An Introduction to C++ Through Annotated Examples

by

Henry M. Walker

Grinnell College, Grinnell, IA

Program 1: quarts-1.cc

The first program converts a number of quarts to liters, using stream IO from the keyboard.

```
// A simple program to convert a number of quarts to liters
// Version 1:  global variables only

#include <iostream.h>                          /* reference to I/O library */

int quarts;                                    /* declarations */
double liters;                                 /* double = real */

int main (void)                               /* beginning of main program */
{  cout << "Enter volume in quarts: " ;        /* input prompt */

   cin >> quarts;                              /* read */

   liters = quarts / 1.056710 ;                /* arithemtic, assignment */

   cout << quarts << " quarts = "
        << liters << " liters" << endl;        /* write text and new line */

   return 0;                                   /* no errors found when program run*/
}
```

Annotations for `quarts-1.cc`*:*

- Comments in C++ may be denoted in either of two ways:
  § A comment begins with `//` anywhere on a line and continues to the end of the line.
  § A comment may begin anywhere with the symbols `/*`, continuing until the symbols `*/`, on the same or later lines.
- C++ makes use of libraries for many common operations. The statement
  `#include <iostream.h>`
  instructs the machine how to use the iostream operations.
- Variables may be declared almost anywhere within a C++ program. Here, `quarts` and `liters` are declared globally as integer and real variables, respectively. The term `double` specifies a real number, stored using double precision. (The term `float` may be used for single precision, real numbers.)
- Each C++ program contains a driver function/program, called `main`. Here, main uses no input parameters (hence the word void in parentheses). At termination, `main` returns an error code (an integer), so the word `int` appears before `main`. By convention, code 0 indicates no errors were encountered in the program.
- Braces { and } are used in C++ to mark the beginning and ending of blocks. In this case, the braces indicate the statements for the main program.
- Semicolons (;) are used to terminate every statement in C++.
- The equal sign (=) is used for assignment. (We will see later that == is used for the Boolean comparison operator.)
- Arithmetic operations include +, −, *, and / for addition, subtraction, multiplication, and division. For integers, the division operation / yields the integer quotient, while the modulus operation % gives the remainder.
- cin and cout are defined within iostream for simple input and output, respectively. endl is the symbol to move to a new line in output.

Sample Run of `quarts-1.cc`*:*

```
steenrod% g++ -o quarts-1 quarts-1.cc
steenrod% quarts-1
Enter volume in quarts: 4
4 quarts = 3.78533 liters
steenrod% quarts-1
Enter volume in quarts: 1
1 quarts = 0.946333 liters
steenrod% quarts-1
Enter volume in quarts: 3.5
3 quarts = 2.839 liters
```

In this sample run (run on the machine *steenrod*), the program (called `quarts-1.cc`) is compiled and run using the g++ compiler.

Since the program reads quarts as an integer, the input 3.5 is read as the integer 3 in the third run of the program. If additional input were requested within the program, the next characters read would be .5 .

Program 2: quarts-2.cc

Another program to convert a number of quarts to liters.

```
// A simple program to convert a number of quarts to liters
// Version 2:  using local variables and simple error checking

#include <iostream.h>
#include <assert.h>

int main (void)
{  int quarts;                                      /* variables declared   */
   double liters;                                   /* at the start of main */

   cout << "Enter volume in quarts: " ;

   cin >> quarts;
   assert (quarts > 0);                             /* abort if quarts <= 0 */

   liters = quarts / 1.056710 ;

   cout << quarts << " quarts = "
        << liters << " liters" << endl;

   return 0;
}
```

Annotations for `quarts-2.cc`*:*

- Variables can be declared within a function any time before they are used. Thus, variables are commonly declared at the beginning of a function, so they can be used any time within that function.
- The `assert` statement checks the validity of a Boolean expression. If the statement is true, program execution continues without interruption. If the condition is false, however, the program is terminated, with an appropriate error message printed.
- The `assert` statement provides a simple mechanism to verify pre-conditions and post-conditions, although more sophisticated approaches are needed if the program is to resolve the problem and resume processing. (This latter approach is accomplished by exception handlers, to be discussed at a later time.)

Sample Run of `quarts-2.cc`*:*

```
steenrod% g++ -o quarts-2 quarts-2.cc
steenrod% quarts-2
Enter volume in quarts: 4
4 quarts = 3.78533 liters
steenrod% quarts-2
Enter volume in quarts: -2
quarts-2.cc:14: failed assertion `quarts > 0'
IOT trap (core dumped)
```

Program 3: quarts-3.cc

Program illustrating variable declaration and initialization, together with simple error checking.

```
// A simple program to convert a number of quarts to liters
// Version 3:  declaring local variables just as needed

#include <iostream.h>

int main (void)
{  cout << "Enter volume in quarts: " ;

   int quarts;                                  /* declaring quarts just
                                                   before its use */
   cin >> quarts;

   // if value of quarts is invalid, ask user again
   while (quarts <= 0) {
     cout << "Value of quarts must be positive; enter value again: " ;
     cin >> quarts;
   }

   double liters = quarts / 1.056710 ;          /* declaring and initializing
                                                   liters just as needed */
   cout << quarts << " quarts = "
        << liters << " liters" << endl;

   return 0;
}
```

Annotations for `quarts-3.cc`*:*

- C++ allows variables to be declared at any point within a function prior to the use of the function. Thus, in this program, `quarts` is declared after the `cout` line, and `liters` is declared after either the `cout` or `cin` lines.
- C++ also allows a variable to be initialized when it is declared by specifying the variable's initial value as part of the declaration. If this initial value were not given as part of the declaration, then the declaration line for `liters` would be replaced by the two lines:
  `double liters;`
  `liters = quarts / 1.056710;`
- Experienced C++ programmers disagree upon whether variable declarations should normally be placed at the start of a function or whether variables should be declared just before they are needed.
  - § Variables declared at the start of a function often are easier for a programmer to find and check.
  - § When variables are declared as needed, memory allocation may not be needed if a section of code is skipped during specific runs of a program.
- For clarity in the following examples, variables normally are declared at the start of functions, unless there are special reasons to proceed otherwise.
- The `while` loop continues as long as the condition specified is maintained. The general syntax is:
  `while (condition) statement ;` where condition is any Boolean condition and the statement is any single statement. When several statements are to be executed in the body of a loop, they are enclosed in braces { }, as shown in the example.

Sample Run of `quarts-3.cc`*:*

```
steenrod% g++ -o quarts-3 quarts-3.cc
steenrod% quarts-3
Enter volume in quarts: 4
4 quarts = 3.78533 liters
steenrod% quarts-3
Enter volume in quarts: -4
Value of quarts must be positive; enter value again: 0
Value of quarts must be positive; enter value again: 2
2 quarts = 1.89267 liters
```

Program 4: smallest3-1.cc

Simplistic program to find the smallest of three integers.

```
// A simple program to find the smallest of three numbers
// Version 1:  using simple if-then statements

#include <iostream.h>

int main (void)
{  int i1, i2, i3;

   cout << "Program to determine the smallest of three integers" << endl;
   cout << "Enter three integer values:  ";
   cin >> i1 >> i2 >> i3;

   if ((i1 <= i2) && (i1 <= i3))
      cout << "The smallest value is " << i1 << endl;

   if ((i2 < i1) && (i2 <= i3))
      cout << "The smallest value is " << i2 << endl;

   if ((i3 < i1) && (i3 < i2))
      cout << "The smallest value is " << i3 << endl;

   return 0;
}
```

Annotations for `smallest3-1.cc`*:*

- The simple `if` statement has the form:
  `if (condition) statement`
  Here, the "condition" is evaluated, and the "statement" is executed if the "condition" is true. Otherwise, the "statement" is skipped.
- Simple comparisons may use the relations: `<, <=, >, >=, ==` and `!=`. Here `!` means "not", so `!=` means "not equal".
- Technically in C++ any zero value is considered as false, while any nonzero value is considered true.
- *Caution:* Programmers who may have programmed in other languages sometimes mistakenly write `=` for `==`. In C++, the compiler will interpret the `=` as an assignment. Thus, the statement
  `if (i = j) cout << "Equal" ;`
  will assign the value of `j` to `i`. If the value of `j` was 0, then the result of the assignment is considered false, and nothing will be printed. On the other hand, if `j` had a nonzero value, then the assignment to `i` returns a true (i.e., nonzero) value, and output is generated.
  Be careful in writing comparisons to use `==` *rather than* `=`*.*
- Comparisons may be combined with `&&` and `||` for "and" and "or", respectively. The precedence of such operations follows the familiar rules of Boolean algebra. However, when in doubt, it is always safer to use parentheses to clarify meaning.

