WORKED EXAMPLE 6.1

Rolling the Dice

© ktsimage/iStockphoto.

Problem Statement Your task is to analyze whether a die is fair by counting how often the values 1, 2, ..., 6 appear. Your input is a sequence of die toss values, and you should print a table with the frequencies of each die value.

Step 1 Decompose your task into steps.

Our first try at decomposition simply echoes the problem statement:

Read the die values into an array.
Count how often the values 1, 2, ..., 6 appear in the array.
Print the counts.

But let's think about the task a little more. This decomposition suggests that we first read and store all die values. Do we really need to store them? After all, we only want to know how often each face value appears. If we keep an array of counters, we can discard each input after incrementing the counter.

This refinement yields the following outline:

For each input value i
  Increment the counter corresponding to i.
Print the counters.

Step 2 Determine which algorithm(s) you need.

We don't have a ready-made algorithm for reading inputs and incrementing a counter, but it is straightforward to develop one. Suppose we read an input into `value`. This is an integer between 1 and 6. If we have an array `counters` of length 6, then we simply call

```
counters[value - 1]++;
```

Alternatively, we can use an array of seven integers, "wasting" the element `counters[0]`. That trick makes it easier to update the counters. When reading an input value, we simply execute

```
counters[value]++; // value is between 1 and 6
```

That is, we declare the array as

```
const int FACES = 6;
int counters[FACES + 1];
```

Why introduce a `FACES` variable? Suppose you later changed your mind and wanted to investigate 12-sided dice:

© Ryan Ruffatti/iStockphoto.

Then the program can simply be changed by setting `FACES` to 12.

The only remaining task is to print the counts. A typical output might look like this:

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

We haven't seen an algorithm for this exact output format. It is similar to the basic loop for printing all elements:

```
for (int i = 0; i <= FACES; i++)
{
   cout << counters[j] << endl;
}
```

However, that loop displays the unused `0` entry, and it does not show the corresponding face values. We adapt the algorithm as follows:

```
for (int i = 1; i <= FACES; i++)
{
   cout << j << ": " << counters[j] << endl;
}
```

Step 3 Use functions to structure your program.

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:

```
/**
   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 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[])
```

The function modifies the `counters` array. This is not a problem because 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 of length faces + 1
*/
void print_counters(int faces, int counters[])
```

Step 4 Assemble and test the program.

The listing at the end of this section shows the complete program.

The following table shows test cases and their expected output. To save space, we only show the counters in the output.

Test Case	Expected Output	Comment
1 2 3 4 5 6	1 1 1 1 1 1	Each number occurs once.
1 2 3	1 1 1 0 0 0	Numbers that don't appear should have counts of zero.
1 2 3 1 2 3 4	2 2 2 1 0 0	The counters should reflect how often each input occurs.
(No input)	0 0 0 0 0 0	This is a legal input; all counters are zero.
0 1 2 3 4 5 6 7	**Error**	Each input should be between 1 and 6.

Here's the complete program:

worked_example_1/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 i = 1; i <= faces; i++) { counters[i] = 0; }
```

```
33    for (int j = 0; j < size; j++)
34    {
35       int value = values[j];
36       counters[value]++;
37    }
38 }
39
40 /**
41    Prints a table of die value counters.
42    @param faces the number of faces of the die
43    @param counters an array of counters of length faces + 1.
44    counters[0] is not printed.
45 */
46 void print_counters(int faces, int counters[])
47 {
48    for (int i = 1; i <= faces; i++)
49    {
50       cout << j << ": " << counters[j] << endl;
51    }
52 }
53
54 int main()
55 {
56    const int FACES = 6;
57    int counters[FACES + 1];
58    const int NUMBER_OF_TOSSES = 12;
59    int tosses[NUMBER_OF_TOSSES];
60
61    generate_test_values(tosses, NUMBER_OF_TOSSES);
62    count_values(tosses, NUMBER_OF_TOSSES, FACES, counters);
63    print_counters(FACES, counters);
64    return 0;
65 }
```