MORE ABOUT FUNCTION PARAMETERS

Default Values for Parameters in Functions

Problem. We wish to construct a function that will evaluate any real-valued polynomial function of degree 4 or less for a given real value x. The general form of such a polynomial is

$$a + bx + cx^2 + dx^3 + ex^4$$

where the coefficients a, b, c, d, and e are real constants.

One Solution. The function in Code 1 below is one way to solve the problem. Its specification is:

Receive: The real values a, b, c, d, e, and x
Return: The real value $a + bx + cx^2 + dx^3 + ex^4$

It simply returns the value of an appropriate C++ expression for $a + bx + cx^2 + dx^3 + ex^4$.

Code 1. Evaluating Polynomials — Version 1.

```
/* Polynomial will evaluate any polynomial up to degree 4.
 *
 * Receive: the real values x, a, b, c, d, and e
 * Return:  the real value a + bx + cx^2 + dx^3 + ex^4
 ****************************************************************/

#include <cmath>

double Polynomial(double x, double a, double b,
                  double c, double d, double e)
{
   return a + b*x + c*pow(x, 2.0) + d*pow(x, 3.0) + e*pow(x, 4.0);
}
```

The Difficulty. The difficulty with `Polynomial()` is that to make `Polynomial()` a (relatively) general function, we used six parameters. In applications that involve lower-degree polynomials, however, this generality becomes a nuisance, because we must pass arguments for all of the parameters each time we call `Polynomial()`. For example, to calculate the value of

$$P(x) = 4 + 4x + x^2$$

when x is 12, we would use the call

```
yVal = Polynomial(12.0, 4.0, 4.0, 1.0, 0.0, 0.0);
```

and to compute the value of the linear function

$$Q(x) = 3 - 4x$$

when x is 7, we would use the call:

```
yVal = Polynomial(7.0, 3.0, -4.0, 0.0, 0.0, 0.0);
```

The Solution. To avoid this inconvenience, C++ allows the programmer to specify a **default value** for a parameter. If a function with default-valued parameters is called and no arguments are passed to these parameters, they receive their specified default values.

To provide a default value for a parameter, we simply use the assignment operator (`=`) to give the parameter its default value, usually in the prototype of the function. For example, suppose we change the heading of `Polynomial()` to

```
double Polynomial(double x, double a = 0.0, double b = 0.0,
                  double c = 0.0, double d = 0.0, double e = 0.0)
```

Now if we need to compute the value

$$P(x) = 4 + 4x + x^2$$

when x is 12, we can use the call:

```
yVal = Polynomial(12.0, 4.0, 4.0, 1.0);
```

and since only four arguments are passed, the last two parameters (`d` and `e`) are given their default values (0.0). Similarly, to compute the function

$$Q(x) = 3 - 4x$$

when x is 8, we can use the call

```
yVal = Polynomial(8.0, 3.0, -4.0);
```

and since only three arguments are passed, the last three parameters (`c`, `d`, and `e`) will be given their default values.

Limitations in Using Default Parameter Values. There are several restrictions that C++ imposes on the use of parameters with default values:

1. *Default values for parameters of a function can be given only once.* The practical implication of this rule is that *a parameter should be given a default value in the prototype of the function, or in its definition, but not both.*
2. *If the default parameter values are given in the prototype of the function, and*

that prototype is stored in the header file of a library, then any program that uses the `#include` *directive to insert that header file can use the default values.* This is the recommended approach.

3. *If the default values are given in the definition of the function, and the function's definition is stored in the implementation file of a library, then the default values cannot be used by a program that inserts the header file of that library (using the* `#include` *directive)* . This is because a program that uses a library (by inserting its header file) "sees" the prototypes in the header file of the library but never "sees" the function definitions in the implementation file. Consequently, we suggest that the default values be given in the function prototype, since otherwise the default parameter values cannot be used outside the library.

4. *If an argument is supplied for a parameter p that has a default value, then an argument must be supplied for every parameter that precedes p in the parameter list.* For example, suppose that we wish to evaluate

$$P(x) = 2 + 3x^2$$

when x is 1. Then we are unable to take advantage of the default value of parameter `b` and must use the expression:

```
Polynomial(1.0, 2.0, 0.0, 3.0)
```

to call `Polynomial()`. The reason for this is that in determining which argument goes with which parameter, C++ matches from left to right, associating the first argument with the first parameter, the second argument with the second parameter, and so on. Thus, the expression:

```
Polynomial(1.0, 2.0, 3.0)
```

evaluates the polynomial

$$2 + 3x$$

instead of the polynomial we intended. Note that making this mistake results in a *logic* error, not a *syntax* error. The compiler will process such calls without generating an error.

5. *Parameters that receive default values must be declared at the end of the parameter list.* Stated differently, a parameter that does not receive a default value must precede all parameters that do. For example, a syntax error results if we try to declare `Polynomial()` as

```
double Polynomial(double x, double a = 0.0, double b,
                  double c = 0.0, double d = 0.0, double e = 0.0);
```

because parameter `a,` which has a default value, precedes parameter `b`, which has no default value. Again, this rule makes sense, since otherwise a call like

```
Polynomial(3.0, 2.0, 1.0);
```

is ambiguous because it is not clear which default values the programmer intended to use. While `x` is clearly meant to be 3.0, the programmer could

have intended that a be 2.0 and b be 1.0; or that a be 0.0, b be 2.0 and c be 1.0; or that a be 0.0, b be 2.0, c be 0.0, and d be 1.0; and so on.

