Chapter 5

Algorithms and Their Object-Oriented Implementation

In this chapter, we discuss computational problems, solution methods, and their efficient implementation. We describe different approaches to writing algorithms to solve a particular problem and compare their storage and computation requirements. The abstract objects used here help not only to implement algorithms but also to develop them in the first place and modify them later on if necessary. We illustrate these points in the implementation of the "polynomial" object, along with useful arithmetic operations and functions. This object is particularly useful in high-order finite elements, discussed later in the book.

5.1 Ideas and Their Implementation

The programmer has an idea in mind of how to complete a particular task. This idea can usually be expressed in words in a natural language such as English. This formulation makes the idea clearer and more practical; indeed, when you explain your idea to a friend or colleague, it becomes clearer and more concrete to you too.

Describing the idea in words, however, is usually insufficient from a practical point of view. A more useful description must take the form of a list of operations that can be carried out one by one to produce the required solution to the problem under consideration. This is called an algorithm.

The individual operations in the algorithm are still written in a natural language, which may be somewhat vague and ambiguous. In fact, the interpretation of the terms in the natural language may well depend on the context in which they are used. Therefore, the algorithm must contain only unambiguous (context-free) instructions that can be carried out by a human being or a machine.

When the idea is about how to solve a computational problem, it can often be written in context-free formal language, using mathematical symbols, structures, and objects. This is really the best way to formulate the idea, because then the algorithm derived from it can also be written in terms of unambiguous mathematical instructions.

The original idea is useless unless it can be communicated to people and machines. While humans can usually understand it in (context-dependent) natural language, machines must have an explicit algorithm, written in context-free formal language. Translating the idea

into a formal algorithm is also helpful for the developers themselves, because it gives them the opportunity to debug it and check whether or not it indeed does what it is supposed to do.

The most important step in using the idea is, thus, to formulate it in mathematical language. This is where C++ comes to our aid: it provides the framework for defining the required mathematical objects. These objects may serve as words in the formal language in which the algorithm is written. Writing the original idea and algorithm in a high-level programming language like C++ is called "implementation."

Actually, the object-oriented approach not only helps to implement the idea and algorithm in their original spirit but also provides the terms and objects required to think about them and develop them in the first place. By introducing useful terms and objects, it provides the required vocabulary to develop, express, and reshape the original raw idea, until it ripens to its final form.

5.2 Multilevel Programming

In a high-level programming language, objects such as characters and numbers are available. One can define variables that may take different values and manipulate them with unary and binary operations and functions. These elementary objects, however, are usually insufficient for implementing practical algorithms. In fact, even for elementary algorithms, abstract mathematical objects are needed. Although these objects can in theory be implemented using characters only, as in the original Turing machine, this approach is, of course, highly impractical. Traditional programming languages that use arrays are of course better, but the implementation of complex data structures in them may still be too complicated and hard to read, use, and modify if necessary. A programming language with a high level of abstraction is clearly necessary to implement complex mathematical structures.

With an object-oriented language like C++, one can implement sophisticated algorithms using objects that are unavailable in the standard language. This is called "high-level programming." In this approach, the programmer assumes that the required objects and functions that manipulate them are available. The actual implementation of these objects and functions can be delayed to a later stage, called "low-level programming."

High-level programming requires a good understanding of the algorithm and concepts and ideas behind it. The programmer who writes the high-level code should write footnotes about what objects exactly are needed and what functions should be available to manipulate them. These requirements should then be passed to a colleague who is experienced in low-level programming. The programmer of the high-level code can now continue implementing the algorithm in its original spirit, without being distracted by the details of the implementation of the objects used for this purpose.

Low-level programming may require knowledge and experience in computer hardware and memory. The programmer who does it should implement the objects as efficiently as possible according to the requirements passed on from the programmer of the high-level code.

The two programmers can thus work rather independently. The programmer of the high-level code can concentrate on realizing the true meaning of the original algorithm, having every abstract object available, while the programmer of the low-level code can concentrate on the optimal storage-allocation strategy to implement the objects.

The above workplan is ideal. In practice, interaction between the two programmers is required, especially in complex numerical applications, where the efficiency requirements in the low-level programming may put constraints on the high-level programming. Still, dividing the project into high-level and low-level tasks is helpful as a starting point to help organize the work on the entire project.

This method of work is called two-level programming. It is suitable not only for a team with members who have different levels of knowledge and expertise but also for a single programmer who must do the entire project. This programmer can still benefit from dividing the job into high-level and low-level tasks and working on each of them separately with full attention.

If the low-level programming is done properly and the required objects are well implemented and sufficiently general, then they can be used in many algorithms and applications. Actually, the programmer of the low-level code can start implementing objects that are commonly used even before having specific requirements, thus creating a library of objects for future use. The objects should be well documented to make them useful for potential users, to implement more complex objects and create more advanced libraries. The process may continue, with higher and higher levels of programming that use objects from lower levels to define new ones. This is called multilevel programming; it actually contributes to the development of the standard programming language by introducing more and more objects ready for future use.

5.3 Information and Storage

Each step in the computational algorithm requires data to complete a particular instruction. These data can be either calculated or fetched from the computer memory. Obviously, data that have already been calculated should be stored for future use unless recalculation is cheaper than fetching.

Surprisingly, it often is. Storing and fetching can be so expensive and slow that it is no longer worth it. Furthermore, it involves the extra effort of allocating sufficient memory for new variables and giving them meaningful names to remind the user what kind of data they contain. All this can make the code less elegant and harder to read and debug.

One of the great advantages of C is the opportunity to use functions that return the required result, be it a number or a pointer to a sequence of numbers. This way, one can use a function to recalculate data rather than fetch it from memory, provided that this recalculation is not too expensive.

This feature is made yet more elegant in C++, where functions may take and return actual objects rather than mere pointers. The high-level programming that uses such functions is thus much more transparent and clear, because it avoids dealing with pointers or addresses in the computer memory. Surely, a function that takes and returns objects is far more useful and transparent than a function that takes and returns arrays of numbers.

C++ functions, however, may be slightly slower than the corresponding C functions, because the returned objects may be constructed by an extra call to the constructor of the class. For example, a C++ function may define and use a local object that contains the required result, but then, in order to be returned as output, this object must be copied (by the copy constructor of the class) to a temporary external object to store the result after the local

object has vanished. Still, this slight overhead is well worth it for the sake of transparent and useful code. Furthermore, some C++ compilers support versions that reduce this overhead to a minimum.

In what follows, we'll illustrate the effectiveness of C++ in implementing the polynomial object and related algorithms.

