WORKED EXAMPLE 6.1 Rolling the Dice

Your task is to analyze whether a six-sided die is fair by counting how often the values 1, 2, ..., 6 appear. You are given an array of die toss values, and you should fill an array of counters, where `counters[i]` is the number of times the value `i` was tossed. (Leave `counters[0]` unused.)

Step 1 Decompose your task into steps.

Overtly, we are given a task that can be solved in one step, namely to visit the die toss values and increment the counters. However, to produce a runnable program, we also need to produce an array of test values, and we need to display the outcome. Thus, we have three steps:

- Produce test values.
- Count the occurrences of each value.
- Display the results.

Step 2 Determine which algorithm(s) you need for each step.

To produce test values, we could generate random numbers between 1 and 6, but then we wouldn't know whether we get the correct answer. Instead, let's generate test values for which we know the answer, and fill the array as follows:

```
1 2 3 4 5 1 2 3 4 5 1 2
```

We now expect the following counts:

```
1: 3
2: 3
3: 2
4: 2
5: 2
6: 0
```

We would like to format the output in this way. We haven't seen this exact output format loop in this chapter, but it is plausible that a simple loop can generate the desired output.

Finally, let us turn to the task of counting the occurrences of each value. Section 4.7.2 shows how to count the number of 1s, with a loop

```
for (int i = 0; i < size; i++)
{
   if (values[i] == 1)
   {
      counters[1]++;
   }
}
```

We could have six loops of this form, but that is not very elegant. After all, when you have a die value, you know which counter needs to be incremented. It is the counter corresponding to the value itself. Therefore, the following loop solves our problem.

```
for (int i = 0; i < size; i++)
{
   int value = values[i];
   counters[value]++;
}
```

Step 3 Determine functions, arguments, and return values.

We will provide a function for each step:

- `generate_test_values`
- `count_values`
- `print_counters`

The `generate_test_values` function fills an array of test values.

```
/**
   Generates a sequence of die toss values for testing.
   @param values the array to be filled with die toss values
   @param size the size of the values array
*/
void generate_test_values(int values[], int size)
```

The `count_values` function receives the die toss values, and it must also receive the array of counters so that it can update them:

```
void count_values(int values[], int size, int counters[])
```

We don't need to supply the size of the counters array because we know it contains seven elements. But let us pause for a minute. Suppose the researchers changed their mind and wanted to investigate 12-sided dice.

The same algorithm works, but we need more counters. Let's implement the more general function.

```
/**
   Counts the number of times each value occurs in a sequence of die tosses.
   @param values an array of die toss values.
   @param size the size of the values array
   @param faces the number of faces on the die
   @param counters an array of counters. counters[j] is filled with the
   count of elements of values that equal j. counters[0] is not used.
*/
void count_values(int values[], int size, int faces, int counters[])
```

The function modifies the `counters` array. This is not a problem since array parameters are always reference parameters.

Finally, the `print_counters` function prints the value of the counters. Again, we will want to support an arbitrary number of die faces, so we will supply that number as an argument.

```
/**
   Prints a table of die value counters.
   @param faces the number of faces on the die
   @param counters an array of counters
*/
void print_counters(int faces, int counters[])
```

Step 4 Implement each function, using helper functions when needed.

We start with the `generate_test_values` function.

```
void generate_test_values(int values[], int size)
{
   int next = 1;
   for (int i = 0; i < size; i++)
   {
      values[i] = next;
      next++;
      if (next == 6) { next = 1; }
   }
}
```

Here are the `count_values` and `print_counters` functions.

```
void count_values(int values[], int size, int faces, int counters[])
{
   for (int j = 1; j <= faces; j++) { counters[j] = 0; }
   for (int i = 0; i < size; i++)
   {
      int value = values[i];
      counters[value]++;
   }
}

void print_counters(int faces, int counters[])
{
   for (int j = 1; j <= faces; j++)
   {
      cout << j << ": " << counters[j] << endl;
   }
}
```

Step 5 Consider boundary conditions for the functions that you are implementing

The `generate_test_values` function fills `values[i]`, and we know that `i` starts at `0` and is less than `size`. In contrast, the `print_counters` function prints `counters[j]` where `j` starts at `1` and is less than *or equal to* `faces`. This is correct since the `counters` array has size `faces + 1`, and we don't use the element at position `0`.

It is a good idea to update the documentation and make it clear which assumptions underlie this function:

```
/**
   Prints a table of die value counters.
   @param faces the number of faces of the die.
   @param counters an array of counters of length faces + 1.
   counters[0] is not printed.
*/
```

Now consider this code in the `count_values` function:

```
int value = values[i];
counters[value]++;
```

How do we know that `counters[value]` is valid? We need to know that the elements of the `values` array are between 1 and `faces`. Again, we should document this information:

```
/**
   Counts the number of times each value occurs in a sequence of die tosses.
   @param values a vector of die toss values. Each element is >= 1 and <= faces.
   @param faces the number of faces of the die
```

```
   @param counters an array of counters of length faces + 1. counters[j]
   is filled with the count of elements of values that equal j. counters[0] is
   not used.
*/
```

Step 6 Assemble and test the complete program.

The `main` function allocates the array of counters, generates the test values, fills the counters, and prints them.

```
int main()
{
   const int FACES = 6;
   int counters[FACES + 1];
   const int NUMBER_OF_TOSSES = 12;
   int tosses[NUMBER_OF_TOSSES];

   generate_test_values(tosses, NUMBER_OF_TOSSES);
   count_values(tosses, NUMBER_OF_TOSSES, FACES, counters);
   print_counters(FACES, counters);
   return 0;
}
```

Here is the complete program, `ch06/dice.cpp`:

```
#include <iostream>

using namespace std;

/**
   Generates a sequence of die toss values for testing.
   @param values the array to be filled with die toss values
   @param size the size of the values array
*/
void generate_test_values(int values[], int size)
{
   int next = 1;
   for (int i = 0; i < size; i++)
   {
      values[i] = next;
      next++;
      if (next == 6) { next = 1; }
   }
}

/**
   Counts the number of times each value occurs in a sequence of die tosses.
   @param values an array of die toss values. Each element is >= 1 and <= faces.
   @param size the size of the values array
   @param faces the number of faces of the die
   @param counters an array of counters of length faces + 1. counters[j]
   is filled with the count of elements of values that equal j. counters[0] is
   not used.
*/
void count_values(int values[], int size, int faces, int counters[])
{
   for (int j = 1; j <= faces; j++) { counters[j] = 0; }
   for (int i = 0; i < size; i++)
   {
      int value = values[i];
      counters[value]++;
   }
```

```
}

/**
   Prints a table of die value counters.
   @param faces the number of faces of the die
   @param counters an array of counters of length faces + 1.
   counters[0] is not printed.
*/
void print_counters(int faces, int counters[])
{
   for (int j = 1; j <= faces; j++)
   {
      cout << j << ": " << counters[j] << endl;
   }
}

int main()
{
   const int FACES = 6;
   int counters[FACES + 1];
   const int NUMBER_OF_TOSSES = 12;
   int tosses[NUMBER_OF_TOSSES];

   generate_test_values(tosses, NUMBER_OF_TOSSES);
   count_values(tosses, NUMBER_OF_TOSSES, FACES, counters);
   print_counters(FACES, counters);
   return 0;
}
```