Sample Run of `smallest3-1.cc`*:*

```
steenrod% g++ -o smallest3-1 smallest3-1.cc
steenrod% smallest3-1
program to determine the smallest of three integers
Enter three integer values:  1 2 3
The smallest value is 1
steenrod% smallest3-1
program to determine the smallest of three integers
Enter three integer values:  1 1 1
The smallest value is 1
steenrod% smallest3-1
program to determine the smallest of three integers
Enter three integer values:  2 1 0
The smallest value is 0
steenrod% smallest3-1
program to determine the smallest of three integers
Enter three integer values:  0 2 0
The smallest value is 0
```

Program 5: smallest3-2.cc

Program to find the smallest of three integers, using two steps.

```
// A simple program to find the smallest of three numbers
// Version 2:  using if-then-else statements with intermediate steps

#include <iostream.h>

int main (void)
{  int i1, i2, i3;
   int smaller, smallest;

   cout << "Program to determine the smallest of three integers" << endl;
   cout << "Enter three integer values:  ";
   cin >> i1 >> i2 >> i3;

   if (i1 <= i2)
        smaller = i1;
   else smaller = i2;

   if (smaller <= i3)
        smallest = smaller;
   else smallest = i3;

   cout << "The smallest value is " << smallest << endl;

   return 0;
}
```

Annotations for `smallest3-2.cc`*:*

- The compound `if` statement has the form:
  `if (condition) statement1`
  `else statement 2`
  As with the simple `if` statement, the "condition" is evaluated and "statement1" is executed if the "condition" is true. In this compound statement, however, "statement2" is executed if the "condition" is false.
- Note that both "statement1" and "statement2" end with semicolons (as with all C++ statements.

This program produces the same output as `smallest3-1.cc`.

Program 6: smallest3-3.cc

Program to find the smallest of three integers, using nested if statements.

```
// A simple program to find the smallest of three numbers
// Version 3:  using nested if-then-else statements

#include <iostream.h>

int main (void)
{  int i1, i2, i3;
   int smallest;

   cout << "Program to determine the smallest of three integers" << endl;
   cout << "Enter three integer values:  ";
   cin >> i1 >> i2 >> i3;

   if (i1 <= i2)
        /* compare i1 and i3; i2 cannot be smallest */
        if (i1 <= i3)
           smallest = i1;
           else smallest = i3;
     else
        /* compare i2 and i3; i1 cannot be smallest */
        if (i2 <= i3)
           smallest = i2;
           else smallest = i3;

   cout << "The smallest value is " << smallest << endl;

   return 0;
}
```

Annotations for `smallest3-3.cc`*:*

- `if` statements can be nested as desired. Since each `if` statement is considered a single entity, such statements may be used in either the "then" or "else" clauses of other `if` statements.

This program also produces the same output as `smallest3-1.cc`.

Program 7: smallest3-4.cc

Program to find the smallest of three integers, using nested if statements with brackets for clarity.

```
// A simple program to find the smallest of three numbers
// Version 4:  using nested if-then-else statements

#include <iostream.h>

int main (void)
{  int i1, i2, i3;
   int smallest;

   cout << "Program to determine the smallest of three integers" << endl;
   cout << "Enter three integer values:  ";
   cin >> i1 >> i2 >> i3;

   if (i1 <= i2)
        /* compare i1 and i3; i2 cannot be smallest */
        {if (i1 <= i3)
           smallest = i1;
           else smallest = i3;
        }
     else
        /* compare i2 and i3; i1 cannot be smallest */
        {if (i2 <= i3)
           smallest = i2;
           else smallest = i3;
        }

   cout << "The smallest value is " << smallest << endl;

   return 0;
}
```

Annotations for `smallest3-4.cc`*:*

- When several statements are to be used within “then” or “else” clauses, braces { } are used to group the statements. Such braces also can be used around single statements for clarity.

This program again produces the same output as `smallest3-1.cc`.

Program 8: quarts-for-1.cc

A program to compute the number of liters for several values of quarts.

```
// A program to compute the number of liters for 1, 2, ..., 12 quarts
// Version 1:  simple table without formatting

#include <iostream.h>

int main (void)
{   int quarts;
    double liters;

    cout << " Table of quart and liter equivalents" << endl
          << endl;                                    /* skip extra line */

    cout << "Quarts        Liters" << endl;

    for (quarts = 1; quarts <= 12; quarts++)
      { liters  = quarts / 1.056710 ;
        cout << "  " << quarts << "          " << liters << endl;
      }

    return 0;
}
```

Annotations for `quarts-for-1.cc`*:*

- The `for` statement follows the syntax:
  `for (initialization; condition; updates) statement;`
  Here, "initialization" (if any) is performed at the start, before any Boolean expressions are evaluated. The Boolean "condition" is evaluated at the top of each loop iteration. If the condition is true, the "statement" is executed. Otherwise, execution of the `for` statement terminates, and processing continues with the statement following the `for`. The "updates" allow any variables or other work to be done after a loop iteration, before the "condition" is evaluated again.
- The ++ operation associated with a variable increments the variable by 1. Thus, `i++` is the same as the statement `i = i + 1`. (Technically, `i++` is a post-increment operation, while `++i` is a pre-increment operation. Here, the incrementing of `i` takes place in a statement by itself, and either operation has the same effect. Further consideration of such subtleties is left to a later time.)
- Note that the `for` statement is more general than in some other languages, as any initialization and updating are possible. For example, the `while` loop is a special case of the `for`, with empty initialization and updating sections: `for (;condition;) statement`; That is, `while (condition) statement;` is equivalent to
  `for (; condition; ) statement;`
- In order to arrange the values approximately in columns, the program prints a sequence of spaces, such as `"  "` and `"          "`. The amount of space allocated for each number depends upon the size and accuracy of the number.

Sample Run of `quarts-for-1.cc`*:*

```
steenrod% g++ -o quarts-for-1 quarts-for-1.cc
steenrod% quarts-for-1
 Table of quart and liter equivalents

Quarts         Liters
  1          0.946333
  2          1.89267
  3          2.839
  4          3.78533
  5          4.73167
  6          5.678
  7          6.62433
  8          7.57067
  9          8.517
  10          9.46333
  11          10.4097
  12          11.356
```

Program 9: quarts-for-2.cc

A table of quart and liter values, using formatted output.

```
// A program to compute the number of liters for 1, 2, ..., 12 quarts
// Version 2:  simple table with formatting of integers and reals

#include <iostream.h>

int main (void)
{   int quarts;
    double liters;

    cout << " Table of quart and liter equivalents" << endl
          << endl;


    cout << "Quarts        Liters" << endl;

    cout.setf(ios::fixed);                      /* used fixed decimal
                                                   representation for float */
    cout.precision(4);                          /* print 4 decimal places */

    for (quarts = 1; quarts <= 12; quarts++)
      { liters  = quarts / 1.056710 ;
        cout.width(4) ;                         /* output width = 4 char */
        cout << quarts ;
        cout.width(16);
        cout << liters << endl;
      }

    return 0;
}
```

Annotations for `quarts-for-2.cc`*:*

- Formatting of integers in C++ is relatively simple: Only a width must be given, and by default integers are right justified in the width given. (If the integer requires more space than given by the width, then the width in expanded as needed.) The width must be specified for each number, as shown in the program.
- Formatting of real numbers in C++ proceeds in several steps. First, the program specifies that a "fixed decimal" representation is needed, using the `cout.setf(ios::fixed)` statement. (Alternatively, the statement `cout.setf(ios::scientific)` prescribes scientific notation for output). Next, a number of digits to the right of the decimal place may be prescribed with `cout.precision(4)`. Finally, the overall width of the number, including the digits to the right of the decimal point, is given following the same syntax as with integers.
- If you want to switch between decimal representations, then you will want to unset one format as you set the next. For example, if a fixed decimal format has already been given, then you would change to a scientific format as follows:
  `cout.unsetf(ios::fixed);`
  `cout.set(ios::scientific);`
- The form of the output depends upon the machine if you omit `cout.setf(---)` or if you have set both the scientific and fixed modes at the same time.

