

# Real Polynomials and Complex Polynomials

## Introduction

The focus of this exercise is the study of polynomials with real or complex coefficients. The direct goal is to develop a program that will plot polynomials (real or complex) in a flexible manner that will permit the easy exploration of geometric issues. The pedagogical goal is to utilize several object-oriented concepts to facilitate the program development:

- The `Complex` number class will be constructed by *inheritance* from the `Point2D` class so that many of the complex arithmetic operations will already be defined and so that integration into the graphics toolkits will be automatic.
- The `Polynomial` class will be defined as *a template class that is inherited* from the `Array` class for the underlying numeric type. The polynomial coefficients will be the elements in the array. The polynomial degree will be an additional data member.
- The tools for defining *scaling transformations, creating graphs, and plotting tables* that are defined in `Graphics.h` will be used whenever possible. In one case, you will adapt a tool to work with complex polynomials.

It is hoped that this assignment will encourage a style of programming in which the programming task is thoughtfully subdivided into a number of issues that can be treated simply and elegantly. You will complete the project by designing and coding certain functions in the supporting toolkits and in the main program. A large part of the user interface is already done.

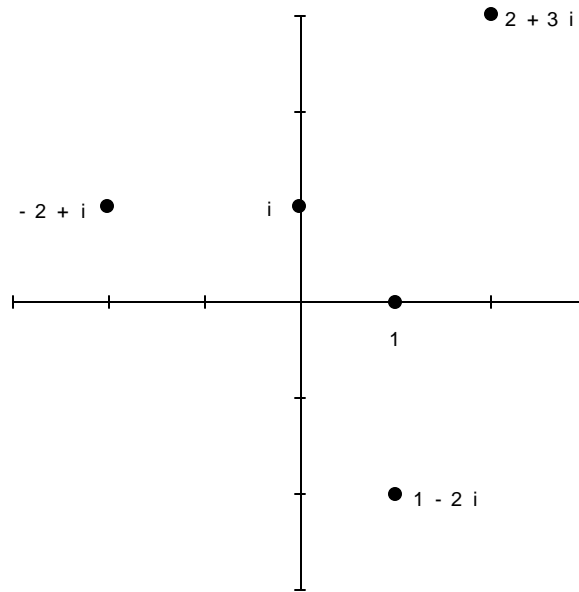
To make sure that everyone knows the relevant mathematical facts about complex numbers and complex polynomials, we will first review this information.

## Mathematical Aspects of Complex Numbers

Complex numbers were discovered about 500 years ago when mathematicians attempted to solve quadratic, cubic, and 4th degree polynomial equations by formulas. The well known “quadratic formula” is the simplest example of this process. It was soon discovered that a mysterious quantity, the square root of negative 1, seemed essential in the formulas. This was a puzzle since negative numbers do not have square roots in the ordinary real numbers. A quantity was therefore “invented” to name this imaginary square root of  $-1$ :

$$i = \sqrt{-1}$$

Since this quantity  $i$  did not exist among real numbers, it was called “imaginary” and any other numbers obtained by combinations of real numbers with  $i$  were also called “imaginary”. Three hundred years later, the mathematician Gauss provided a geometric interpretation of “imaginary” numbers which gave them a concrete meaning.



The idea is to use two perpendicular axes in the plane and to place the real numbers on the horizontal axis and to place  $i$  on the vertical axis. The additive combinations of real numbers and multiples of  $i$  would then fill out the plane determined by the two axes. The diagram above shows several examples of this interpretation of “imaginary” numbers. Gauss’s discovery took the mystery out of “imaginary” numbers. They are simply numbers that inhabit the plane instead of being restricted to a 1-dimensional line. Once the “imaginary” numbers were geometrically explained, the name “imaginary” became less popular and was replaced by “complex” to indicate simply that such numbers are compound numbers with two coordinate components, one along the horizontal or “real” axis and one along the vertical or “imaginary” axis. If  $z = x + y i$  is a complex number, it is still traditional to refer to  $x$  as the “real part of  $z$ ” and to  $y$  as the “imaginary part of  $z$ ”.

Of course, in C++, we represent only the *components* of a complex number. We do not represent  $i = \sqrt{-1}$  explicitly. The magic number  $i$  is revealed *only by the arithmetic operations we define and implement*. These operations are based on the fundamental equation:

$$i^2 = -1$$

The first part of your task will be to implement the basic arithmetic operations for complex numbers. We will simply illustrate these rules by example here using two numbers  $z = 3 + 4i$  and  $w = 2 - i$ .

