, although in a short code fragment it is easy to see that the two variables are leading to the same place: *p_x is an *alias* for y.

In large-scale programs, it may not be so easy to see where pointers are aliases for other variables. This is because the information that two names are pointing to same place may not be available in the same screen-full of code, or even in the same file. A good example of this would be a vector or matrix addition operation in which the vectors or matrices are stored as arrays and passed into a *function* via pointers. We will deal with functions in the next chapter, but for now you need to be aware that the code for the function may be in a different file and that the variables may take different names inside the function definition. The operation to compute the matrix sum $\mathbf{A} = \mathbf{B} + \mathbf{C}$ would probably be implemented in such a function by a nested loop over the elements of the arrays, so that the actual implementation becomes an element-wise `A[i][j] = B[i][j] + C[i][j]`. There may be unknown pointer-aliasing in this function, because the user might wish to increment one matrix by another, i.e. to compute $\mathbf{X} = \mathbf{X} + \mathbf{Y}$. It turns out that this pointer aliasing will be safe, because the inner loop will effectively be calculating `X[i][j] += Y[i][j]` as intended. Each of the (i, j) components of the result is independent of the others.

However, what if the user were using a matrix–matrix product operation? In the computation $\mathbf{A} = \mathbf{BC}$, the component `A[i][j]` depends on parts of **B** and **C** other than `B[i][j]` and `C[i][j]`. This means that, if the user wishes to compute $\mathbf{X} = \mathbf{XY}$ using a function written for calculating $\mathbf{A} = \mathbf{BC}$, there is a chance that some components of **X** will be written to before they are read—leading to an incorrect calculation. One way to resolve this aliasing issue is to produce the matrix–matrix product result in temporary storage before copying it into the output argument **A**. However, this solution is inefficient in cases where there is no pointer aliasing, especially when the sizes of the matrices are large. Another solution to the issue is to provide two versions of the matrix-matrix product operation: one which is efficient but only safe to use when there is no pointer aliasing and one which is safe to use in all circumstances.

One can see that the problem of pointer aliasing is deeper than might appear from the trivial example in Sect. 4.1.4. In general, there is no correct solution to these issues. Compiler writers spend a great deal of time finding places where pointer aliasing has (or has not) definitely happened so that code optimisation is only applied in situations where it is safe to do so.

4.3.2 Tip 2: Safe Dynamic Allocation

There may be circumstances under which it is not possible to allocate memory either because the number of items in an array has been set with a negative argument or because there is not enough physical memory available to the program. Setting the number of elements in an array to a negative number is easier than you might think. If the size of a problem is configured via an input file, then a size may easily be mistyped. More subtly, if a number is assigned to an integer that is larger than the maximum value that can be stored by that integer, then the integer value stored may actually be a negative number: this is known as an *overflow error*.

Implementations of C++ may vary over how they treat such errors. The default behaviour is to throw an *exception* when a memory error is encountered. We will deal with catching exceptions in Chap. 9 and note that an exception could terminate your program. Should your implementation of C++ not throw this sort of exception, then a safe way to program is to test that your variable has been assigned a value as the code fragment below illustrates.

```
double* p_x;
p_x = new double[10000];
assert (p_x != NULL);
```

4.3.3 Tip 3: Every `new` Has a `delete`

We pointed out earlier in this chapter that all dynamically allocated memory must be freed, or else you may run out of memory. This problem is particularly noticeable when memory is dynamically allocated inside the body of a `for` loop, such as the one shown below.

```
for (int i=0; i<10000; i++)
{
   double** A;
   A = new double* [50];
   for (int j=0; j<50; j++)
   {
      A[j] = new double [50];
   }
}
```

Each time the body of the loop in the code above is executed, new memory is allocated for the array `A`. The memory from the previous execution has not yet been freed, although it will not be available as the array `A` will be stored in the memory that has been allocated most recently: there is no automatic garbage collection for memory which is no longer accessible. You will see, when we discuss functions in Chap. 5, that the same problem may arise when memory is allocated inside functions, but not freed before the function ends.

If you do not delete memory which you have allocated dynamically, then that memory will not be accessible until your program finishes (when all memory is handed back to the system). If you request more memory than you need, then it may be that the physical memory of the computer will be exhausted—your computer will run much more slowly and further memory allocation may fail.

There are several ways around this issue. The first and foremost is to ensure that every `new` in your program is matched with a `delete` somewhere else. A second way to make sure that inaccessible or unnecessary memory is freed up is to run your program through a memory debugger (see Sect. 10.6 for more details). Another solution, adopted by seasoned C++ programmers is to use *shared pointers*. These are an advanced language feature which allow memory to be automatically de-allocated once there is no longer any other part of the program which can access it.