Sample Run of `quarts-for-2.cc`*:*

```
steenrod% g++ -o quarts-for-2 quarts-for-2.cc
steenrod% quarts-for-2
 Table of quart and liter equivalents

Quarts        Liters
   1          0.9463
   2          1.8927
   3          2.8390
   4          3.7853
   5          4.7317
   6          5.6780
   7          6.6243
   8          7.5707
   9          8.5170
  10          9.4633
  11         10.4097
  12         11.3560
```

Program 10: quarts-for-3.cc

Another table of quart and liter equivalents – this time including half quarts as well.

```
// A program to compute the number of liters for 0.5, 1, ..., 5.5, 6.0 quarts
// Version 3:  table expanded to half values for quarts

#include <iostream.h>

int main (void)
{   const double conversion_factor = 1.056710;  /* declaration of constant */
    double quarts, liters;
    cout << " Table of quart and liter equivalents" << endl << endl;
    cout << "Quarts        Liters" << endl;
    cout.setf(ios::fixed);

    for (quarts = 0.5; quarts <= 6.0; quarts += 0.5)
      { liters  = quarts / conversion_factor ;
        cout.precision(1);
        cout.width(5) ;
        cout << quarts ;
        cout.precision(4);
        cout.width(15);
        cout << liters << endl;
      }

    return 0;
}
```

Annotations for `quarts-for-3.cc`*:*

- This program illustrates that the `for` statement is quite general. Here the control variable `quarts` is real, rather than integer. While this seems reasonable in this context, note that roundoff error for real numbers sometimes can provide unanticipated or incorrect results within such loops.
- In C++, the assignment `quarts += 0.5` is shorthand for `quarts = quarts + 0.5`. More generally, the operator `+=` adds the value on the right-hand side of the operator to the variable on the left-hand side.
- Constants are declared within a program by specifying the type of the variable, the initial value, and designating the variable as being a constant, `const`. Thus, the declaration `const double conversion_factor = 1.056710;`
  indicates that the variable `conversion_factor` will be a double precision, real number with value 1.056710, and this value is not allowed to change within the `main` program where this constant is defined.

Sample Run of `quarts-for-3.cc`*:*

```
steenrod% g++ -o quarts-for-3 quarts-for-3.cc
steenrod% quarts-for-3
 Table of quart and liter equivalents

Quarts        Liters
  0.5         0.4732
  1.0         0.9463
  1.5         1.4195
  2.0         1.8927
  2.5         2.3658
  3.0         2.8390
  3.5         3.3122
  4.0         3.7853
  4.5         4.2585
  5.0         4.7317
  5.5         5.2048
  6.0         5.6780
```

Program 11: quarts-for-4.cc

This program illustrates the nesting of `for` loops to produce a table of liter equivalents.

```
// A program to compute the number of liters for gallons and quarts
// Version 4:  table of liters for quarts and gallons

#include <iostream.h>

int main (void)
{  const double conversion_factor = 1.056710;
   int gals, quarts;
   double liters;
   cout << "Table of liter equivalents for gallons and quarts" << endl << endl;
   cout << "                           Quarts" << endl;
   cout << "Gallons          0           1           2           3" << endl;
   cout.setf(ios::fixed);

   cout.precision(4);
   for (gals = 0; gals <= 5; gals++)
     { cout.width(4);
       cout << gals << "     ";
       for (quarts = 0; quarts <= 3; quarts++)
         { liters  = (4.0*gals + quarts) / conversion_factor ;
           cout.width(12) ;
           cout << liters ;
         }
     cout << endl;
      }

   return 0;
}
```

Sample Run of `quarts-for-4.cc`*:*

```
steenrod% g++ -o quarts-for-4 quarts-for-4.cc
steenrod% quarts-for-4
Table of liter equivalents for gallons and quarts

                           Quarts
Gallons          0           1           2           3
   0          0.0000      0.9463      1.8927      2.8390
   1          3.7853      4.7317      5.6780      6.6243
   2          7.5707      8.5170      9.4633     10.4097
   3         11.3560     12.3023     13.2487     14.1950
   4         15.1413     16.0877     17.0340     17.9803
   5         18.9267     19.8730     20.8193     21.7657
```

Program 12: darts.cc

A Monte Carlo simulation to approximate Pi. More precisely, consider a circle C of radius 1 in the plane, centered at the origin. Also, consider the square S of side 2, centered at the origin, so that the corners of the square are at $(\pm1, \pm1)$. If we pick a point P at random in S or in the first quadrant of S. Then the probability that P is also in C is $\pi/4$.

```
//  This program approximates Pi by picking a points in a sqaure and
//  and determining how often they are also in an appropriate circle.

#include <iostream.h>

// libraries for the random number generator
#include <stdlib.h>
#include <time.h>

// Within the stdlib.h library,
//    time returns a value based upon the time of day
//    on some machines, rand returns a random integer between 0 and 2^31 - 1
//        although on some machines rand gives values between 0 and 2^32 - 1
//        and on other machines rand gives values between 0 and 2^15 - 1
//    MaxRandInt is this maximum integer minus 1
//        (Note:  2^32 = 2147483648, 2^31 = 1073741824 and 2^15 = 32768)
//    Use 2^32-1 for SparkStations, 2^15-1 for IBM Xstation 140s and HP 712/60s

const int MaxRandInt = 32767;            /* declaration of program constants*/
const int NumberOfTrials = 5000;

int main (void)
{  int i;
   int counter = 0;                      /* declare and initialize counter */
   double x, y;
   double MaxRandReal = (double) MaxRandInt; /* make MaxRandInt a real */

   cout << "This program approximates Pi by picking " << NumberOfTrials
        << " points in a square and" << endl;
   cout << "counting the number in an appropriate circle." << endl << endl;

   // initialize random number generator
   // change the seed to the random number generator, based on the time of day
   srand (time ((time_t *) 0) );

   // pick points in first quadrant with coordinates between 0 and 1
   // determine how many are in the circle of radius 1
   for (i = 1; i <= NumberOfTrials; i++) {
       x = rand() / MaxRandReal;
       y = rand() / MaxRandReal;
       if (x*x + y*y <= 1) counter++;
   }

   cout << counter << " points were inside the circle, and " << endl ;
   cout << "the approximate value of Pi is " << 4.0 * counter / NumberOfTrials
        << " ." << endl;
   return 0;
}
```

Annotations for `darts.cc`*:*

- This program illustrates the use of the random number generator `rand`, which is based upon a seed value kept internally. `rand` returns pseudo-random positive integers. `srand` is used to set the initial seed value, based upon the built-in `time` function. The range of the integer value returned by `rand` depends upon the local machine executing the code. Some choices are noted in the program.
- The `const` statements before the `main` specify global constants (and other expressions) for use throughout all functions.
- The expression `(double) MaxRandInt` performs type conversions. In the definitions from this program, `MaxRandInt` represents an integer value. The prefix `(double)` converts this to a real value for `MaxRandReal`.
- The function `rand ()` returns a random integer value from the uniform distribution from 0 to `MaxRandInt`. Since the program divides this by a real value (`MaxRandReal`), C++ converts the integer to a real before the division takes place. If we had not declared `MaxRandReal` as a real, but used `MaxRandInt` instead, then we would need to convert both `rand ()` and `MaxRandInt` to integers before performing the division (otherwise we would obtain an integer result, in contrast to what was intended). This alternative conversion could be performed with the revised statements
  ```
  x = (double) rand () / (double) MaxRandReal;
  y = (double) rand () / (double) MaxRandReal;
  ```

Sample Run of `darts.cc`*:*

```
steenrod% g++ -o darts darts.cc
steenrod% darts
This program approximates Pi by picking 5000 points in a square and
counting the number in an appropriate circle.

3919 points were inside the circle, and
the approximate value of Pi is 3.1352 .
steenrod% darts
This program approximates Pi by picking 5000 points in a square and
counting the number in an appropriate circle.

3927 points were inside the circle, and
the approximate value of Pi is 3.1416 .
steenrod% darts
This program approximates Pi by picking 5000 points in a square and
counting the number in an appropriate circle.

3908 points were inside the circle, and
the approximate value of Pi is 3.1264 .
```

Program 13: genfile.cc

This program illustrates the creation and printing of files.

```
// program to write 10 random integers to "integer.file"
//              and 10 random real numbers to "real.file"

// library for file streams
#include <fstream.h>

// library for stream I/O with the keyboard
#include <iostream.h>

// libraries for the random number generator
#include <stdlib.h>
#include <time.h>
//    time, from time.h,  returns a value based upon the time of day
//    rand , from stdlib.h, returns a random integer between 0 and MaxRandInt
//       the value of MaxRandInt is machine dependent:
//           2^32-1 = 2147483647 for SparkStations,
//           2^15-1 = 32767 for IBM Xstation 140s and HP 712/60s
const int MaxRandInt = 32767;

int main (void)
{  ofstream file1, file2; /* output file streams */
   int i;
   int ival;
   double rval;
   double MaxRandReal = (double) MaxRandInt;  /* make MaxRandInt a real */

   // initialize random number generator
   // change the seed to the random number generator, based on the time of day
   cout << "initializing random number generator" << endl;
   srand (time ((time_t *) 0) );

   // place integer values on first file
   cout << "generating file of integers" << endl;
   file1.open ("integer.file");

   for (i = 1; i <= 10; i++)
       { ival = rand ();    /* two lines could be abbreviated */
         file1 << ival << endl;} /* file1 << rand() << endl;       */
   file1.close ();

   // place real values on second file
   cout << "generating file of reals" << endl;
   file2.open ("real.file");

   for (i = 1; i <= 10; i++)
       { file2 << rand() / MaxRandReal << endl;}
   file2.close ();

   cout << "writing of files completed" << endl;
   return 0;
}
```

Annotations for `genfile.cc`*:*

- Stream I/O with files is analogous to stream I/O with the keyboard, except for the initial steps of opening and closing the file.
- Output streams are declared of class `ofstream`.
- Once a stream `f` is declared, it is opened or closed with the statements `f.open (<file-name>)` and `f.close ()`, respectively.
- Output to a file uses the `<<` operator, as with keyboard output to `cout`.