Addition:  $(3 + 4i) + (2 - i) = 5 + 3i$ .

Subtraction:  $(3 + 4i) - (2 - i) = 1 + 5i$ .

Multiplication:  $(3 + 4i) * (2 - i) = 6 - 3i + 8i - 4i^2 = (6 + 4) + (-3 + 8)i = 10 + 5i$ .

Conjugate: For  $z = 3 + 4i$ , the Conjugate of  $z$  is:  $\bar{z} = 3 - 4i$ .

RadiusSquare: The product of a complex number  $z$  and its conjugate  $\bar{z}$  is always a positive real number (or zero if  $z$  is precisely zero):

$$z * \bar{z} = x^2 + y^2 = 3^2 + 4^2 = 25$$

By Pythagoras's Theorem, this number is the square of the distance from  $z$  to the origin. We call this quantity  $z * \bar{z}$  the RadiusSquare.

Radius: The "square root of the RadiusSquare" is the distance from  $z$  to the origin. We call this quantity the Radius or Length:

$$\text{Radius or Length} = \text{sqrt}(z * \bar{z}) = \sqrt{25} = 5$$

Inverse: Simple algebra shows the following identity:

$$1 = z * (\bar{z} / (z * \bar{z})) = z * (\bar{z} / \text{RadiusSquare})$$

Hence the reciprocal or inverse of  $z$  may be computed by:

$$1/z = \bar{z} / \text{RadiusSquare} = (3/25) - (4/25) i$$

Notice that the left hand side,  $1/z$ , represents the division of 1 by the complex number  $z$  in the denominator. The expression on the right,

$$\text{Inverse}(z) = \bar{z} / \text{RadiusSquare} = \text{Conjugate}(z) / \text{RadiusSquare}$$

computes its value using division by the real number, RadiusSquare, which can be done coordinate-wise. This conversion is critical because it computes  $1/z$  in which the denominator is complex as  $\bar{z} / \text{RadiusSquare}$  in which the denominator is real.

Division: Once we have the Inverse, we can do any division by using multiplication together with the Inverse function:

$$w / z = w * (1 / z) = w * \text{Inverse}(z)$$

$$= (2 - i) * [(3/25) - (4/25)i] = (2/25) - (11/25)i$$

## Implementation of a Complex Number Class in C++

In the Core Tools, we have a class `Point2D` in `Basic2D.h` and `Basic2D.cpp` that represents a point in the 2-dimensional plane with two `double` coordinates `x` and `y`. The `Point2D` class has several features that we wish for a `Complex` class:

- Many complex-like operators including addition, subtraction, multiplication-by-scalars, and division-by-scalars are already defined for `Point2D` objects
- `Point2D` objects have IO Tools operations defined
- `Point2D` objects integrate into the graphics system in Core Tools

It is therefore convenient to construct the `Complex` class by *inheritance* from the `Point2D` class and to simply add the additional operations needed to complete the definition of the complex arithmetic. Therefore, the framework for the definition of the `Complex` class in `Complex.h` is:

```
class Complex : public Point2D {
public:
    // additional public functions and operators ...
};
```

Notice that there are *no additional data members* in the class `Complex` only additional functions. The `x` and `y` coordinate data *inherit* from the `Point2D` class. Inheritance does not mean that the private data `x` and `y` in the `Point2D` class can be accessed directly by functions in the `Complex` class but only that the public functions of the `Point2D` class can be called as public functions in the `Complex` class. Since the `Point2D` class has `set` and `get` functions, the `Complex` class can deal with the `x` and `y` data through these functions (as well as by calling the other public member and friend functions defined in `Point2D`).

Since the `Complex` class is based on the `Point2D` class, it is absolutely essential that you read and understand the `Point2D` class definition in `Basic2D.h` and `Basic2D.cpp`. *Do this before reading further in this document.*

Since the `Point2D` class already defines many functions and operators, our task in defining the `Complex` class is to add the additional functionality of complex numbers. These requirements come in three categories:

### (1) Constructors

The necessary constructors should enable building a `complex` object either from `x, y` data or from a `point2D` object. These definitions are in fact given in `complex.h` as follows:

```

Complex(double X = 0, double Y = 0) : Point2D(X, Y) { };

Complex(const Point2D& P) : Point2D(P) { };

```

The first definition is subtle since it permits a `Complex` object to be created with one double `x` and with the `y` value defaulted to `0`. In accordance with the conversion rules of C++, this permits the automatic conversion of a double `x` to a `Complex` object representing  $x + 0i$ . This auto-conversion is consistent with standard mathematical usage.