Varying the Number of Arguments in Functions

The default parameter value mechanism allows us to call a function with *fewer* arguments than the specified number of parameters. Now, we consider the problem of constructing a function that can be called with *more* arguments than the number specified. To illustrate, consider the following generalization of the preceding polynomial problem.

Problem. We wish to construct a function that will evaluate a polynomial of degree *n*, for any positive integer *n.*

Solution. A specification for the function is

Receive: the *degree* of the polynomial
x, the value at which the polynomial is to be evaluated
a, the constant term in the polynomial
the coefficients of higher-order terms (if any)

Return: The value of the polynomial at *x*

To compute the value of the polynomial, we can use the following algorithm:

Algorithm for Evaluating a Polynomial

1. Initialize *polyValue* to *a*, and *power_of_x* to 1.0.
2. For each value *i* in the range 1 through *degree:*
   a. Get the *i*th coefficient of the polynomial, storing it in *nextCoef.*
   b. Multiply *power_of_x* by *x*.
   c. Multiply *nextCoef* by *powerOfX* and add the product to *polyValue.*
3. Return the value of *polyValue.*

Coding. To code the function, we need a mechanism for *passing different numbers of arguments to the function,* depending on the degree of the polynomial we want to use. We need a function in which there is no limit on the number of arguments. A stub for a function that accomplishes this is as follows:

```
#include <cstdarg>   // declarations to permit
                     // varying numbers of arguments

double Polynomial(int degree, double x, double a, ...)
{
}
```

The declarations in the `stdarg` library allow the use of **ellipses** (. . .) within the parameter list to inform the C++ compiler that if this function is called with more than three arguments, the extra parameters should not be treated as errors. The compiler thus "turns off" argument-checking when processing such calls. It is left to programmers that use this ellipses mechanism to ensure that the number and type of arguments are correct.

When a function whose parameter list includes ellipses is called, any additional arguments that are present are placed into a special type of list, called a

va_list(for varying-argument list). The type va_list and the operations for manipulating it are declared in the library stdarg, so its header file must be inserted (using the #include directive) before the function definition. There are three basic operations for manipulating a varying-argument list. In the following descriptions of these operations, *list* is of type va_list:

- va_start(*list, lastParam*): initializes *list* for processing; *lastParam* is the name of the last parameter in the function declaration.
- va_arg(*list, type*): retrieves and returns the next value of the specified *type* from *list* (assuming *list* has been initialized with va_start()).
- va_end(*list*): "cleans up" *list* after processing is completed.

This type va_list and the preceding operations make it possible to implement the polynomial-evaluation algorithm as shown in the following function.

Code 2. Evaluating Polynomials — Version 2.

```
/* Polynomial will evaluate a polynomial of any degree.
 *
 * Receive: the int degree, a real value x, and the real
 *          coefficients a, ... of a polynomial
 * Return:  the real value of the polynomial at x
 ****************************************************************/

#include <cstdarg>

double Polynomial(int degree, double x, double a, ...)
{
   double
      power_of_x = 1.0,                    // powers of x
      nextCoef,                            // next coefficient
      polyValue = a;                       // polynomial's value at x

   va_list argList;

   va_start(argList, a);                   // argList begins after a

   for (int i = 1; i <= degree; i++)
   {
      power_of_x *= x;                     // i-th power of X
      nextCoef = va_arg(argList, double); // get the ith coefficient

      polyValue += nextCoef * power_of_x; // ith term of polynomial
   }

   va_end(argList);                        // clean up the list

   return polyValue;
}
```

The following is a driver program that tests this function for polynomials of degree ≤ 3.

Code 3. A Driver Program for Polynomial().

```
/* polytester.cpp is a driver program to test function Polynomial().
 *
 * Output: the value of Polynomial() for polynomials of various degrees
 ************************************************************************/

#include <iostream>
using namespace std;

double Polynomial(int degree, double x, double a, ...);

int main()
{
   cout // P(1.0) for P(x) = 2
        << Polynomial(0, 1.0, 2.0) << endl

        // P(1.0) for P(x) = 2 + 3x
        << Polynomial(1, 1.0, 2.0, 3.0) << endl

        // P(1.0) for P(x) = 2 + 3x + 4x^2
        << Polynomial(2, 1.0, 2.0, 3.0, 4.0) << endl

        // P(1.0) for P(x) = 2 + 3x + 4x^2 + 5x^3
        << Polynomial(3, 1.0, 2.0, 3.0, 4.0, 5.0) << endl;

   return 0;
}

/* Insert the #include directive and the definition
   of function Polynomial() from Code 2 here.  */
```

Sample run:

```
2
5
9
14
```

Note that it is the programmer's responsibility to ensure that `Polynomial()` is called correctly. If we were to erroneously pass `int` values instead of `double` values to `Polynomial()`,

```
Polynomial(4, 1, 2, 3, 4, 5)
```

then the arguments would be stored as `int` (instead of `double`) values within the `va_list`. If `int` values are stored in one memory word and `double` values in two

memory words, then the first call to `va_arg()`, `va_arg(argList, double)`, would interpret the two (`int`) words storing 3 and 4 as a `double` value, and consequently return an erroneous result. This is one situation where different types of numeric data cannot be freely intermixed: If a function that uses this ellipses mechanism is expecting a series of arguments of a particular type, then it must receive arguments of that type.