Assessments

The programming solutions for each chapters' questions can be found in our GitHub repository at the following URL: `https://github.com/PacktPublishing/Demystified-Object-Oriented-Programming-with-CPP/tree/master`. Each full program solution can be found in the GitHub under the appropriate chapter heading (subdirectory, such as `Chapter01`) in the subdirectory `Assessments`, in a file that corresponds to the chapter number, followed by a dash, followed by the solution number in the chapter at hand. For example, the solution for question 3 in chapter 1 can be found in the subdirectory `Chapter01/Assessments` in a file named `Chp1-Q3.cpp` under the aforementioned GitHub directory.

The written responses for non-programming questions can be found in the following sections. Should an exercise have a programming portion and a follow-up question, the answer to the follow-up question may be found both in the next sections and in a comment at the top of the programming solution on GitHub (as it may be appropriate to review the solution in order to fully understand the answer to the question).

Chapter 3 – Indirect Addressing: Pointers

1. **a – f**: Please see `Chapter03/Assessments/Chp3-Q1.cpp` in the GitHub repository.
   d: (follow-up question) `Print(Student)` is less efficient than `Print(const Student *)` as the initial version of this function passes an entire object on the stack, whereas the overloaded version passes only a pointer on the stack.

2. Assuming we have an existing pointer to an object of type `Student`, such as:

   `Student *s0 = new Student;` (this `Student` is not yet initialized with data)

   a: `const Student *s1;` (does not require initialization)

   b: `Student *const s2 =  s0;` (requires initialization)

   c: `const Student *const s3 = s0;` (also requires initialization)

3. Passing an argument of type `const Student *` to `Print()` would allow a pointer to a `Student` to be passed into `Print()` for speed, yet the object pointed to could not be dereferenced and modified. Yet passing a `Student * const` as a parameter to `Print()` would not make sense because a copy of the pointer would be passed to `Print()`. Marking that copy additionally as `const` (meaning not allowing changing where the pointer points) would then be meaningless as disallowing a *copy* of a pointer to be changed has no effect on the original pointer itself. The original pointer was never in jeopardy of its address being changed within the function.

4. There are many programming situations that might use a dynamically allocated 3-D array. For example, if an image is stored in a 2-D array, a collection of images might be stored in a 3-D array. Having a dynamically allocated 3-D array allows for any number of images to be read in from a filesystem and stored internally. Of course, you'd need to know how many images you'll be reading in before making the 3-D array allocation. For example, a 3-D array might hold 30 images, where 30 is the third dimension to collect the images in a set. To conceptualize a 4-D array, perhaps you would like to organize sets of the aforementioned 3-D arrays.

   For example, perhaps you have a set of 31 images for the month of January. That set of January images is a 3-D array (2-D for the image and the third dimension for the set of 31 images comprising January). You may wish to do the same for every month. Rather than having separate 3-D array variables for each month's image set, we can create a fourth dimension to collect the years' worth of data into one set. The fourth dimension would have an element for each of the 12 months of the year. How about a 5-D array? You can extend this image idea by making the fifth dimension a way to collect various years of data, such as collecting images for a century (fifth dimension). Now we have images organized by century, then organized by year, then by month, and then by image (the image requiring the first two dimensions).