The second definition is a “copy constructor” which converts a `Point2D` object into the `Complex` object with the same data. Since `Complex` inherits from `Point2D` and has no additional member data variables, this also serves to define the “copy constructor” for `Complex` to `Complex`.

## (2) Arithmetic Functions and Operators

The functions in this category implement the multiplication and division operators for `Complex` objects and the related `Conjugate` and `Inverse` functions. The headers with comments are:

```

// computes A * B
// returns the object constructed by value

friend Complex operator* (const Complex& A, const Complex& B);

// computes (the object) * A
// returns the object by reference

Complex& operator*= (const Complex& A);

// computes Complex conjugate of A
// returns the object constructed by value

friend Complex Conjugate(const Complex& A);

// computes Complex inverse of A, that is, 1 / A
// returns the object constructed by value

friend Complex Inverse(const Complex& A);

// computes A / B
// returns the object constructed by value

friend Complex operator/ (const Complex& A, const Complex& B);

// computes (the object) / A
// returns the object by reference

Complex& operator/= (const Complex& A);

```

These 6 functions (4 friend functions and 2 member functions) are the ones you need to define in `Complex.h`. Let us give some brief hints about how to do this. For further hints, you should examine the definitions in the `Point2D` class.

For the friend multiplication function, `operator*`, you must use `Get` to extract the coordinates of the parameters A and B. Let's call these coordinates X1, Y1 and X2, Y2 respectively. Then imagine the product:

$$\begin{aligned} (X1 + Y1 i) * (X2 + Y2 i) &= (X1 * X2) + (X1 * Y2) i + (Y1 * X2) i + (Y1 * Y2) * (i * i) \\ &= (X1 * X2 - Y1 * Y2) + (X1 * Y2 + Y1 * X2) i \end{aligned}$$

Here we have used the fact that  $i * i = -1$ . You see that this formula gives you the coordinates X and Y of the product:

$$X = X1 * X2 - Y1 * Y2$$

$$Y = X1 * Y2 + Y1 * X2$$

After this calculation is done, you can complete `operator*` with the line:

```
return Complex(X, Y);
```

This line constructs a new `Complex` object with coordinates X, Y and returns this object by value. Usually, the compiler completes these actions in one step using an optimization called the “return value optimization”.

The member multiplication function, `operator*=`, can be defined using `operator*` in the same manner as in similar functions in the `Point2D` class.

The friend function `conjugate` returns the `Complex` object constructed with coordinates X, -Y if the parameter object has coordinates X, Y.

The friend function `inverse` first computes the double `radiusSquare` in a local variable. This is done using the `DotProduct` function in the `Point2D` class. Then the `conjugate` is computed and stored in a second local variable `c`. Finally, the function returns `c / radiusSquare`. This division may use the coordinate-wise division operation defined in the `Point2D` class or may be done by direct calculation within the `inverse` function.

The friend division function, `operator/`, is defined using the identity:

$$A / B = A * (1 / B) = A * Inverse(B)$$

Hence, division should be calculated using the multiplication operator, `operator*`, together with the `inverse` function.

The member division function, `operator/=`, can be defined using `operator/` in the same manner as in similar functions in the `Point2D` class.

### (3) Redefinitions of Certain `Point2D` Arithmetic Functions and Operators

Unfortunately, when we define the four operators `operator*`, `operator*=`, `operator/`, and `operator/=` in the `Complex` class, the corresponding definitions with the same names in the `Point2D` class are hidden. This should not really happen but is a bug in our compiler. The final functions we must redefine are therefore these hidden `Point2D` functions. Since we felt it is a compiler bug that necessitates these definitions, we have put them into the `Complex` class definition for you. You need to do no work here.