4.4 Modern C++ Memory Management

In Sect. 1.1.2, when discussing why you should write scientific programs in C++, we claimed that its flexible memory management gave it an advantage over languages which use garbage collection, such as Java. However we also gave a caveat: this flexible memory management means that you, the programmer, are responsible for making sure that memory is managed properly. Many novice C++ programmers are confused by dynamic memory allocation and become deterred when they learn that it is up to them to know when dynamically created data should be freed up with `delete`. The good news for C++ programmers is that over recent years the C++ standard has introduced smart pointer constructs which facilitate memory management—providing an efficient compromise between giving responsibility to the programmer and automatic run time garbage collection. These constructs were first introduced in the C++11 specification and have been refined in subsequent specifications.[1] In this chapter we restrict attention to modern C++ *memory management* but we will return to other modern C++ functionality in Chap. 8.

4.4.1 The `unique_ptr` Smart Pointer

In our first tip of this chapter, in Sect. 4.3.1, we warned about the dangers of pointer aliasing. In particular we noted that there may be times when a programmer assumes that two pointers are pointing to different pieces of data, but that this assumption may not be true. When two pointers are pointing to the same piece of data it may lead to bugs such as an element of a matrix being overwritten before its value has been read.

C++11 provides a smart pointer type which can guard against pointer aliasing errors. This smart pointer `unique_ptr` allows the run-time system to monitor certain pointers on an individual basis. The example of its use, given in Listing 4.2, is a little contrived because the true power of the construct cannot be seen until it is used with functions. The program will, however, serve to illustrate a few of the main features. Your C++ compiler may not accept this program since most current compilers are set to read older C++98 standard programs by default. In order to compile the program you will need to add a flag to indicate that the code adheres to the C++11 standard. In the case of the GNU compiler this means

```
g++ -std=c++11 -o Unique Unique.cpp
```

or similar.

In line 6 of Listing 4.2 a new `int` is dynamically created via the `new` keyword and its address assigned to a `unique_ptr` called `p_x`. Note that the type description of

[1]At the time of writing the second edition of this book the relevant specifications are C++11, C++14 and C++17.

the unique_ptr contains the type of the entity to which it points, which in this case is int, in angle brackets. This angle bracket notation is a *template* description and we will see more of this in Chap. 8. The purpose of new int in round brackets on line 6 is to dynamically create an int and pass its location into p_x. The variable p_x now acts as a facade through which the actual address of the dynamically-created integer storage may be accessed. There is more happening behind the scenes, but the reader may interpret line 7 as a de-reference used to store a value in the memory location at this address.

We demonstrate that the compiler won't allow us to easily assign the value of p_x by two lines which have been commented out: line 11 attempts to assign it to a raw pointer and line 16 attempts to assign it to another unique_ptr. The correct way to get the value out of p_x (line 12) is to use the get() function to get the actual address of the managed data. Meanwhile the correct way to assign from one unique_ptr to another is for the ownership of the resource to be transferred between them with the function std::move(). This is demonstrated in line 17. Lines 18 and 19 show that the unique_ptr variables can be evaluated as Boolean values: true if the variable is managing a resource and false if not.

Note that in Listing 4.2 there is no explicit call to delete to match with the new on line 6. It is actually the case that, because the unique_ptr is managing the resource, it is able to automatically free up memory. On line 20, p_z is told to relinquish ownership and this implicitly calls delete on the memory originally created on line 6.

Listing 4.2 Example program to demonstrate the use of unique_ptr

```
1  #include <memory> // Requires C++11 or above
2  #include <cassert>
3
4  int main()
5  {
6    std::unique_ptr<int> p_x(new int);
7    *p_x = 5; // 'de-reference' to alter contents
8
9    // The following won't compile because p_x
10   // is not a raw pointer to int
11   //   int* p_y = p_x;
12   int* p_y = p_x.get(); // Get raw pointer
13
14   std::unique_ptr<int> p_z;
15   // The following won't compile
16   // p_z = p_x;
17   p_z = std::move(p_x); // Transfer ownership
18   assert(p_z); // Test p_z is in use
19   assert(!p_x); // Test that p_x not in use
20   p_z.reset();
21   assert(!p_z); // Test p_z is also not in use
22   return 0;
23 }
```

4.4.2 The `shared_ptr` Smart Pointer

The mismatch between the last code example (Listing 4.2) and our previously sound advice in Sect. 4.3.3, "Every `new` has a `delete`", prompts us to introduce the variable type `shared_ptr`. This smart pointer construct was not available in the official C++ standard until C++14 but some C++11 compilers such as the GNU compiler support it anyway.