Sample Run of `genfile.cc`*:*

```
steenrod% g++ -o genfile genfile.cc
steenrod% genfile
initializing random number generator
generating file of integers
generating file of reals
writing of files completed
steenrod% cat integer.file
20924
17394
9999
11237
12099
5015
26393
31440
13363
2281
steenrod% cat real.file
0.474837
0.38139
0.808008
0.0730613
0.395367
0.249611
0.0936613
0.918882
0.676473
0.419721
```

Program 14: printfile.cc

This program shows two simple ways to read from files.

```
// program to read 10 integers from "integer.file"
//              and 10 real numbers from "real.file"

#include <iostream.h>
#include <fstream.h>

int main (void)
{  ifstream intfile, realfile;

   int i;
   double r;

   // read integer values from "integer.file"
   cout << "The integers from 'integer.file' follow:" << endl;
   intfile.open ("integer.file");

   intfile >> i;
   while (!intfile.fail())          /* continue until reading fails */
      { cout << i << endl;
        intfile >> i;
      }

   intfile.close () ;

   cout << "Integer file processed" << endl;

   // read real values from "real.file"
   cout << "The real numbers from 'real.file' follow:" << endl;

   realfile.open ("real.file");

   while(realfile >> r) {/* when reading fails, test evaluates to false */
     cout << r << endl;
   }

   realfile.close () ;

   cout << "Real file processed" << endl;

   return 0;
}
```

Annotations for `printfile.cc`*:*

- Input streams are declared of class `ifstream`.
- As with files for output, input streams `f` are opened and closed with the statements `f.open (<file-name>)` and `f.close ()`, respectively.
- Similarly, input from a file uses the `>>` operator, as with keyboard input from `cin`.

- The end of a file can be detected in either of two ways:
  - § The function `f.fail()` indicates whether the machine was successful in its last operation on the file `f`. Thus, when `f.fail()` is used following an attempted read, `f.fail()` will return false when a desired value was read and true otherwise (e.g., at the end of the file).
  - § The expression `(f >> r)` reads a value from `f` to `r`, as long as one exists, and it returns an updated file `f` with the value read extracted. As long as this file is non-empty, it is considered non-zero or `true`. Thus, this updated file may serve as a valid test within a `while` loop. When reading fails, `(f >> r)` returns 0 or `false`, so the `while` loop will terminate.
- Another way to read a file uses the the end-of-file procedure as demonstrated in the following code segment:

```
while (true)
    { intfile >> i; /* if <eof> encountered, i is unchanged */
      if (intfile.eof()) break;
      cout << i << endl;
    }
```

  The Boolean function `f.eof()` returns `true` when the end of the file is encountered and `f.eof()` returns false otherwise. Thus, `f.eof()` indicates an attempt to read beyond the last next character in the file.

  While this approach often works, different systems may handle `<eof>` differently. Thus, the use of `f.eof()` may give unpredictable results, and some authors advise against the use of this procedure.
- As ! is the Boolean "not" operation, the program uses `!f.fail()` to test when more data are present in the file for reading.

Sample Run of `printfile.cc`*:*

```
steenrod% g++ -o printfile printfile.cc
steenrod% printfile
The integers from 'integer.file' follow:
17850
12939
32064
13799
4331
13433
21735
19142
7146
9133
Integer file processed
The real numbers from 'real.file' follow:
0.098941
0.12656
0.877712
0.57268
0.710746
0.809351
0.673574
0.326212
0.289254
0.727073
Real file processed
```

Program 15: max.min-1.cc

This following program computes the maximum, minimum, and average of n real numbers.

```
// A program to read n numbers, compute their maximum,  minimum, and average,
// and to print a specified jth item.
// Version 1:  Using built-in C++ arrays.

#include <iostream.h>

int main (void)
{    int j, n;
     double max, min, sum;

     cout << "Program to process real numbers." << endl;
     cout << "Enter number of reals: ";
     cin >> n;

     double a[n];                /* declare array of n values,
                                    with subscripts 0, ..., n-1  */
     cout << "Enter " << n << " numbers: " ;
     for (j = 0; j < n; j++)
          cin >> a[j];           /* subscripts given in brackets [ ] */

     sum =  max = min = a[0];    /* right to left assignment operator */

     for (j = 1; j < n; j++)
       { if (a[j] > max)
            max = a[j];
         if (a[j] < min)
            min = a[j];
         sum += a[j];
       }

     cout << "Maximum:  " << max << endl;
     cout << "Minimum:  " << min << endl;
     cout << "Average:  " << sum/n << endl << endl;

     cout << "Enter the index (1..n) of the number to be printed: ";
     cin >> j;
     cout << "The " << j << "th number is "  << a[j-1] << endl;

     return 0;
}
```

Annotations for `max.min-1.cc`*:*

- Arrays within C++ are declared by indicating the number of array elements in square brackets [ ], as in the statement `double a[n]`. Also, since declarations may be made anywhere within a program, C++ allows an array size to be read first and then used within an array declaration.
- Individual array elements are accessed by placing a subscript in brackets, such as `a[0]` or `a[j]`.

Sample Run of `max.min-1.cc`*:*

```
steenrod% g++ -o max.min-1 max.min-1.cc
steenrod% max.min-1
Program to process real numbers.
Enter number of reals: 6
Enter 6 numbers:  1 2 4 6 8 9
Maximum:  9
Minimum:  1
Average:  5

Enter the index (1..n) of the number to be printed: 5
The 5th number is 8
steenrod% max.min-1
Program to process real numbers.
Enter number of reals: 5
Enter 5 numbers: 3 2 1 4 5
Maximum:  5
Minimum:  1
Average:  3

Enter the index (1..n) of the number to be printed: 200
The 200th number is 1.4013e-45
```

- C++ does not perform bounds checking when accessing array elements. Thus, nonsense is printed for `a[200]` when only `a[0]` through `a[9]` are declared and given values. Note especially that no *Subscript Out Of Range* error message is generated for the attempt to access `a[200]`.
- The output $1.4013e-45$ may be interpreted as representing the number 1.4013×10^{-45}, written in scientific notation. Normally, `cout` will use this scientific notation for very large numbers and for numbers very close to 0 in order to avoid printing numbers with many zeros.

Program 16: max.min-2.cc

Use of vectors, defined by AP CS, behaves much like arrays in C++, but includes the checking of subscript bounds.

```
// A program to read n numbers, compute their maximum,  minimum, and average,
// and to print a specified jth item.
// Version 2:  Using "safe" C++ vectors from the AP CS subset
//             rather than arrays.

#include <iostream.h>
#include "apvector.cpp"                /* Vectors defined in local file */

int main (void)
{    int j, n;
     double max, min, sum;

     cout << "Program to process real numbers." << endl;
     cout << "Enter number of reals: ";
     cin >> n;

     apvector<double> a(n);            /* declare vector of n values,
                                          with subscripts 0, ..., n-1  */
     cout << "Enter " << n << " numbers: " ;
     for (j = 0; j < n; j++)
         cin >> a[j];           /* subscripts given in brackets [ ] */

     sum =  max = min = a[0];  /* right to left assignment operator */

     for (j = 1; j < n; j++)
       { if (a[j] > max)
            max = a[j];
         if (a[j] < min)
            min = a[j];
         sum += a[j];
       }

     cout << "Maximum:  " << max << endl;
     cout << "Minimum:  " << min << endl;
     cout << "Average:  " << sum/n << endl << endl;

     cout << "Enter the index (1..n) of the number to be printed: ";
     cin >> j;
     cout << "The " << j << "th number is "  << a[j-1] << endl;

     return 0;
}
```

Annotations for `max.min-2.cc`*:*

- Vectors, as defined by AP CS, are declared by identifying the type of data to be stored in the array together with the number of array elements. For example, in the statement, `vector<double> a(n);`, an array of `n` floating point numbers is created, and these values are accessible through subscripts 0 through n-1.