## Testing the Complex Class

The file `complex.h` should be edited and tested in the `Complex` subfolder in the `PolyLab` folder. You should not change the test program `shell.cpp` in any way. The test program repeatedly asks for a pair of complex numbers `A` and `B` and then computes and prints 8 quantities:

$$A + B, A - B, A * B, A / B, \text{Conjugate}(A), \text{Inverse}(A), \text{Conjugate}(B), \text{Inverse}(B)$$

To help you check your programming, we will give you two data sets with answers that should match your calculations exactly:

	$A = 1 + 2i$ and $B = -1 + 2i$	$A = 5 + 12i$ and $B = 3 + 4i$
$A + B$	$0.000000 + 4.000000i$	$8.000000 + 16.000000i$
$A - B$	$2.000000 + 0.000000i$	$2.000000 + 8.000000i$
$A * B$	$-5.000000 + 0.000000i$	$-33.000000 + 56.000000i$
$A / B$	$0.600000 + 0.800000i$	$2.520000 + 0.640000i$
$\text{Conjugate}(A)$	$1.000000 - 2.000000i$	$5.000000 - 12.000000i$
$\text{Inverse}(A)$	$0.200000 - 0.400000i$	$0.029586 - 0.071006i$
$\text{Conjugate}(B)$	$-1.000000 - 2.000000i$	$3.000000 - 4.000000i$
$\text{Inverse}(B)$	$-0.200000 - 0.400000i$	$0.120000 - 0.160000i$

Once you are convinced that your file `complex.h` is correct, then copy and paste the file into the `Polynomial` subfolder in the `PolyLab` folder. From this point onward, you will work with the `Polynomial` subfolder.

## Polynomials

A polynomial is a mathematical expression of the form

$$P(X) = p_0 + p_1 X + p_2 X^2 + \dots + p_D X^D$$

The index  $D$  of the highest degree term is called the *degree* of the polynomial and the numbers  $p_0, p_1, p_2, \dots, p_D$  are called the *coefficients*. In mathematics, a polynomial automatically defines a function: replace the formal symbol  $X$  by a number and calculate the expression to get the function value.

In the 19<sup>th</sup> century, a mathematician by the name of Horner discovered a faster technique for the evaluation of polynomials than the “obvious method” of computing the powers of  $X$  and then computing the sum of products. We will illustrate this method with an example and then describe it in general.

Consider the polynomial:

$$P(X) = 1 - 3 X + 5 X^2 - 7 X^3 + 9 X^4$$

Now consider the following steps:

Initialize:     $\text{answer} = 9$

Compute:     $\text{answer} = \text{answer} * X - 7$   
                   so  $\text{answer} = 9 * X - 7$

Compute:     $\text{answer} = \text{answer} * X + 5$   
                   so  $\text{answer} = (9 * X - 7) * X + 5$

Compute:     $\text{answer} = \text{answer} * X - 3$   
                   so  $\text{answer} = ((9 * X - 7) * X + 5) * X - 3$

Compute:     $\text{answer} = \text{answer} * X + 1$   
                   so  $\text{answer} = (((9 * X - 7) * X + 5) * X - 3) * X + 1$   
                            $= 9 X^4 - 7 X^3 + 5 X^2 - 3 X + 1$   
                            $= P(X)$

Horner’s method requires 4 multiplications and 4 additions to calculate a polynomial of degree 4 whereas the “obvious method” requires 7 multiplications and 4 additions. Hence Horner’s method is faster than the “obvious method”. In addition, numerical analysts have shown that Horner’s method is numerically more *stable*, that is, it is less likely to introduce errors due to accumulated round-off.

Horner’s method can be summarized as an algorithm:

Initialize:     $\text{answer} = p_D$

Loop down for index  $k$  from  $(D-1)$  to  $0$ :

$\text{answer} = \text{answer} * X + p_k$

Our next goal is to introduce a polynomial template class that uses Horner's method to perform polynomial evaluation.

## The Polynomial Template Class

A polynomial needs a degree and an array of numerical coefficients. We can use the `Array` class to store the coefficients so the natural design is to make the `Polynomial` class be a derived class from the `Array` class. Since there are several numerical types (`int`, `long`, `short`, `float`, `double`, and `Complex`), the `Polynomial` class must be a template class. The framework definition in the file `PolynomialT.h` is:

```
template <class Number>
class Polynomial : public Array<Number> {

    // array holds coefficients for 0 <= i <= degree
    // totalsize == validsize == degree + 1

    int degree;

public:

    // additional public functions and operators ...

};
```

Notice that `Polynomial<Number>` has one data member `degree` in addition to the array of coefficients it acquires by inheritance from `Array<Number>`. The comment indicates that we plan to maintain the condition that the array members `totalsize` and `validsize` will both equal `degree + 1`.

