argument. We want \( n \) to have a default value of 1, so we assign that value to \( n \). A default argument value is an initialization value. Thus, the prototype above initializes \( n \) to the value 1. If you leave \( n \) alone, it has the value 1, but if you pass an argument, the new value overwrites the 1.

When you use a function with an argument list, you must add defaults from right to left. That is, you can't provide a default value for a particular argument unless you also provide defaults for all the arguments to its right:

```cpp
int harpo(int n, int m = 4, int j = 5); // VALID
int chico(int n, int m = 6, int j);    // INVALID
int groucho(int k = 1, int m = 2, int n = 3); // VALID
```

The \texttt{harpo()} prototype, for example, permits calls with one, two, or three arguments:

```cpp
beeps = harpo(2);       // same as harpo(2,4,5)
beeps = harpo(1,8);     // same as harpo(1,8,5)
beeps = harpo(8,7,6);   // no default arguments used
```

The actual arguments are assigned to the corresponding formal arguments from left to right; you can't skip over arguments. Thus, the following isn't allowed:

```cpp
beeps = harpo(3, ,8);   // invalid, doesn't set m to 4
```

Default arguments aren't a major programming breakthrough; rather, they are a convenience. When you get to class design, you'll find they can reduce the number of constructors, methods, and method overloads you have to define.

\textbf{Listing 8.7} puts default arguments to use. Note that only the prototype indicates the default. The function definition is the same as it would have been without default arguments.

\textbf{Listing 8.7} left.cpp

// left.cpp -- string function with a default argument
#include <iostream>
using namespace std;
const int ArSize = 80;