- When using vectors, the program must include information about the appropriate AP CS material. This is done with the beginning statement `#include "vector.cc"`. Here, the use of double quotes about `vector.cc` indicates that this material is located in the current user's directory.
- Actually, vector information is included in two files, `vector.h` and `vector.cc`, and the second file contains a statement to include the first. On many systems, the compiler is sophisticated enough, so that `vector.cc` can be compiled separately. In such circumstances, a user's program, such as `max.min-2.cc`, includes only the header file, with the statement `#include "vector.h"`. The compiler then links in the compiled material from a compiled version of `vector.cc`. Since this alternative was unavailable on the author's machine, the entire `vector.cc` file is included here.
- Use of these vectors is the same as for C++ arrays, although all accesses to vectors includes an explicit test that a subscript is between 0 and n-1, inclusive. If the subscript is not within range, an error message is printed, and the program is terminated.
- Details of AP CS vectors include the concepts of C++ classes and objects and will be discussed later.

Sample Run of `max.min-2.cc`*:*

```
steenrod% g++ -o max.min-2 max.min-2.cc
steenrod% max.min-2
Program to process real numbers.
Enter number of reals: 6
Enter 6 numbers: 1 2 4 6 8 9
Maximum:  9
Minimum:  1
Average:  5

Enter the index (1..n) of the number to be printed: 5
The 5th number is 8
steenrod% max.min-2
Program to process real numbers.
Enter number of reals: 5
Enter 5 numbers: 3 2 1 4 5
Maximum:  5
Minimum:  1
Average:  3

Enter the index (1..n) of the number to be printed: 200
The 200th number is Illegal vector index: 199 max index = 4
IOT trap (core dumped)
```

- The output for this program is the same as for `max.min-1.out` when all subscripts are within the declared range. However, an error message is generated and the program is terminated when any attempt is made to access an element outside this range.

Program 17: trap-1.cc

This program approximates Pi as the area of one-fourth of a circle of radius 2.

```
// A program for approximating the area under a function y = f(x)
// between a and b using the Trapezoidal Rule.
//    The Trapezoidal Rule divides [a, b] into n evenly spaced intervals
//    of width W = (b-a)/n.  The approximate area is
//    W*(f(a)/2 + f(a+W) + f(a+2W) + ... + f(a+(n-2)W) + f(a+(n-1)W) + f(b)/2)
// Version 1:  Approximating area under sqrt(4-x^2) on [0, 2].
//    As this is 1/4 of circle of radius 2, result should be Pi.

#include <iostream.h>
#include <math.h>

const double a = 0.0;      /* limits for the area under y = f(x) */
const double b = 2.0;      /* area will be computed under f(x) on [a,b] */

double f(double x)
/* function to be used in the area approximation */
{  return (sqrt(4.0 - x*x));
}

int main (void)
{  int  n;
   double width, sum, xvalue, area;

   cout << "Program approximates the area under a function using the "
        << "Trapezoidal Rule." << endl;
   cout << "Enter number of subintervals to be used: ";
   cin  >> n;

   width = (b - a) / n;

   /* compute the sum in the area approximation */
   sum = (f(a) + f(b)) / 2.0;            /* first and last terms in sum */
   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);

   area = sum * width;

   cout << "The approximate area is " << area << endl;

   return 0;
}
```

Annotations for `trap-1.cc`*:*

- The main program uses the Trapezoidal Rule to approximate the area under a function $y = f(x)$ on the interval $[a, b]$.
- The function $f(x)$ is defined as a separate function. Specifically, `double f` indicates that the function f will return a floating point number, and the header `(double x)` specifies that f will have a single, floating point parameter.
- The format of function f is similar to that of main. Both contain a header (with possible parameters), followed by a function body in braces, { }.

- Here, the entire function is declared first, before the main program.
- A function computes and returns a value, and this value is designated using a `return` statement. Here, the value returned is `(sqrt(4.0 - x*x))`.
- The *math.h* library of C++ contains many common mathematics functions, including $\sin(x)$, $\cos(x)$, $\tan(x)$, $exp(x)$, $log(x)$, $log10(x)$, $pow(x, y)$, and $sqrt(x)$. Here, $exp(x)$ computes e^x, $log(x)$ computes the natural log, $log10(x)$ computes the common log, and $pow(x, y)$ computes x^y.
- In the computation of the Trapezoidal Rule, the constants a and b are specified as constants using `#define` statements in this program.

Sample Run of `trap-1.cc`*:*

```
steenrod% g++ -o trap-1 -lm trap-1.cc
steenrod% trap-1
Program approximates the area under a function using the Trapezoidal Rule.
Enter number of subintervals to be used: 20
The approximate area is 3.12846
```

- In compiling programs involving functions in *math.h*, some compilers require the use of the `-lm` option, as shown.

Program 18: trap-2.cc

This program uses a function for computations involving the Trapezoidal Rule.

```
// Approximating the area under sqrt(4-x^2) on [0, 2] using the Trapezoidal Rule.
// Version 2:  A function performs the area computation.

#include <iostream.h>
#include <math.h>

double f(double x);
/* function to be used in the area approximation */

double area(double a, double b, int n);
/* Approximation of area under f(x) on [a, b] using the Trapezoidal Rule */

int main (void)
{  int n;
   cout << "Program approximates the area under a function using the "
        << "Trapezoidal Rule." << endl;
   cout << "Enter number of subintervals to be used: ";
   cin  >> n;
   cout << "The approximate area is " << area (0.0, 2.0, n) << endl;
   return 0;
}

double f(double x)
/* function to be used in the area approximation */
{  return (sqrt(4.0 - x*x));
}

double area (double a, double b, int n)
/* Finding area via the Trapezoidal Rule */
{  double width = (b - a) / n;
   double sum = (f(a) + f(b)) / 2.0;   /* first and last terms in sum */
   double xvalue;

   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);
   return (sum * width);
}
```

Annotations for `trap-2.cc`*:*

- Functions either may be declared in full before a main program or a header may be defined at the start with details given later. This program illustrates the second approach. At the start, functions `f` and `area` are identified with a header, which is terminated by a semicolon to indicate that details are forthcoming later. This header contains information about the type of value returned by the function (`double` in each case here) and about the number and type of the function parameters.
- When function details are given later, the header information is repeated, followed by the specific code elements.
- Within a function, variables may be declared and initialized, following the same rules illustrated in previous examples for the main program.
- This program produces the same output as the previous program.

Program 19: trap-3.cc

This program computes the Trapezoidal Rule using a procedure.

```
// Approximating the area under sqrt(4-x^2) on [0, 2] using the Trapezoidal Rule.
// Version 3:  A procedure performs the area computation.

#include <iostream.h>
#include <math.h>

double f(double x);
/* function to be used in the area approximation */

void compute_area(double a, double b, int n, double &area);
/* Approximation of area under f(x) on [a, b] using the Trapezoidal Rule */

int main (void)
{  int n;
   double area;
   cout << "Program approximates the area under a function using the "
        << "Trapezoidal Rule." << endl;
   cout << "Enter number of subintervals to be used: ";
   cin  >> n;

   compute_area(0.0, 2.0, n, area);
   cout << "The approximate area is " << area << endl;
   return 0;
}

double f(double x)
/* function to be used in the area approximation */
{  return (sqrt(4.0 - x*x));
}

void compute_area (double a, double b, int n, double &area)
/* Finding area via the Trapezoidal Rule */
{  double width = (b - a) / (double) n;
   double sum = (f(a) + f(b)) / 2.0;   /* first and last terms in sum */
   double xvalue;

   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);

   area = sum * width;
}
```

Annotations for `trap-3.cc`*:*

- A function returning no value (a void result) is a procedure.
- By default, C++ parameters are passed by value (at least for simple variables). Thus, in declaring `double f(double x)` and calling `f(a)`, the value of a is copied to x before the computation of f proceeds.
- If a value is to be returned in a parameter (e.g., as an area), then parameter passage by reference may be used, by adding an ampersand `&` in the function header.
- Once again, this program produces the same output as *trap-1.cc*.