The `Polynomial` class has 4 functions that you will have to enter:

```
// constructor
Polynomial(int D = 0) : degree(-1) { SetDegree(D); };

// function to set degree
// with error check and initialization of new coefficients to zero
Polynomial& SetDegree(int D);

// function to get degree
int Degree() const { return degree; };
```

```
// function call operator to implement polynomial evaluation
Number operator() (Number x) const;
```

The `Polynomial` class also has 3 functions for doing input and these have been done for you. As is common in many classes, the *constructor* function passes its main work to the set function called `setDegree`. Notice the trick in the constructor of setting `degree` to `-1` just before invoking `setDegree`. Since `setDegree` initializes new coefficients to 0, this will cause all of the coefficients in the brand new polynomial to be set to 0.

The `setDegree` function should do a number of tasks:

- Force its parameter `D` to be valid, that is, if `D` is less than 0 then make it 0
- Use `changeMemory` to allocate `D+1` cells in the underlying array object
- Error check array allocation by reading back `D` via `D = TotalSize() - 1`
- Validate all array cells
- Fill the new coefficients (from the old `degree+1` up to `D`) with 0
- Copy `D` into `degree`
- Return a reference to the polynomial object

The `GetDegree` function is obvious.

The `operator()` definition is how to make a polynomial object acquire the behavior of a function. If `P` is any object then C++ interprets the function expression `P(x)` as the call `P.operator()(x)` assuming an appropriate `operator()` is defined. By defining `operator()` for the `Polynomial` class, we can make a `Polynomial` object appear to be a function.

The `operator()` definition for the `Polynomial` class should implement Horner's algorithm. To access the `i`-th polynomial coefficient, use `Access(i)` which invokes an inherited function from the `Array` class. You will need to save the intermediate calculations in a local variable of type `Number` (the template parameter type). When done, return this result (not the object!).

As mentioned, the user input operations have already been programmed. All you need to do is call `P.Request()`.

Once the `Polynomial` template is defined, the header file introduces `typedef`'s for the six kinds of polynomial: `IntPoly`, `LongPoly`, `ShortPoly`, `FloatPoly`, `DoublePoly`, and `ComplexPoly`. In this exercise, you will actually work with only `DoublePoly` and `ComplexPoly`.

## Polynomial Plots

The main program is designed to draw plots of real polynomials (using `DoublePoly`) and plots of complex polynomials (using `ComplexPoly`). In setting up this exercise, we decided to give

you the complete code for the plots of real polynomials. We felt that seeing this example in full detail would better help you to understand the issues than a lengthy description in English. We will therefore assume that you have examined the code for the user interface and the code for real polynomial plots in `she11.cpp`. *Do this before reading further in this document.*

The user interface is structured so that:

- The user is asked for the type of polynomial: real or complex
- The user is asked to enter the polynomial data: degree and coefficients
- The user is given an opportunity to select the values to be plotted

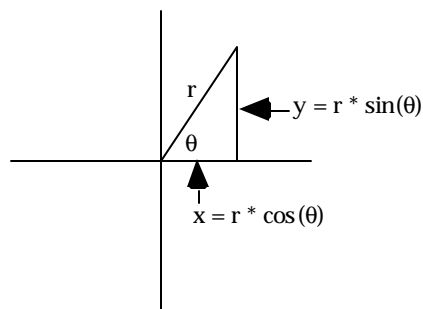
In the case of real polynomials, the user is asked to select the lower and upper bound of the  $x$  values to be plotted. The plot is then a traditional  $y = P(x)$  plot.

In the case of complex polynomials, it is necessary to do something more sophisticated. If  $Z$  is a complex number and  $P$  is a complex polynomial, then  $Z$  and  $P(Z)$  vary over 2-dimensional planes. This means that a “full plot” requires a 4dimensional space that is not easy to simulate on the computer screen.

We solve this plot dilemma for complex polynomials by choosing to do a plot where  $Z$  is restricted to a circle of fixed radius  $r$ . We let  $Z$  vary uniformly over such a circle and collect in a table the corresponding values  $P(Z)$ . We then plot these polynomial values (which are complex numbers). The result is a loop-the-loop plot that shows the image of the polynomial  $P$  on the circle of radius  $r$ .