5.4 Example: The Polynomial Object

Here, we show how convenient it is to implement polynomials as objects in C++. The object-oriented approach gives one the opportunity to define functions that take and return objects rather than pointers or arrays, thus avoiding the need to deal with details of storage. Furthermore, this approach enables the definition of useful arithmetic operations between polynomials, such as addition, multiplication, etc.

We mainly consider two common problems: the multiplication of two polynomials and the calculation of the value of a polynomial at a given point.

Consider the polynomial

$$p(x) \equiv \sum_{i=0}^{n} a_i x^i,$$

where x is the independent variable, n is the degree of the polynomial (maximal power of x), and $a_0, a_1, \ldots, a_n$ are the coefficients.

The first question is how to store the polynomial. To answer this question, observe that the polynomial $p(x)$ is characterized by its coefficients $a_0, a_1, \ldots, a_n$. Thus, to store a polynomial, it is sufficient to store its coefficients.

In C, one would naturally store the coefficients in an array; but then again, passing them to a function involves getting into details of storage and distracts the programmer from the mathematical algorithms.

It is far more efficient to do this in C++, using, e.g., the "list" object in Chapter 3, Section 4. Indeed, the "polynomial" class can be derived from a list of numbers, so the "polynomial" object is actually a list of coefficients. This object can then be passed easily to functions by reference as usual, regardless of its internal implementation. Furthermore, "polynomial" objects can also be returned as output from functions and used further as input in subsequent calls to other functions.

Because the type of x and the a_i's is not yet specified, we use another powerful tool available in C++: templates. This way, the parameter 'T' in the "polynomial" template class stands for the type of independent variable and coefficients and can be used in the definition of the function. Because the particular type is immaterial in the algorithm used in the function, the template also means that the type can be disregarded and the mathematical concepts in the algorithm can be focused on. The concrete type substituted for 'T' will be specified later, when the compiler encounters calls to functions that use "polynomial" objects. These functions can then be called for polynomials of the specified type: integer, real, complex, etc.

Here is the code that derives the "polynomial" class from the base "list" class (Figure 5.1):

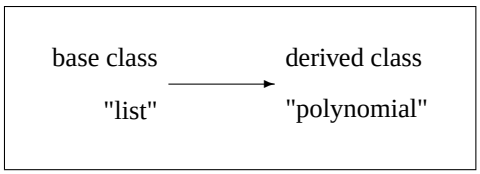

Figure 5.1. *Schematic representation of inheritance from the base class "list" to the derived class "polynomial".*

```
template<class T> class polynomial:public list<T>{
  public:
    polynomial(int n=0){
      number = n;
      item = n ? new T*[n] : 0;
      for(int i=0; i<n; i++)
        item[i] = 0;
    }  //  constructor

    polynomial(int n, const T&a){
      number = n;
      item = n ? new T*[n] : 0;
      for(int i=0; i<n; i++)
        item[i] = new T(a);
    }  //  constructor with 'T' argument
```

These constructors first implicitly invoke the default constructor of the base "list" class, which constructs a trivial list with no items in it. Thanks to the fact that the "item" field in the base "list" class is declared "protected" rather than private, the above constructors can access and reconstruct it to contain meaningful coefficients.

The copy constructor and assignment operator don't have to be defined, because the corresponding operators in the base "list" class do the right thing, that is, copy or assign the items in the argument one by one to the current object.

The following destructor also needs to do nothing, because the destructor of the base "list" class (invoked implicitly at the end of it) destroys first the individual items in the underlying list and then the list itself, as required:

```
    ~polynomial(){
    }  //  destructor
```

The following member function returns the degree of the polynomial:

```
    int degree() const{
      return number-1;
    }  //  degree of polynomial
};
```

This concludes the block of the "polynomial" class.

Next, we also implement operators that add two polynomials. In particular, the "+=" operator takes two "polynomial" arguments and adds the second one to the first one:

```
template<class T>
const polynomial<T>&
operator+=(polynomial<T>& p, const polynomial<T>&q){
  if(p.size() >= q.size())
    for(int i=0; i<q.size(); i++)
      p(i) += q[i];
  else{
    polynomial<T> keepQ = q;
    p = keepQ += p;
  }
  return p;
}  //  add polynomial
```

This operator works as follows. If the degree of 'p' is larger than or equal to the degree of 'q', then the coefficients in 'q' are added one by one to the corresponding coefficients in 'p'. In the addition of individual components, the "operator()" inherited from the base "list" class is used to change the item "p(i)" on the left, whereas the "operator[]" is used to read the item "q[i]" on the right.

If, on the other hand, the degree of 'p' is smaller than that of 'q', then the above algorithm is no longer applicable. Instead, an inner call to the "+=" operator is made, with the roles of 'p' and 'q' interchanged.

Because the "+=" operator is implemented here as a nonmember function, its first (nonconstant) argument cannot be a temporary variable returned as output from another function. Indeed, the compiler won't accept such a call, because it makes no sense to change a temporary object that is going to disappear soon anyway. This is why the extra "polynomial" object "keepQ" is defined and passed as the first argument to the inner call to the "+=" operator.

The above "+=" operator is now further used in the '+' operator, which returns the sum of two polynomials:

```
template<class T>
const polynomial<T>
operator+(const polynomial<T>& p,
        const polynomial<T>&q){
  polynomial<T> keep = p;
  return keep += q;
}  //  add two polynomials
```

Next, we implement the multiplication of a polynomial by a scalar. The "*=" operator takes two arguments, a polynomial and a scalar, and multiplies the first by the second:

```
template<class T>
const polynomial<T>&
operator*=(polynomial<T>& p, const T&a){
  for(int i=0; i<p.size(); i++)
    p(i) *= a;
  return p;
}  //  multiplication by scalar
```

The above "*=" operator is now used in the '*' operator that returns the product of a scalar and a polynomial:

```
template<class T>
const polynomial<T>
operator*(const T&a, const polynomial<T>&p){
  polynomial<T> keep = p;
  return keep *= a;
}  //  scalar times polynomial
```

Once the "polynomial" object, along with its arithmetic operations and other useful functions, is properly implemented as above, it can be placed in a library of objects for further use in high-level programming, such as the implementation of high-order finite elements (Chapter 15). Because the details of implementation are well hidden in the private part of the "polynomial" class, the implementation can later be changed if necessary, without affecting the high-level codes and with no need to debug them again. For example, if one deals mostly with sparse polynomials with only a few nonzero coefficients a_i, then it makes sense to store only these nonzero coefficients in a connected list (as in Chapter 16, Section 3) rather than a list. The low-level programming required for this change has absolutely no effect on codes that use "polynomial" objects, provided that the interface remains the same; that is, the reimplemented functions must still take and return the same arguments as before.