Program 20: trap-4.cc

Another area computation, involving a home-grown square-root function which tests that square roots are taken of non-negative numbers only.

```
// Approximating the area under y = f(x) on [a, b] using the Trapezoidal Rule.
// Version 4:  Approximating area under sqrt(4-x^2) on [0, 2].
//    Here sqrt is computed via Newton's Method, rather than through math.h.

#include <iostream.h>
#include <assert.h>
const double a = 0.0;
const double b = 2.0;

double sqrt(double r)
/* function to compute square root of r */
{ if (r == 0.0)
     return 0.0;         /* the square root of 0.0 is a special case */
  assert (r > 0.0);      /* negative square roots are not defined */
  double change = 1.0;    /* square roots are found using Newton's method */
  double x = r;
  while ((change > 0.00005) || (change < -0.0005))
    { change = (x*x - r) / (2*x);
      x -= change;
    }
  return x;
}

double f(double x)
/* function to be used in the area approximation */
{  return sqrt(4.0 - x*x);
}

int main (void)
{  int  n;
   double width, sum, xvalue, area;

   cout << "Program approximates the area under a function using the "
        << "Trapezoidal Rule." << endl;
   cout << "Enter number of subintervals to be used: ";
   cin  >> n;

   width = (b - a) / n;

   /* compute the sum in the area approximation */
   sum = (f(a) + f(b)) / 2.0;           /* first and last terms in sum */
   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);

   area = sum * width;
   cout << "The approximate area is " << area << endl;
   return 0;
}
```

Annotations for `trap-4.cc`*:*

- This program again produces the same output as *trap-1.cc*.

Program 21: trap-5.cc

Using the same area function for the computation of areas under two functions.

```
// Approximating the area under several functions using the Trapezoidal Rule.
// Version 5:  An area function has numeric and functional parameters.

#include <iostream.h>
#include <math.h>

double circle(double x);
/* function for a circle of radius 2, centered at the origin */

double parabola(double x);
/* function for the standard parabola y = x^2 */

double area(double a, double b, int n, double f (double));
/* Approximation of area under f(x) on [a, b] using the Trapezoidal Rule */

int main (void)
{  int number;
   cout << "Program approximates the area under several functions using the "
        << "Trapezoidal Rule." << endl;
   cout << "Enter number of subintervals to be used: ";
   cin  >> number;

   cout << endl;
   cout << "Approximation of 1/4 area of circle of radius 2 is "
        << area (0.0, 2.0, number, circle) << endl << endl;
   cout << "Approximation of area under y = x^2 between 1 and 3 is "
        << area (1.0, 3.0, number, parabola) << endl << endl;
   return 0;
}

double circle(double x)
/* function for a circle of radius 2, centered at the origin */
{  return (sqrt(4.0 - x*x));
}

double parabola(double x)
/* function for the standard parabola y = x^2 */
{ return x*x;
}

double area (double a, double b, int n, double f (double))
/* Finding area via the Trapezoidal Rule */
{  double width = (b - a) / n;
   double sum = (f(a) + f(b)) / 2.0;   /* first and last terms in sum */
   double xvalue;

   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);
   return (sum * width);
}
```

Annotations for `trap-5.cc`*:*

- This program represents a variation of program *trap-2.cc*. Here, `area` computes the area under $y = f(x)$ on $[a, b]$, using n trapezoids, and all of these elements (`a, b, n, f`) are passed as parameters.
- To declare a function parameter, the function is given a formal name (e.g., `f`), and information concerning its parameters and return type are specified. In the example, the header of `area` contains the information
  `double f (double)`
  which indicates that a function will be passed to `area`. This function will take one double precision, real parameter and will return a double precision, real number. Further, when the details of `area` are defined, this function will be referred to as `f` (just as the numbers `a, b` and `n` will be used within `area`).
- When `area` is called, an appropriate function name is specified for `f`, just as numbers are specified for `a, b` and `n`. Thus, in the call
  `area (0.0, 2.0, number, circle)`
  `a` is given the value 0.0, `b` is given the value 2.0, `n` is given the value *number*, and `f` will refer to the function *circle*. Whenever `f` is mentioned within `area` during the execution of this call, the function *circle* will be used. Similarly, for the call
  `area (1.0, 3.0, number, parabola)`
  the function *parabola* will be used whenever `f` appears.
- In using function parameters, the actual functions (`circle` or `parabola`) must be of the type as the formal parameter (`f`). In this case, the functions use one double parameter, and they returned a double.

Sample Run of `trap-5.cc`*:*

```
steenrod% g++ -o trap-5 -lm trap-5.cc
steenrod% trap-5
Program approximates the area under several functions using the Trapezoidal Rule.
Enter number of subintervals to be used: 100

Approximation of 1/4 area of circle of radius 2 is 3.14042

Approximation of area under y = x^2 between 1 and 3 is 8.6668
```

Code Segment 22: rational.h

The following code segment, called a *header file*, defines some simple elements of a new, rational-number data type. In object-oriented programming, such a data type is called a *class*. Subsequent code segments show how a class is used and how implementation details are specified.

```
// Definition of a rational number class; a rational number object
// will be a fraction, with a numerator and a non-zero denominator.
// Throughout, any operation which results in a denominator being zero
// generates an error.

// A class is a type (or abstract data type), with both data and operations

class Rational
{  /* the following extends the standard arthmetic operator + to rationals */
  friend Rational operator+ (Rational r1, Rational r2);

public:  /* this presents the user's view of rational numbers */

  /* The first operations allow varying declarations of rational numbers */
  Rational (void); /* A rational with no designated value will be 0 or 0/1 */
  Rational (int num); /* A rational with a numerator only will be num/1    */
  Rational (int num, int denom);  /* The rational num/denom                 */

  /* I/O operations */
  void read(void); /* reads a rational of the form num / denom */
  void print(void);

  /* Return rational number as a decimal */
  double eval (void);

private:  /* the following details are for implementation only and
             are not accessible by users */
  int numerator;
  int denominator;
};
```

Annotations for `rational.h`*:*

- Conceptually, *rational numbers* are fractions, with an integer numerator and an integer denominator. *rational.h* provides a simple, user's view of such numbers, allowing a programmer to create rational numbers, use elementary I/O operations, and perform addition and decimal division.
- In object-oriented programming generally and specifically in C++, a data type combining both data and operations is called a *class*. This `Rational` class contains 3 main elements.
  - § An initial section (where the operator + is defined) extends familiar operations or references common operations using various details of this class.
  - § A *public* section defines operations for use by an applications programmer.
  - § A *private* section defines data types and operations involved in the implementation of this class. Such details may not be used directly by applications; any explicit reference to these elements in a user's program will yield a compiler-generated error.

  While all three of these elements may be specified in the definition of a class, any of the above elements also may be omitted.

- The definition of the operation + given here allows the familiar addition symbol to be used for rational numbers in addition to its built-in capabilities for integers and real numbers. This use of an operator for several purposes (or data types) is called *operator overloading* or just *overloading*. The definition here is analogous to any definition of a function, as shown in previous examples. The use of the modifier `friend` grants the implementer of this operator permission to access the internal fields `numerator` and `denominator`.
- The first part of any public section usually includes operations which indicate how initialization will take place. In C++, these *constructor* operations always have the same name as the class itself. Here, initialization may take any of three forms, depending upon how many parameters are specified. The following examples illustrate how these constructors may be used.
  - § A variable declaration
    `Rational a;`
    would declare `a` to be a variable of type `Rational`. (In the jargon of object-oriented programming, `a` is an object of class `Rational`.) Here, `a` is declared without any further information, so the first constructor is used. The comments for this constructor indicate that `a` will be initialized to zero, conceptually in the form 0/1.
  - § A statement
    `Rational b(5);`
    also would declare `b` to be an object of class `Rational`. In this declaration, the parameter `5` is given, so the second constructor will be used. The comments for that constructor indicate that `b` will be initialized to the number 5, in the form 5/1.
  - § The declaration
    `Rational c(5, 8);`
    would declare `c` to be an object of class `Rational`. With two parameters given, the third constructor will be used, yielding the value 5/8 according to the comments.