The table for collecting the  $P(Z)$  values will be a `Polygon2D` object `Data`. Since  $P(Z)$  is of type `Complex` and `Complex` inherits from `Point2D` (which is the type of item a `Polygon2D` stores), you can immediately append  $P(Z)$  values via `Data.Append(P(Z))`. Your main task is to write the loop that fills `Data` so that the  $Z$  values vary uniformly over a circle of radius  $r$ .

To obtain  $Z$  values on a circle of radius  $r$ , you need trigonometry:



This diagram shows how to compute the x and y coordinates of a point on the circle of radius  $r$  at angle  $\theta$ . The traditional trigonometric functions in C assume  $\theta$  is in radians. If you prefer to use trigonometric functions in degrees, use the functions `cosdeg` and `sinddeg` from `MathUtil.h`.

In any case, you may use the trigonometric formulas to arrange that the complex number  $Z$  takes on a sequence of equally spaced values. All of this table building work should be encapsulated in a function:

```
void BuildTableComplex(
    Polygon2D& Data,
    int ValidSize,
    double Radius,
    const ComplexPoly& P);
```

This function should do the following steps:

- Return immediately if `validsize` is less than or equal to 0 since there is nothing to do
- Use `validateNone` to make the polygon `Data` have no valid cells
- Use `ChangeMemory` to make `Data` have a `totalsize` equal to the parameter `validsize`
- Use a loop to `Append` to `Data` the values  $P(Z)$  where  $Z$  is spaced equally around a circle of radius equal to `Radius`. The first and last point  $Z$  should have coordinates  $(\text{Radius}, 0)$ .

Be careful in case the caller passes a `validsize` of 1. This “degenerate case” can lead to division by 0 if you are not cautious.

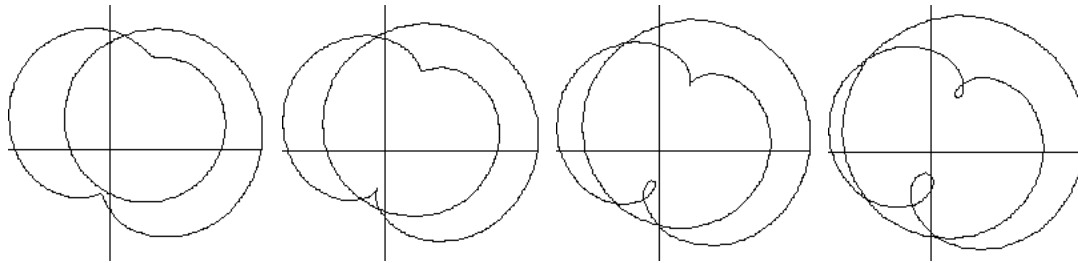
The remaining plot functions for complex polynomials can be constructed in a manner similar to those for real polynomials using the tools in `Graphics.h`. We will leave it to you to examine the code and make the appropriate intelligent modifications.

We want to make one final point on the user interface. If you follow the model provided for real polynomials in the complex case then you will erase the screen and scale anew for each value of the radius. It turns out to be more interesting to scale once and then draw plots for several values of the radius on the same window. This permits comparisons. Play with the sample solution and then try to build this more sophisticated interface. If you are pressed for time then you can fall back on the easier one-radius-one-plot interface.

To give you a feeling for the kind of plots the program will produce, we include four successive plots for the following polynomial:

$$p(z) = i z^4 + z^3 + (1 - 2i) z^2 + (1 + 2i) z + (1 + i)$$

The plots use the following values for the radius: 1.1, 1.2, 1.3, 1.4.



Radius 1.1

Radius 1.2

Radius 1.3

Radius 1.4

## **Conclusion**

From a software design perspective, this project exhibits many features of large scale professional programs. The foundation of the project rests on several general purpose toolkits which are designed to be used in more than one program. To solve the plotting problem, it is necessary to work with multiple data structures which are developed in the various toolkits. At key stages of the solution, it is essential to make deft transformations from one data structure to another. All activities, even input-output, are organized into clean modular procedures and functions so that no portion of the program degenerates into a mess. As you design programs in the future, you should strive for elegance and think in terms of data structures, toolkits, and data transformations.