The concept behind a smart shared pointer is simple. Alongside the address of the underlying resource the pointer also keeps track of a count of the number of times this resource has been used. Initially the count will be 1, but it will increment when the pointer is passed between various parts of the program. Whenever a use of the pointer finishes the usage count will be decremented. When the count drops to 0, and there are no known uses of the pointer, the original resource will be freed up. This all happens automatically, without the user having to worry about it. It is effectively a local garbage collector which manages a small piece of memory.

The code presented in Listing 4.3, illustrating the use of a smart shared pointer, is again a little contrived, but it represents how this automatic memory management might work in practice. In line 6 a new integer value is dynamically created and its location is stored in a `shared_ptr` variable `p_x`. As with the previous C++11 example this new smart pointer is templated with the type of its argument in angle brackets. In line 10 another `shared_ptr` variable is created and it is assigned to the same value as `p_x` (this is an assignment which would not be possible with the `unique_ptr` type). In line 12 `p_y` is reset so that it relinquishes any claim on memory. While lines 10 and 12 are just a simple assignment and a reset, respectively, their use here actually represents general wider uses of a shared pointer. Copies of pointers may be made when they are passed into functions, as we will see in Chap. 5, or passed into *containers*—of the kind introduced in Chap. 8. When functions or containers finish, their copy of the pointer is not needed and is, in effect, reset.

Listing 4.3 Example program to demonstrate the use of `shared_ptr`

```
 1  #include <memory> // Requires C++11 or above
 2  #include <iostream>
 3
 4  int main()
 5  {
 6     std::shared_ptr<int> p_x(new int);
 7     std::cout<<"p_x use count: "<<p_x.use_count()<<"\n";
 8     *p_x = 5; // 'de-reference' to alter contents
 9     // Use this pointer elsewhere
10     std::shared_ptr<int> p_y = p_x;
11     std::cout<<"p_x use count: "<<p_x.use_count()<<"\n";
12     p_y.reset();
13     std::cout<<"p_x use count: "<<p_x.use_count()<<"\n";
14     p_x.reset();
15     std::cout<<"p_x use count: "<<p_x.use_count()<<"\n";
16     return 0;
17  }
```

Lastly in line 14 the original pointer is reset. This has, again, the same effect as p_x going out of use: its claim on the data is relinquished. In this case the use count will drop to 0 and the smart pointer will automatically free up the original memory which was created on line 6.

Throughout Listing 4.3 the use count of the main shared pointer p_x is written to the console. The output of this program is given below and reflects the number of uses of the shared resource. This count is originally 1 when p_x is created, then 2 when p_y shares the resource, and 1 when p_y relinquishes its use on line 12. Finally, when p_x relinquishes its use, the count drops to 0.

```
p_x  use  count: 1
p_x  use  count: 2
p_x  use  count: 1
p_x  use  count: 0
```

4.5 Exercises

4.1 Write code that declares an integer i to take the value 5. Declare a pointer to an integer p_j, and store the address of i in this pointer. Multiply the value of the variable i by 5 by using a line of code that *only uses the pointer variable*. Declare another pointer to an integer p_k and use the new keyword to allocate a location in memory that this pointer stores. Then store the contents of the variable i in this location. Now change the value pointed to by p_j to 0. Check that your program is correct by outputting the value of i and values pointed to by p_j and p_k.

4.2 Assign values to two integer variables. Swap the values stored by these variables using only pointers to integers.

4.3 Write code that allocates memory dynamically to two vectors of double precision floating point numbers of length 3, assigns values to each of the entries, and then de-allocates the memory before the code terminates. Extend this code so that it calculates the scalar (dot) product of these vectors and prints it to screen before the memory is de-allocated. Put the allocation of memory, calculation and de-allocation of memory inside a for loop that runs 1,000,000,000 times: if the memory is not de-allocated properly your code will use all available resources and your computer may struggle.

4.4 Write code that dynamically allocates memory for three 2×2 matrices of double precision floating point numbers, A, B, C, and assigns values to the entries of A and B. Let $C = A + B$. Extend your code so that it calculates the entries of C, and then prints the entries of C to screen. Finally, de-allocate memory. Again, check you have de-allocated memory correctly by using a for loop as in the previous exercise.

4.5 In Sect. 4.4 we introduced the `unique_ptr` and `shared_ptr` constructs. A useful further smart pointer is the `weak_ptr`, which is a smart pointer that does not contribute to the use count. It can be used in situations where variables need to be accessed, but only when they exist. It has functions `expired` and `lock` which can be used to check if its resource has been deleted and, if it has not been deleted, to get to the resource.

Copy Listing 4.3 and compile it with a compatible C++11 compiler. Now add an extra smart pointer: a `weak_ptr` which is initialised to the value `p_x`. Experiment with printing the value original of `p_x` (i.e. the value 5) via this weak smart pointer. Try this before, and after, the `p_x` is reset on line 14.