  These three variations of the constructor operation provide a second example of overloading – this time for the name `Rational`.
- Other public operations, defined here and available for general programming, will include `read`, `print`, and `eval`.
- Details of all of these operations are not provided in the class definition itself – they must be given elsewhere.
- As another part of object-oriented jargon, the operations defined within a class are called *methods*. Thus, `Rational`, `read`, `print`, and `eval` are all methods for objects of this new class.
- The private section provides implementation information which may be needed by a compiler (e.g., to allocate space or to provide linkage addresses). Here, the integer components of a rational number (i. e., the `numerator` and `denominator`) are defined.
- Note that the comments at the start of the file indicate that the denominator of a rational number cannot be zero, and an error is reported whenever any operation (e.g., `read` or variable declaration) leads to a zero denominator. This restriction that denominator is non-zero is one motivation for placing the fields, `numerator` and `denominator`, in the private section. With this placement, a user cannot access the denominator directly, possibly changing it to zero. Thus, this class can enforce the condition that the denominator of a rational number can never be zero.
- Since this header file only defines logical operations, it only identifies what capabilities might be available for rational numbers. It does not produce any output by itself.

Code Segment 23: rational.cc

An implementation file specifies the details of how class operations are implemented.

```
// Implementation of a rational number class

#include <iostream.h>
#include <assert.h>
#include "rational.h"

// Define the + operation
Rational operator+ (Rational r1, Rational r2)
{  Rational r;
   r.numerator = (r1.numerator * r2.denominator) +
                 (r1.denominator * r2.numerator);
   r.denominator = r1.denominator * r2.denominator;
   return r;
}

// Define the various ways to declare rational numbers
Rational::Rational (void)
{ /* construct/initialize the fraction 0/1 */
  numerator = 0;
  denominator = 1;
}

Rational::Rational (int num)
{ /* construct/initialize the fraction num/1 */
  numerator = num;
  denominator = 1;
}

Rational::Rational (int num, int denom)
{ /* construct/initialize the fraction num/denom ,
     an error results if denom == 0 */
  numerator = num;
  assert (denom != 0);
  denominator = denom;
}

// Define the I/O operations
void Rational::read(void)
{ /* only numbers of the form num/denom, with denom!=0 are legal */
  char divSign;      /* use divSign to read the / symbol within the input */
  cin >> numerator >> divSign >> denominator;
  assert (divSign == '/');
  assert (denominator != 0);
}

void Rational::print(void)
{  cout << numerator << " / " << denominator;
}

// Define decimal division
double Rational::eval (void)
{ return ((double) numerator / (double) denominator);
}
```

Annotations for `rational.cc`*:*

- Each operation defined for class `Rational` is defined, following much the same syntax as other functions.
- In providing the details of a specific operation or method, the function header includes the prefix `Rational::` to indicate that we are providing code for the method for that class. Thus, the header `Rational::print(void)` indicates that this code is defining the `print` method within class `Rational`.
- Since the methods `Rational`, `read`, `print`, and `eval` are all associated with an object or variable of class `Rational`, every such object will have its own `numerator` and `denominator`. These fields are referenced as part of these methods.
- This file only defines the details of various methods – it does not use the methods in an application. Thus, this file does not produce any output. On the other hand, this implementation file can be compiled for linking into later programs as needed.

Sample Run of `rational.cc`*:*

```
steenrod% g++ -c rational.cc
```

- In using the specific compiler illustrated here, the `-c` option indicates that the file is to be compiled but not linked to other files.
- The above compiler produces a file `rational.o` which can be linked with application programs, as they are developed.

Program 24: rat-driver-1.cc

A program that demonstrates the use of the rational number class.

```
// Program demonstrating the use of a simple rational-number class
#include <iostream.h>
#include "rational.h"

int main(void)
{ Rational z;                    /* by default, z is zero */
  Rational i(4);                 /* simple integer initialization for 4/1 */
  Rational r(3, 5);              /* rational number 3/5 */

  /* printing of initialized rational numbers */
  cout << "The following rational numbers have been defined" << endl;
  cout << "  z            i(4)         r(3, 5)" << endl;
  z.print ();                    /* use z's print operation */
  cout << "          ";
  i.print ();                    /* this print refers to object i */
  cout << "          ";
  r.print ();                    /* parentheses () are needed for functions */
  cout << endl;
  cout.setf(ios::fixed);
  cout.precision(5);
  cout << "The corresponding decimal values are:" << endl;
  cout << z.eval() << "       " << i.eval() << "        " << r.eval()
       << endl << endl;

  /* using z's read operation */
  cout << "Enter a fraction:  ";
  z.read ();

  /* using the addition operation */
  Rational s = z + r;

  /* more printing */
  cout << "Two additional rational numbers are:" << endl;
  cout << "Your z       sum z + r" << endl;
  z.print ();
  cout << "         ";
  s.print ();
  cout << endl << endl;
  return 0;
}
```

Annotations for `rat-driver-1.cc`*:*

- C++ initializes variables when they are declared, following the details specified within constructors. Thus, `z, i,` and `r` are all initialized, following the appropriate constructor, at the beginning of the program when they are declared.
- When using methods (operations) related to an object (variable), a dot notation is used to associate the object with its method. Thus, `z.print ()` invokes the `print` method for the object `z`, and `z.eval ()` invokes `z`'s `eval` method.
- Since the operation + was overloaded for rational numbers, it may be used in the familiar context `z + r`, as the compiler now knows how to apply this operation to elements in this new data type.

Sample Run of `rat-driver-1.cc`*:*

```
steenrod% g++ -o rat-driver-1 rat-driver-1.cc rational.o
steenrod% rat-driver-1
The following rational numbers have been defined
  z             i(4)         r(3, 5)
0 / 1          4 / 1          3 / 5
The corresponding decimal values are:
0.00000       4.00000        0.60000

Enter a fraction:  2/3
Two additional rational numbers are:
Your z       sum z + r
2 / 3         19 / 15

steenrod% rat-driver-1
The following rational numbers have been defined
  z             i(4)         r(3, 5)
0 / 1          4 / 1          3 / 5
The corresponding decimal values are:
0.00000       4.00000        0.60000

Enter a fraction:  2/0
rational.cc:43: failed assertion `denominator != 0'
IOT trap (core dumped)
steenrod% rat-driver-1
The following rational numbers have been defined
  z             i(4)         r(3, 5)
0 / 1          4 / 1          3 / 5
The corresponding decimal values are:
0.00000       4.00000        0.60000

Enter a fraction:  2   /   3
Two additional rational numbers are:
Your z       sum z + r
2 / 3         19 / 15

steenrod% rat-driver-1
The following rational numbers have been defined
  z             i(4)         r(3, 5)
0 / 1          4 / 1          3 / 5
The corresponding decimal values are:
0.00000       4.00000        0.60000

Enter a fraction:  2 * 3
rational.cc:42: failed assertion `divSign == '/''
IOT trap (core dumped)
```

- In compiling the program `rat-driver-1.cc`, the machine is told to link the details of rational numbers, as found in `rational.o`.
- The second and fourth runs illustrate that error checking is performed as advertised.
- The third run illustrates that reading using the >> operator strips initial blanks, so that spaces may be placed before the `/` character or before either integer without generating errors.

Code Segment 25: rational2.h

A simpler form for operations, using default parameters.

```
// Definition of a rational number class; a rational number object
// will be a fraction, with a numerator and a non-zero denominator.
// Throughout, any operation which results in a denominator being zero
// generates an error.

// A class is a type (or abstract data type), with both data and operations

class Rational
{  /* the following extends the standard arthmetic operator + to rationals */
  friend Rational operator+ (Rational r1, Rational r2);

public:  /* this presents the user's view of rational numbers */

  /* Constructor */
  Rational (int num = 0, int denom = 1);  /* the rational number num/denom */
  /* if no parameters are given, use 0/1
     if the second parameter is omitted, use num/1 */

  /* I/O operations */
  void read(void); /* reads a rational of the form num / denom */
  void print(void);

  /* Return rational number as a decimal */
  double eval (void);

private:  /* the following details are for implementation only and
             are not accessible by users */
  int numerator;
  int denominator;
};
```

Annotations for `rational2.h`*:*

- This header file specifies the same constructors as the previous version in a simpler form, using default parameters. In particular, the statement
  `Rational (int num = 0, int denom = 1);`
  provides for two parameters, `num` and `denom`. If no parameters are supplied during initialization, however, this header specifies that the values 0 and 1 should be used for these parameters. Similarly, if only one parameter is given, this header assumes the value should correspond to the first parameter (`num`), and the default value of 1 will be used for `denom`.
- A similar use of default values for parameters may be used with any C++ function.