Here, we are not particularly interested in sparse polynomials, so we stick to our original implementation of a polynomial as a list of coefficients. The reason for this will become clear in Section 5.13.

5.5 Multiplication of Polynomials

Let us now consider the problem of multiplying two polynomials. Let $q(x)$ be the polynomial

$$q(x) \equiv \sum_{i=0}^{k} b_i x^i.$$

The product polynomial pq is given by

$$
\begin{aligned}
(pq)(x) &\equiv p(x)q(x) \\
&= \sum_{i=0}^{n}\sum_{j=0}^{k} a_i b_j x^{i+j} \\
&= \sum_{m=0}^{n+k}\left(\sum_{i+j=m} a_i b_j\right) x^m \\
&= \sum_{m=0}^{n+k}\left(\sum_{j=\max(0,m-n)}^{\min(m,k)} a_{m-j} b_j\right) x^m .
\end{aligned}
$$

Thus, the required polynomial pq is of degree $n+k$, and its coefficients are given in the above formula in terms of the coefficients of p and q.

Once the polynomials are implemented as objects, the above formula can be used to define the "operator*()" that takes two polynomials p and q and produces their product pq:

```
template<class T>
polynomial<T>
operator*(const polynomial<T>&p,const polynomial<T>&q){
  polynomial<T> result(p.degree()+q.degree()+1,0);
  for(int i=0; i<result.size(); i++)
    for(int j=max(0,i-q.degree());
        j<=min(i,p.degree()); j++){
      if(j == max(0,i-q.degree()))
        result(i) = p[j] * q[i-j];
      else
        result(i) += p[j] * q[i-j];
    }
  return result;
}  //  multiply two polynomials
```

The above '*' operator is also used to define the "*=" operator:

```
template<class T>
polynomial<T>&
operator*=(polynomial<T>&p, const polynomial<T>&q){
  return p = p * q;
}  //  multiply by polynomial
```

The following program defines a polynomial with three coefficients of value 1, namely, $p(x) = 1 + x + x^2$. Then, it calls the '*' operator to produce the polynomial p^2 and uses the "print()" function in the base "list" class to print it onto the screen:

```
int main(){
  polynomial<double> p(3,1);
  print(p * p);
  return 0;
}
```

5.6 Calculation of a Polynomial

Another common task that involves the polynomial p is to calculate its value $p(x)$ at a given point x. Here, we also benefit from the present implementation of the polynomial object, which gives us the opportunity to pass it to the function as a whole, without bothering with storage details.

Here is the function that takes the polynomial p and the point x and returns $p(x)$:

```
template<class T>
const T
calculatePolynomial(const polynomial<T>&p, const T&x){
  T powerOfX = 1;
  T sum=0;
  for(int i=0; i<p.size(); i++){
    sum += p[i] * powerOfX;
    powerOfX *= x;
  }
  return sum;
}  //  calculate a polynomial
```

Note that we have used here the local variable "powerOfX" to store the powers x^i used in the polynomial. This extra variable slightly reduces the elegance of the code. In what follows, we'll see an improved algorithm that is not only more efficient but also more elegantly implemented.

5.7 Algorithms and Their Implementation

So far, we have discussed mostly implementation issues. We mentioned that object-oriented languages such as C++ give us the opportunity to divide the entire project into two parts: a low-level part, where elementary objects are implemented, and a high-level part, where these objects are actually used to implement the algorithm in its original spirit. This two-level approach also has the advantage that the well-implemented objects can be used not only in the present algorithm but also in other algorithms and applications. Actually, the low-level part of the project could extend to create an entire library of objects, ready for future use. Furthermore, the hierarchy of the libraries could be built one on top of the other, each of which uses objects from lower libraries to form more sophisticated objects. This is called multilevel programming.

The low-level part of the code, where frequently used objects are implemented, should be particularly efficient in terms of memory allocation and data access. The high-level part, on the other hand, where the mathematical algorithm is implemented, should be as efficient as possible in terms of operation count. Even more importantly, it should be modular, transparent, and reader friendly, to aid not only potential readers but also the programmer in the process of writing, debugging, and modifying if necessary.

Although efficiency is an important property, transparency and clarity may be even more important to guarantee the correctness and usefulness of the code. The objects, in particular, should be complete and ready to use in every future application. In particular, storage details should be hidden from the users, who should remain completely unaware of

the internal structure of the objects and know them only by their mathematical properties, available through interface functions.

Moreover, we would even recommend compromising efficiency for the sake of transparency and clarity if necessary. For example, a function that returns an object uses an extra call to the copy constructor of the class to construct the temporary output object. Although this extra call reduces efficiency, it may be well worth it to make the code clearer. Indeed, the returned object may be further used as input in other calls to other functions in the same code line. Returning an object is far more appropriate than returning an array of numbers with no apparent interpretation as a complete object.

Most often, there is mutual interaction between the high-level and low-level programmers. The high-level programmer may pass requirements to the low-level programmer, but also receive feedback about how realistic the requirements are, and what the limits and drawbacks are in the required objects. This feedback may lead the high-level programmer to modify the original implementation and adapt it to hardware issues and limits.

Actually, even the developer of a mathematical algorithm is not entirely independent. An algorithm may look efficient in terms of theoretical operation count but require objects that are too hard to implement. Thus, although the algorithm is usually developed in a purely mathematical environment, more practical implementation issues shouldn't be ignored. (See also Chapter 18, Section 14.)

The first concern of the algorithm developer is, of course, to reduce the theoretical operation count. Fortunately, it turns out that algorithms that are efficient in this theoretical sense are often also straightforward and efficient in terms of practical implementation.

One of the common principles of efficient algorithm is to avoid recalculating data that can be easily fetched from the memory. In the next section, we'll see an algorithm that uses this principle in the present problem of calculating the value of a polynomial at a given point. We'll also see that this algorithm indeed has a straightforward and efficient implementation.

5.8 Horner's Algorithm

The first rule in efficient calculation is as follows:

Don't open parentheses unless absolutely necessary!

Indeed, the distributive law says:

$$A(B + C) = AB + AC,$$

where A, B, and C are members of some mathematical field. Now, the right-hand side, where no parentheses are used, requires two multiplications and one addition to calculate, whereas the left-hand side, where parentheses are used, requires only one addition and one multiplication. This is also the idea behind Horner's algorithm for calculating the value of a polynomial. In fact, this algorithm introduces in the polynomial as many parentheses as possible.

