The code is given below. A compilable version is in the file `input.cpp`.

```
template<class T>
bool Input(T& x)
{// Input a nonnegative value.
   int NumOfAttempts = 3;

   // try at most NumOfAttempts times
   for (int i = 0; i < NumOfAttempts; i++) {
      cout << "Enter a nonnegative value" << endl;
      cin >> x;
      if (x >= 0) return true;
      cout << x << " is not nonnegative" << endl;
      }
   // did not get nonnegative value
   cout << "Sorry, you get only "
        << NumOfAttempts << " chances."
        << endl;
   return false;
}
```

The code is given below. A compilable version is in the file `fact.cpp`.

```
int factorial (int n)
{// Non-recursive program to compute n!
   if (n <= 1) return 1;
   int fact = 2;
   for (int i = 3; i <= n; i++)
      fact *= i;
   return fact;
}
```

The code is given below. The relevant files are `resize1d.h`.

```
template<class T>
void ChangeSize1D(T * &a, int n, int ToSize)
{// Change the size of the one-dimensional
 // array a to ToSize.  n is the number
 // of elements in the array at present.

   // make sure new size is adequate
   if (n > ToSize) throw BadInput();

   // allocate a new array of desired size
   T *temp = new T [ToSize];

   // copy from old space to new space
   for (int i = 0; i < n; i++)
      temp[i] = a[i];

   // free old space
   delete [] a;

   // make a point to new space
   a = temp;
}
```

(a) The code for the input function is given below. This asks for the input as a floating point number and then uses the function `Set` to convert the input floating point number into the `Currency` representation.

```
void Currency::Input()
{// Input currency value in floating format.

   float x;
   cout << "Enter the currency amount as "
        << "a floating point number as in dd.cc or -dd.cc"
        << endl;

   cin >> x;
   Set(x);
}
```

(b) The code for `Subtract` is very similar to that for `Add` and is given below:

```
Currency Currency::Subtract(const Currency& x) const
{// Subtract x and *this.
   long a1, a2, a3;
   Currency ans;
   // convert invoking object to signed integers
   a1 = dollars * 100 + cents;
   if (sgn == minus) a1 = -a1;

   // convert x to signed integer
   a2 = x.dollars * 100 + x.cents;
   if (x.sgn == minus) a2 = -a2;

   a3 = a1 - a2;

   // convert to currency representation
   if (a3 < 0) {ans.sgn = minus; a3 = -a3;}
   else ans.sgn = plus;
   ans.dollars = a3 / 100;
   ans.cents = a3 - ans.dollars * 100;

   return ans;
}
```

(cde) The codes for `Percent`, `Multiply`, and `Divide` are similar. All convert the `Currency` amount into a floating point number, perform the operation, and convert the resulting floating point number back into a `Currency` object. The codes are given below. All codes, test program, and data can be found in the files `curr1.*`.

```
Currency Currency::Percent(float x) const
{// Return x percent of *this.

   float a;
   Currency ans;
   // convert *this to a float
   a = dollars + cents / 100.0;
   if (sgn == minus) a = -a;

   ans.Set((a * x) / 100);

   return ans;
}

Currency Currency::Multiply(float x) const
{// Return x * (*this).

   float a;
   Currency ans;
   // convert *this to a float
   a = dollars + cents / 100.0;
   if (sgn == minus) a = -a;

   ans.Set(a * x);

   return ans;
}

Currency Currency::Divide(float x) const
{// Return (*this)/x.

   float a;
   Currency ans;
   // convert *this to a float
   a = dollars + cents / 100.0;
   if (sgn == minus) a = -a;

   ans.Set(a / x);

   return ans;
}
```

Program 1.30 has only two decisions—`(d > 0)` and `(d == 0)`. The statements in the `// two real roots` block are execute only when the first decision is true; those in the block `// both roots are the same` are executed only when the the first decision is false and the second true; and those in the block `// complex conjugate roots` are executed only when both decisions are false. Therefore, any data set that causes all statements to be excecuted must cause each decision to take on both the truth values true and false. Hence, statement coverage implies decision coverage.

Program 1.20 has three execution paths. Each of these paths contains at least one statement that is not on any other path. So statement coverage must result in execution path coverage.

For each value of n, Program 1.8 has exactly one execution path.