Code Segment 26: rational2.cc

A simpler and more efficient implementation of rational numbers.

```
// Implementation of a rational number class

#include <iostream.h>
#include <assert.h>
#include "rational2.h"

// Define the + operation
Rational operator+ (Rational r1, Rational r2)
{  Rational r;
   r.numerator = (r1.numerator * r2.denominator) +
                 (r1.denominator * r2.numerator);
   r.denominator = r1.denominator * r2.denominator;
   return r;
}

Rational::Rational (int num, int denom)
  :numerator (num),            /* initialize numerator and denominator */
   denominator (denom)         /* with the specified values num and denom */
{ /* an error results if denom == 0 */
  assert (denom != 0);
}

// Define the I/O operations
void Rational::read(void)
{ /* only numbers of the form num/denom, with denom!=0 are legal */
  char divSign;      /* use divSign to read the / symbol within the input */
  cin >> numerator >> divSign >> denominator;
  assert (divSign == '/');
  assert (denominator != 0);
}

void Rational::print(void)
{  cout << numerator << " / " << denominator;
}

// Define decimal division
double Rational::eval (void)
{ return ((double) numerator / (double) denominator);
}
```

Annotations for `rational2.cc`*:*

- Since this implementation goes with the revised header file, default values from the header are assumed in this implementation. C++ allows the default parameters to be repeated in the implementation header, with the statement
  `Rational::Rational (int num = 0, int denom = 1)`
  but the above code shows that there is no need to repeat the default values again in the implementation. (An error is generated, however, if new and different default values are given in the implementation.)

- This revised constructor `Rational` also illustrates a more efficient way to construct rational numbers. In particular, the first version contained the lines
  ```
  numerator = num;
  assert (denom != 0);
  denominator = denom;
  ```
  While this code correctly initializes a rational number and checks that the denominator is nonzero, the execution actually proceeds in several steps. First, space is allocated for the integer fields `numerator` and `denominator`. Second, these fields are initialized with a default value for integers (e.g., 0). Third, new values (`num` and `denom`) are copied to the respective fields.

  In this revised version, `numerator` and `denominator` are given values before the brace {. In C++, this means that the values specified are used to initialize the fields, before any default value for integers is stored. Thus, the second step of initialization from the previous code is skipped in this revised code.

- This revised code may be used with the previous program `rat-driver-1.cc` simply by including the new header file `rational2.h` and by linking with the new compiled code `rational2.o`. The output of the user program remains unchanged.

Code Segment 27: rational3.h

A variation of the rational number data type, allowing << and >> for I/O and automatically reducing rationals to lowest terms.

```
// Definition of a rational number class; a rational number object
// will be a fraction, with a numerator and a non-zero denominator.
// Throughout, any operation which results in a denominator being zero
// generates an error.

// A class is a type (or abstract data type), with both data and operations

class Rational
{  /* the following extends the standard arthmetic operator + to rationals */
  friend Rational operator+ (Rational r1, Rational r2);

  /* Overload familiar I/O operations */
  friend istream& operator>> (istream& istr, Rational& r);
  friend ostream& operator<< (ostream& ostr, Rational r);


public:  /* this presents the user's view of rational numbers */

  /* Constructor */
  Rational (int num = 0, int denom = 1);  /* the rational number num/denom */
  /* if no parameters are given, use 0/1
     if the second parameter is omitted, use num/1 */

  /* Return rational number as a decimal */
  double eval (void);

private:  /* the following details are for implementation only and
             are not accessible by users */
  int numerator;
  int denominator;
  int gcd (int a, int b);  /* function to find the greatest common divisor */
};
```

Annotations for `rational3.h`*:*

- In this version of the rational number data type, the I/O stream operators << and >> are overloaded for use with rational numbers. This eliminates the need for the separate operations `read` and `print`, which were defined in previous versions of the rational number data type.
- In this revision, rational numbers will be kept as fractions reduced to lowest terms. To accomplish this, a greatest common division function, gcd, is useful. Since this function is intended for use within this class only, not by any outside application, `gcd` is declared as a private operation. Application programs may neither access nor redefine this `gcd` function. (Of course, these programs may use their own *gcd* function, but any such function will be considered distinct from the one defined here.)

Code Segment 28: rational3.cc

The implementation details for extending << and >> and for reducing fractions to lowest terms.

```
// Implementation of a rational number class
#include <iostream.h>
#include <assert.h>
#include "rational3.h"

// Define the + operation
Rational operator+ (Rational r1, Rational r2)
{ Rational r;
  r.numerator = (r1.numerator * r2.denominator) +
                (r1.denominator * r2.numerator);
  r.denominator = r1.denominator * r2.denominator;

  /* reduce the fraction stored by eliminating common factors */
  int common_fac = r.gcd (r.numerator, r.denominator);
  r.numerator = r.numerator / common_fac;
  r.denominator = r.denominator / common_fac;
  return r;
}

/* Overload familiar I/O operations */
istream& operator>> (istream& istr, Rational& r)
{ /* only numbers of the form num/denom, with denom!=0 are legal */
  char divSign;      /* use divSign to read the / symbol within the input */
  istr >> r.numerator >> divSign >> r.denominator;
  assert (divSign == '/');
  assert (r.denominator != 0);

  /* reduce the fraction stored by eliminating common factors */
  int common_fac = r.gcd (r.numerator, r.denominator);
  r.numerator = r.numerator / common_fac;
  r.denominator = r.denominator / common_fac;

  return istr;            /* the revised input stream is returned */
}

ostream& operator<< (ostream& ostr, Rational r)
{ /* first use the << operation for integers and strings */
  ostr << r.numerator << " / " << r.denominator;
  return ostr;           /* return the revised output stream */
}

// Constructor
Rational::Rational (int num, int denom)
{ /* an error results if denom == 0 */
  assert (denom != 0);

  /* reduce the fraction stored by eliminating common factors */
  int common_fac = gcd (num, denom);
  numerator = num / common_fac;
  denominator = denom / common_fac;
}
```

```
// Define decimal division
double Rational::eval (void)
{ return ((double) numerator / (double) denominator);
}

// Define the greatest common divisor function
int Rational::gcd (int a, int b)
/*  Pre-condition:  b != 0;
    Since ince b always == denom when called in this package,
    this pre-condition is always met within this class */
{  if (a == 0)
      return 1;
   /* if the numbers are nonzero, the classical Euclidean algorithm is used
   to compute the gcd  -- see a mathematics text for more information */
   int x = a;
   int y = b;
   int r = y % x;
   while (r != 0)
     { y = x;
       x = r;
       r = y % x;
     }
   return x;
}
```

Annotations for `rational3.cc`*:*

- The I/O operations << and >> are defined for output and input streams, respectively. Within each operation, work is built upon similar operations for numbers or integers. In the case of input, error checking is added, and the number entered is reduced to lowest terms.
- The gcd function follows the standard Euclidean algorithm of finding successive remainders to find the greatest common divisors. For more information on this algorithm, the reader is referred to a mathematics textbook.
- In reducing a rational number to lowest terms, the greatest common factor of the numerator and denominator is computed. Then both the numerator and the denominator are divided by this common factor.

Program 29: rat-driver-3.cc

This program uses the new rational number data type.

```
// Program demonstrating the use of a simple rational-number class
#include <iostream.h>
#include "rational3.h"

int main(void)
{ Rational z;                   /* by default, z is zero */
  Rational i(4);                /* simple integer initialization for 4/1 */
  Rational r(3, 5);             /* rational number 3/5 */

  /* printing of initialized rational numbers */
  cout << "The following rational numbers have been defined" << endl;
  cout << "  z            i(4)         r(3, 5)" << endl;
  cout << z << "          " << i << "          " << r << endl;
  cout.setf(ios::fixed);
  cout.precision(5);
  cout << "The corresponding decimal values are:" << endl;
  cout << z.eval() << "       " << i.eval() << "        " << r.eval() << endl;

  /* using cin to read operation */
  cout << "Enter two fractions (a and b):  ";
  Rational a, b;
  cin >> a >> b;

  /* using the addition operation */
  Rational s = a + b;

  /* more printing */
  cout << "Two additional rational numbers are:" << endl;
  cout << "Your a        Your b        s = a + b" << endl;
  cout << a << "          " << b << "         " << s << endl << endl;
  return 0;
}
```

Annotations for `rat-driver-3.cc`*:*

- Since the operations, << and >>, have been extended to rational numbers, I/O streams may be used for this new data type as for built-in types.

Sample Run of `rat-driver-3.cc`*:*

```
steenrod% g++ -c rational3.cc
steenrod% g++ -o rat-driver-3 rat-driver-3.cc rational3.o
steenrod% rat-driver-3
The following rational numbers have been defined
  z            i(4)         r(3, 5)
0 / 1          4 / 1         3 / 5
The corresponding decimal values are:
0.00000       4.00000       0.60000
Enter two fractions (a and b):  2 / 4   3 / 5
Two additional rational numbers are:
Your a        Your b        s = a + b
1 / 2          3 / 5         11 / 10
```