The polynomial in Section 5.6 is actually calculated directly from the formula

$$p(x) = \sum_{i=0}^{n} a_i x^i.$$

Because the right-hand side contains no parentheses, it requires $2n$ multiplications (n multiplications to calculate the powers x^i, and another n multiplications to multiply them by the coefficients a_i) and n additions. Can this number be reduced?

Yes, it can. The Horner algorithm uses the following formula:

$$p(x) = (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})\cdots)x + a_0.$$

The process starts from the innermost parentheses, where the term with coefficient a_n is calculated, and progresses gradually to the outer ones, to which the free coefficient a_0 is added at the end. Because of the large number of parentheses, the algorithm requires only a total of n multiplications and n additions and is also implemented nicely as follows:

```
template<class T>
const T
HornerPolynomial(const polynomial<T>&p, const T&x){
  T result = p[p.degree()];
  for(int i=p.degree(); i>0; i--){
    result *= x;
    result += p[i-1];
  }
  return result;
}  //  Horner algorithm to calculate a polynomial
```

The algorithm is not only more efficient but also more efficiently implemented. Indeed, the code is slightly shorter than the code in Section 5.6 and avoids the extra variable "powerOfX" used there.

In the next section, we use the above algorithms to efficiently calculate the value of the single power x^n at a given point x. It turns out that, for this purpose, although Horner's algorithm is slightly less efficient, it is far more elegant and easy to understand and debug. This tradeoff between efficiency and transparency arises often not only in simple examples such as this one but also in complex applications, where elegance and transparency are crucial for the sake of well-debugged and useful code.

5.9 Calculation of a Power

Usually, a polynomial is calculated for a real or complex argument x. Here, however, we show that the concept of a polynomial with an integer argument is also helpful. Actually, this polynomial is never calculated explicitly, because its value is already available. Still, the actual representation of an integer number as a polynomial (as in Chapter 1, Section 18) helps to solve the present problem.

Consider the following problem: for a given x and a large integer n, calculate x^n efficiently. Of course, this can be done using n multiplications in the recursive formula

$$x^n = x \cdot x^{n-1}$$

(see Chapter 1, Section 17), but can it be done more efficiently?

The answer is yes, it can be calculated in $2\log_2 n$ multiplications only. For this purpose, we use the binary representation of the integer n as a polynomial in the number 2:

$$n = \sum_{i=0}^{k} a_i 2^i,$$

where the coefficients a_i are either 0 or 1. With this representation, we have

$$x^n = \Pi_{i=0}^{k} x^{a_i 2^i},$$

which is the product of all the $a_i 2^i$-powers of x. Now, the 2^i-power of x can be calculated by i applications of the substitution

$$x \leftarrow x^2.$$

The total cost of the calculation is, thus, at most $2\log_2 n$ multiplications.

The algorithm is implemented nicely as follows:

```
template<class T>
const T
power(const T&x, int n){
  T result = 1;
  T powerOfX = x;
  while(n){
    if(n % 2) result *= powerOfX;
    powerOfX *= powerOfX;
    n /= 2;
  }
  return result;
}  //  compute a power
```

In the body of the "while" loop above, the last digit in the binary representation of n is found by the modulus operation n % 2. Once this digit has been used, it is dropped by dividing n by 2 without residual. With this approach, the code is particularly elegant.

There is, however, an even better approach to calculating the power x^n. This approach is based on Horner's polynomial. Recall that Horner's representation of a polynomial $p(x)$ is actually based on the recursion

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + x p_1(x),$$

where

$$p_1(x) \equiv \sum_{i=0}^{n-1} a_{i+1} x^i$$

is a polynomial of degree $n-1$. In fact, the polynomial $p_1(x)$ can be reformulated recursively in the same way, leading eventually to the representation in Section 5.8.

Of course, the above recursive formulation is never used explicitly due to the large cost of constructing $p_1(x)$ as a "polynomial" object. This is why the code in Section 5.8

actually "opens up" the recursion and starts from the end of it (the innermost parentheses), going back to the beginning in an ordinary loop.

In the present problem, however, where the polynomial $p(2) = n$ is just the binary representation of the integer n, $p_1(2) = n/2$ is immediately available by dividing n by 2 without residual. In fact, we have

$$n = (n \,\%\, 2) \; + \; 2(n/2),$$

where $n/2$ means integer division without residual and $n\%2$ contains the residual (see Chapter 1, Section 18). This implies that

$$x^n = x^{n \,\%\, 2}(x^2)^{n/2}.$$

This leads to the following recursive implementation of the "power()" function:

```
template<class T>
const T
power(const T&x, int n){
  return n ? (n%2 ? x * power(x * x,n/2)
              : power(x * x,n/2)) : 1;
}  //  compute a power recursively
```

This way, the "power()" function contains only one command. This style is particularly efficient, because it avoids explicit definition of local variables. Because the "power()" function is called recursively $\log_2 n$ times, the total cost in terms of operation count is still at most $2 \log_2 n$ multiplications, as before. However, one should also take into account the extra cost involved in the recursive calls. In fact, in each recursive call, the computer needs to allocate memory for the local variables 'x' and 'n'. Although 'x' is passed by reference, its address must still be stored locally in each recursive call. Nevertheless, this extra cost may be well worth it for the sake of short and elegant code. Indeed, the recursive reformulation of n is much shorter and simpler than the original explicit formulation, hence also easier to debug. Although debugging is unnecessary for the present well-tested codes, it may pose a crucial problem in complex applications. This is why writing short and elegant code is a most attractive skill.

5.10 Calculation of Derivatives

In the above discussion, it is assumed that one needs to calculate only the nth power of x, x^n. But what if all powers $x^2, x^3, \ldots, x^n$ are required? In this case, of course, it makes no sense to use the above algorithms. The original approach is much more sensible:

$$x^k = x \cdot x^{k-1}, \quad k = 2, 3, \ldots, n.$$

Indeed, this method computes all the required powers in n multiplications only.

The question raised now is where to store all these calculated powers. In C, one is forced to use an array. But then again, an array is not a meaningful object, and passing it to a function or returning it from a function makes little sense. The "list" object in Chapter 3,

Section 4, is far more suitable for this purpose. Although the advantage of the "list" object over the standard array may look tiny, it becomes essential in complex applications, where many functions are applied to objects returned from other functions. Thus, using suitable objects rather than standard arrays is most useful.

Consider, for example, the following problem: let $f \equiv f(x)$ be a function of the independent variable x. At a given point x, calculate the derivatives of f up to order n, denoted by

$$\begin{aligned} f^{(0)}(x) &= f(x), \\ f^{(1)}(x) &= f'(x), \\ f^{(2)}(x) &= f''(x), \\ f^{(k)}(x) &= f^{(k-1)'}(x), \end{aligned}$$

and so on. A suitable candidate for storing these derivatives for future use is the "list" object. For example, when

$$f(x) = \frac{1}{x},$$

we have

$$f^{(k)}(x) = -\frac{k}{x} f^{(k-1)}(x).$$

The code that calculates and stores these derivatives is as follows:

```
template<class T>
const list<T>
deriveRinverse(const T&r, int n){
  list<T> Rinverse(n+1,0);
  Rinverse(0) = 1/r;
  for(int i=0; i<n; i++)
    Rinverse(i+1) = -double(i+1)/r * Rinverse[i];
  return Rinverse;
}  //  derivatives of 1/r
```

This function returns the list of derivatives of $1/x$ up to and including order n. In the next section, we'll see how lists can also be used in the Taylor expansion of a function.

5.11 The Taylor Expansion

In this section, we discuss efficient ways to calculate the Taylor expansion of a function $f(x)$. Let x be fixed, and let h be a small parameter. Assume that f has sufficiently many derivatives in the closed interval $[x, x+h]$. The Taylor expansion of order n at $x+h$ gives the value of the function at $x+h$ in terms of the values of the function and its derivatives at x, plus an error term that involves the $(n+1)$th derivative at an intermediate point $x \le \xi \le x+h$:

$$f(x+h) = \sum_{i=0}^{n} \frac{f^{(i)}(x)h^i}{i!} + \frac{f^{(n+1)}(\xi)h^{n+1}}{(n+1)!}.$$

When f is sufficiently smooth, one can assume that the $(n+1)$th derivative of f is bounded in $[x, x+h]$, so for sufficiently large n the error term is negligible. In this case, a good approximation to $f(x+h)$ is given by

$$f(x+h) \doteq \sum_{i=0}^{n} \frac{f^{(i)}(x)h^i}{i!}.$$

The computational problem is to calculate this sum efficiently. This problem contains two parts: first, to find an efficient algorithm, and then to implement it efficiently and elegantly on a computer.

The elementary task in the implementation is to pass the required information about f and its derivatives at x to the computer. The function "Taylor()" in the code below must have as input the numbers $f(x)$, $f'(x)$, $f''(x)$, ..., $f^{(n)}(x)$ before it can start calculating the Taylor approximation to $f(x+h)$. As discussed above, these numbers are placed and passed to the function in a single "list" object. This way, the programmer who writes the function can disregard storage issues and concentrate on the mathematical algorithm.

Since the above Taylor approximation is actually a polynomial of degree n in h, one may use a version of the algorithm in Section 5.6. In this version, the terms in the polynomial are calculated recursively by

$$\frac{h^i}{i!} = \frac{h}{i} \cdot \frac{h^{i-1}}{(i-1)!}.$$

These terms are then multiplied by $f^{(i)}(x)$ (available in the input "list" object) and added on to the sum. We refer to this version as the standard algorithm; it is implemented as follows:

```
template<class T>
const T
Taylor(const list<T>&f, const T&h){
  T powerOfHoverIfactorial = 1;
  T sum=0;
  for(int i=0; i<f.size()-1; i++){
    sum += f[i] * powerOfHoverIfactorial;
    powerOfHoverIfactorial *= h/(i+1);
  }
  return sum;
}  //  Taylor approximation to f(x+h)
```

Note that the last item in the input list, which contains the $(n+1)$th derivative, is not actually used here; it is reserved for the purpose of estimating the error (see Chapter 6, Section 10).

The above standard algorithm requires a total of $3n$ multiplications and n additions. A more efficient algorithm is a version of the Horner algorithm in Section 5.8. This version is based on the observation that the Taylor approximation can be written in the form

$$\left(\cdots\left(\left(f^{(n)}(x)\frac{h}{n} + f^{(n-1)}(x)\right)\frac{h}{n-1} + f^{(n-2)}(x)\right)\frac{h}{n-2}\cdots\right)\frac{h}{1} + f^{(0)}(x).$$

The following code implements this formula:

```
template<class T>
const T
HornerTaylor(const list<T>&f, const T&h){
  T result = f[f.size()-2];
  for(int i=f.size()-2; i>0; i--){
    result *= h/i;
    result += f[i-1];
  }
  return result;
}  //  Horner algorithm for Taylor approximation
```

This code requires only $2n$ multiplications and n additions. Furthermore, its implementation is slightly shorter and more elegant.

The question is, though, is it worth it? The standard algorithm has one major advantage: it adds the terms in the natural order, from 0 to n. This is not only more in the spirit of the original formula but also more economic in terms of storage in some cases.

Consider, for example, a "short-memory" process, in which $f^{(i)}(x)$ depends only on the previous number $f^{(i-1)}(x)$, but not on the yet previous numbers $f(x)$, $f'(x)$, $f''(x)$, ..., $f^{(i-2)}(x)$ (Figure 5.2). In this case, it makes more sense to calculate $f^{(i)}(x)$ inside the loop and drop it once it has contributed to the sum and has been used to calculate the next number $f^{(i+1)}(x)$. Actually, the new number $f^{(i+1)}(x)$ can be stored in the same variable used previously to store the old number $f^{(i)}(x)$. This strategy, however, is possible only in the standard algorithm, where terms are added in the natural order, but not in Horner's algorithm, where they are added in the reverse order (Figure 5.3). Thus, the standard algorithm becomes in this case much more attractive, because it can avoid storing and passing input to the "Taylor()" function.

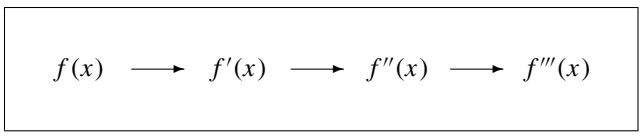

Figure 5.2. *Short-memory process: for fixed x, each derivative can be calculated from data about the previous one only.*

In our applications, however, we are mainly interested in "long-memory" processes, in which $f^{(i)}(x)$ depends not only on the previous number $f^{(i-1)}(x)$ but also on all the yet previous numbers $f(x)$, $f'(x)$, $f''(x)$, ..., $f^{(i-2)}(x)$ (Figure 5.4). In this case, the list of derivatives must be stored in its entirety as in the above codes, so the Horner algorithm is preferable thanks to its lower cost in terms of operation count.

We'll see examples of the short-memory process in Chapter 6, Section 9, and the long-memory process in Chapter 6, Section 14. In the next section, we illustrate how useful it is to pass entire lists to a function, particularly when all the items in them are combined to produce the required result.

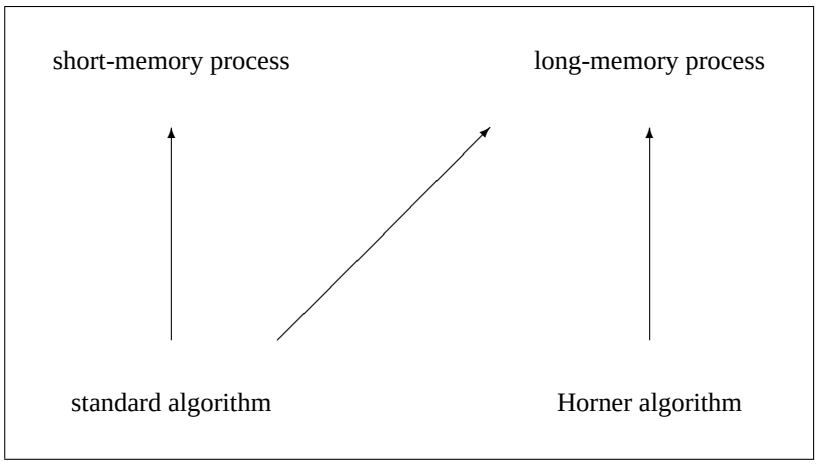

Figure 5.3. *Horner's algorithm is preferable in a long-memory process, where the derivatives must be calculated and stored in advance anyway, but not in a short-memory process, where they are better used and dropped, as can be done only in the standard algorithm.*

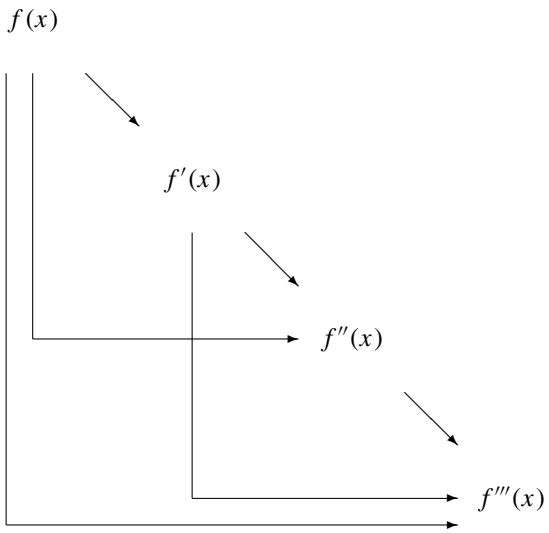

Figure 5.4. *Long-memory process: for fixed x, the calculation of each derivative requires data about all the previous ones.*

5.12 Derivatives of a Product

In this section, we show why it is particularly important to have the opportunity to store the list of derivatives as an object. For this purpose, we present a function that takes list

arguments and combines all the items in them in the calculation. This function will be particularly useful in the long-memory process in Chapter 6, Section 14.

Assume that the derivatives of the functions $f(x)$ and $g(x)$ at a given point x up to and including order n are available. Compute the nth derivative of the product fg at x, denoted by

$$(fg)^{(n)}(x).$$

Note that this quantity has the same algebraic structure as Newton's binomial:

$$(f+g)^n = \sum_{i=0}^{n} \binom{n}{i} f^i g^{n-i}.$$

The only difference is that here the sum is replaced by a product, and the power is replaced by a derivative. Therefore, we have the formula

$$(fg)^{(n)} = \sum_{i=0}^{n} \binom{n}{i} f^{(i)} g^{(n-i)}$$

at the given point x.

This formula is implemented in the code below. The function "deriveProduct()" takes as input two "list" objects that contain the derivatives of f and g up to and including order n at the fixed point x and returns the nth derivative of fg at x. It is assumed that a global array of integers that contains Pascal's triangle is available. This array, named "triangle", is formed in advance, as in Chapter 1, Section 19, and placed in a global domain accessible to the "deriveProduct()" function. This seems to be a better strategy than recalculating the required binomial coefficients each time the function is called:

```
template<class T>
const T
deriveProduct(const list<T>&f,const list<T>g,int n){
  T sum = 0;
  for(int i=0; i<=n; i++)
    sum += triangle[n-i][i] * f[i] * g[n-i];
  return sum;
}  //  nth derivative of a product
```

5.13 Polynomial of Two Variables

The polynomial $\sum a_i x^i$ in Section 5.4 is implemented as the list of coefficients a_0, a_1, $a_2, \ldots, a_n$. This implementation is appropriate because the polynomial is defined uniquely by its coefficients.

Still, one may ask, why use a list rather than a vector? After all, the coefficients are all of the same type, so they can be safely stored in a vector of dimension $n+1$. Furthermore, a vector is more efficient than a list thanks to direct indexing, that is, using an array to store the coefficients themselves rather than their addresses. This saves not only storage but also valuable time, by using efficient loops over the coefficients stored continuously in the computer memory. Why then use a list?

The answer is that, in some cases, the coefficients in the polynomial occupy different amounts of memory, and hence cannot be stored in an array. Consider, for example, the polynomial of two independent variables x and y:

$$p(x, y) \equiv \sum_{i+j \le n} a_{i,j} x^i y^j,$$

where i and j are nonnegative indices, $a_{i,j}$ are the given coefficients, and n is the degree of the polynomial. This polynomial may also be written in the form

$$\begin{aligned} p(x, y) &= \sum_{k=0}^{n} \sum_{i+j=k} a_{i,j} x^i y^j \\ &= \sum_{k=0}^{n} \left(\sum_{j=0}^{k} a_{k-j,j} \left(\frac{y}{x}\right)^j \right) x^k. \end{aligned}$$

In this form, $p(x, y)$ can be viewed as a polynomial of degree n in x, with coefficients that are no longer scalars but rather polynomials in y/x. In fact, the kth coefficient is by itself the polynomial of degree k in y/x, given by

$$a_k(y/x) \equiv \sum_{j=0}^{k} a_{k-j,j} \left(\frac{y}{x}\right)^j.$$

Thus, the original polynomial $p(x, y)$ can be implemented as the list $a_0(y/x)$, $a_1(y/x)$, $a_2(y/x), \ldots, a_n(y/x)$. Each item in this list is a polynomial in its own right, thus also implemented as a "polynomial" object. More specifically, the original polynomial $p(x, y)$ is implemented as a polynomial of polynomials, or a "polynomial<polynomial<T> >" object, whose kth item is the polynomial $a_k(y/x)$. The polynomial $a_k(y/x)$ is defined and stored in terms of its own coefficients $a_{k,0}, a_{k-1,1}, a_{k-2,2}, \ldots, a_{0,k}$.

Clearly, this implementation is possible thanks to the fact that the "polynomial" class is derived from the "list" class, which may contain items of different sizes (such as polynomials of different degrees). With this implementation, it is particularly easy to multiply two polynomials of two variables. (This operation is particularly useful in high-order finite elements in Chapter 15.) Here is how this is done.

Let $q(x, y)$ be another polynomial of two variables:

$$q(x, y) \equiv \sum_{k=0}^{m} b_k(y/x) x^k,$$

where $b_k(y/x)$ is by itself a polynomial of degree k in y/x. Then, the product of p and q is

$$\begin{aligned} (pq)(x, y) &\equiv p(x, y) q(x, y) \\ &= \sum_{k=0}^{m+n} \left(\sum_{j=\max(0,k-n)}^{\min(k,m)} a_{k-j}(y/x) b_j(y/x) \right) x^k. \end{aligned}$$

Note that the product of polynomials $a_{k-j}(y/x)b_j(y/x)$ in the above parentheses is by itself a polynomial of degree k in y/x, so the sum in these parentheses is also a polynomial of degree k in y/x. This sum of products can be calculated using arithmetic operations between polynomials of a single variable (Sections 5.4 and 5.5). Thus, the required product of p and q can be carried out in the same way as in Section 5.5; the only difference is that the coefficients a_{k-j} and b_j are polynomials rather than scalars, and hence the arithmetic operations between them are interpreted as arithmetic operations between polynomials rather than scalars. This interpretation is used automatically once the compiler encounters polynomials rather than scalars.

The multiplication operator in Section 5.5 can thus be used for polynomials of two variables as well. Indeed, it works just as before, with the template 'T' in it (denoting the type of coefficient) specified to be "polynomial". For example, the following code computes and prints to the screen the coefficients of the polynomial $(1+x+y)^2$:

```
int main(){
  polynomial<polynomial<double> >
      p2(2,polynomial<double>(1,1));
  p2(1) = polynomial<double>(2,1);
  print(p2 * p2);
  return 0;
}
```

5.14 Integration of a Polynomial

Here we show how convenient it is to use the "polynomial" object to calculate integrals in certain domains. (This task is required, e.g., in high-order finite elements in Chapter 15.)

We start with a polynomial of one independent variable:

$$p(x) = \sum_{i=0}^{n} a_i x^i.$$

The integral of this polynomial in the unit interval [0, 1] is

$$\int_0^1 p(x)dx = \sum_{i=0}^{n} a_i \int_0^1 x^i dx = \sum_{i=0}^{n} \frac{a_i}{i+1}.$$

This formula is implemented in the following code:

```
template<class T>
const T
integral(const polynomial<T>&p){
  T sum = 0;
  for(int i=0; i<p.size(); i++)
    sum += (1./(i+1)) * p[i];
  return sum;
}  //  integral in the unit interval
```

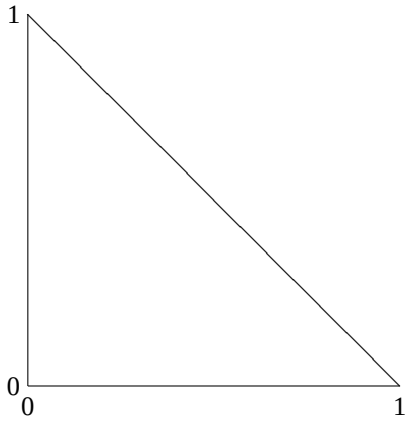

Figure 5.5. *The triangle in which the polynomial $p(x, y)$ is integrated.*

Next, we consider the problem of integrating a polynomial of two independent variables x and y in the right-angle triangle in Figure 5.5. Consider the polynomial $p(x, y)$ given by

$$\begin{aligned} p(x, y) &= \sum_{k=0}^{n} \sum_{i+j=k} a_{i,j} x^i y^j \\ &= \sum_{k=0}^{n} \sum_{j=0}^{k} a_{k-j,j} y^j x^{k-j}. \end{aligned}$$

The integral of this polynomial in the triangle in Figure 5.5 takes the form

$$\begin{aligned} \int p(x, y) dx dy &= \sum_{k=0}^{n} \sum_{j=0}^{k} a_{k-j,j} \int_0^1 \left(\int_0^{1-x} y^j dy \right) x^{k-j} dx \\ &= \sum_{k=0}^{n} \sum_{j=0}^{k} a_{k-j,j} \int_0^1 \frac{(1-x)^{j+1}}{j+1} x^{k-j} dx. \end{aligned}$$

Thus, we have every tool required for calculating this integral. Indeed, since $1 - x$ is a polynomial in x, and we already know how to multiply polynomials, we can compute the polynomials

$$(1-x)^{j+1} = (1-x)^j (1-x) \quad (j = 2, 3, 4, \ldots, n).$$

Furthermore, we know how to multiply these polynomials by the polynomials x^{k-j} and scalars $a_{k-j,j}/(j+1)$. Finally, we also know how to sum these polynomials and integrate in the unit interval. This completes the algorithm for integrating $p(x, y)$ in the triangle in Figure 5.5.

Here is the code that implements this algorithm:

```
template<class T>
const T
integral(const polynomial<polynomial<T> >&p){
  polynomial<T> sum(p.size()+1,0);
  polynomial<T> one(1,1);
  polynomial<T> x(2,0);
  x(1) = 1;
  polynomial<T> oneMinusX(2,1);
  oneMinusX(1) = -1;
  list<polynomial<T> > xPowers(p.size(),one);
  list<polynomial<T> > oneMinusXpowers(p.size()+1,one);
  for(int i=1; i<p.size(); i++)
    xPowers(i) = x * xPowers[i-1];
  for(int i=1; i<=p.size(); i++)
    oneMinusXpowers(i) = oneMinusX * oneMinusXpowers[i-1];
  for(int k=p.degree(); k>=0; k--)
    for(int j=0; j<=k; j++)
      sum += (p[k][j]/(j+1))
          * oneMinusXpowers[j+1] * xPowers[k-j];
  return integral(sum);
}  //  integral in the triangle
```

Although this function bears the same name as the previous function that integrates in the unit interval, no ambiguity occurs. Indeed, when the function is actually called, the compiler looks at its concrete argument. If it is a polynomial of one variable, then the previous version is invoked. If, on the other hand, it is a polynomial of two variables, then this version is invoked.

5.15 Exercises

1. Use the code in Section 5.5 to compute the coefficients of the polynomial
$$(1 + 2x + 2x^2 + x^3)^2.$$
2. Use the above code in a loop to compute the coefficients of the polynomial
$$(a + bx)^n,$$
where a and b are some real numbers and n is a large integer number. Verify that the result agrees with Newton's binomial:
$$(a + bx)^n = \sum_{i=0}^{n} \binom{n}{i} a^{n-i} b^i x^i.$$
3. Compare the efficiency and accuracy of the functions in Sections 5.6 and 5.8 in calculating the value of the polynomial
$$\sum_{n=0}^{N} x^n = \frac{x^{N+1} - 1}{x - 1}$$
at the points $x = 2$, $x = 3$, and $x = 4$.

4. It would be more in the spirit of object-oriented programming to rename the functions in Sections 5.6 and 5.8 as "operator()". This function should then be a member of the "polynomial" class and take only one argument 'x' to return the value of the current "polynomial" object at 'x'. This way, the function can be simply called as "p(x)" to return the value of the polynomial 'p' at 'x'. This way, the function doesn't act on 'p' but rather reflects its property to return different values for different values of 'x'. Why is this impossible here? Is it because the base "list" class already uses an "operator()" that shouldn't be overridden? Can you get around this problem?

5. Use the code segments in Section 5.9 to calculate

$$2^{10}, 2^{20}, 3^{17}, 7^{16}, \ldots.$$

Compare the results with those of the recursive "power()" function in Chapter 1, Section 17, in terms of efficiency and correctness.

6. Which code segment in Section 5.9 do you find particularly easy to read, understand, and modify? What writing style and programming strategy is most suitable for you?

7. Use the code in Section 5.10 to compute the lists of derivatives of the function $f(x) = 1/x$ at the points $x = 2$, $x = 3$, and $x = 4$.

8. Use the above lists and the code in Section 5.11 to approximate $1/(2.1)$, $1/(2.9)$, and $1/(3.9)$, using the Taylor expansion around $x = 2$, $x = 3$, and $x = 4$ (respectively). Verify that the error is indeed within the expected error estimate.

9. Which algorithm in Section 5.11 is more efficient in the calculation in the previous exercise?

10. Apply the code in Section 5.12 to construct the list of derivatives of the function $1/x^2 = (1/x)(1/x)$ at $x = 1$, $x = -1$, and $x = 0.5$.

11. Use the lists from the previous exercise in the code in Section 5.11 to obtain the Taylor approximation to $1/(0.9)^2$, $1/(-1.1)^2$, and $1/(0.45)^2$. Are the errors within the expected error estimates?

12. Apply the code in Section 5.12 with the above lists to construct the list of derivatives of the function $1/x^4 = (1/x^2)(1/x^2)$ at $x = 1$, $x = -1$, and $x = 0.5$.

13. Use the lists from the previous exercise in the code in Section 5.11 to obtain the Taylor approximation to $1/(0.9)^4$, $1/(-1.1)^4$, and $1/(0.45)^4$. Are the errors within the expected error estimates?

14. Construct the lists of derivatives of the functions $\sin(x)$, $\cos(x)$, and $\sin(2x)$ at the points $x = 0$, $x = \pi/4$, and $x = \pi/3$.

15. Use the lists of derivatives of $\sin(x)$ and $\cos(x)$ calculated above in the code in Section 5.12 to construct the list of derivatives of the function

$$\sin(2x) = 2\sin(x)\cos(x).$$

Compare the results with those from the previous exercise.

16. Use the lists from the previous exercise in the Taylor expansion around the above points, and verify that the errors are indeed within the expected error estimates.

17. Which code in Section 5.11 do you find particularly easy to read, understand, and modify? What writing style and programming strategy is most suitable for you?

18. Use the code in Section 5.13 to compute the coefficients of the polynomial of two variables
$$(1 + 2x + 2y + x^2 + xy + y^2)^2.$$

19. Use the above code in a loop to compute the coefficients of the polynomial of two variables
$$(ax + by)^n,$$
where a and b are some real numbers and n is a large integer number. Verify that the result agrees with Newton's binomial:
$$(ax + by)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i} x^i y^{n-i}.$$

20. Use the code in Section 5.14 to calculate the integral of the above polynomials in the triangle in Figure 5.5. Verify that the result is indeed correct by calculating it manually.

21. Write a function that calculates the value of a polynomial of two variables $p(x, y)$ as follows. First, the polynomial is written in the form
$$p(x, y) = \sum_{k=0}^{n} a_k(y/x)x^k,$$
as in Section 5.13. In this form, $p(x, y)$ can be passed to the function as a "polynomial<polynomial<T> >" object, in which the coefficients are themselves polynomials. In fact, the kth coefficient in $p(x, y)$ is $a_k(y/x)$, which contains the coefficients $a_{k,0}$, $a_{k-1,1}$, $a_{k-2,2}$, ..., $a_{0,k}$. Now, the "HornerPolynomial" function in Section 5.8 is called to compute the individual $a_k(y/x)$. These values are stored in a local "polynomial<T>" object, to which the "HornerPolynomial()" function is applied to produce the required value $p(x, y)$.

22. Rewrite your code from the previous exercise in a short-memory approach, in which the $a_k(y/x)$'s are calculated one by one and deleted right after the term $a_k(y/x)x^k$ is calculated and added to the current sum that will eventually produce $p(x, y)$. In this approach, the $a_k(y/x)$'s don't have to be stored in a local "polynomial<T>" object. However, the final call to the Horner algorithm must be replaced by the standard algorithm in Section 5.6. How does your present code compete with the code in the previous exercise?

23. Let $f(x, y)$ be a given function of two variables, and let x and y be fixed. Consider the polynomial $p(h_x, h_y)$ of the two variables h_x and h_y, whose coefficients are given by

$$a_{i,j} \frac{1}{i! \cdot j!} \cdot \frac{\partial^{i+j} f}{\partial^i x \partial^j y}(x, y).$$

Actually, $p(h_x, h_y)$ is the Taylor approximation of $f(x + h_x, y + h_y)$:

$$f(x + h_x, y + h_y) \doteq p(h_x, h_y).$$

Use the algorithms in the two previous exercises to calculate the Taylor approximation of the function $f(x, y) = \sin(x)\cos(y)$ at $x = \pm\pi/4$, $y = \pm\pi/3$, $h_x = \pm 0.1$, and $h_y = \pm 0.1$. Verify that the approximation indeed improves as n (the degree of p) increases.

24. Compare your code from the previous exercise to your answer to a similar exercise at the end of Chapter 3. Do you obtain the same numerical results? Which code is more efficient? Which code is more transparent?

25. Use the guidelines in Section 5.13 to implement polynomials of three variables x, y, and z.