Chapter 1
C++ Basics

Key Terms
functions
program
`int main()`
`return 0`
identifier
case-sensitive
keyword or reserved word
declare
floating-point number
unsigned
assignment statement
uninitialized variable
assigning `int` values to *`double`* variables
mixing types
integers and Booleans
literal constant
scientific notation or floating-point notation
quotes
C-string
string
escape sequence
`const`
modifier
declared constant
mixing types
precedence rules
integer division
the % operator
negative integers in division
type cast
type coercion
increment operator
decrement operator
v++ versus ++v
`cout`
expression in a `cout` statement
spaces in output
newline character
deciding between `\n` and `endl`
format for *`double`* values
magic formula
outputting money amounts

`cerr`
`cin`
how `cin` works
separate numbers with spaces
when to comment
`#include`,
preprocessor
namespace
using namespace

Brief Outline

1.1 Introduction to C++
- Origins of the C++ Language
- C++ and Object-Oriented Programming
- The Character of C++
- C++ Terminology
- A Sample C++ Program

1.2 Variables, Expressions, and Assignment Statements
- Identifiers
- Variables
- Assignment Statements
- More Assignment Statements
- Assignment Compatibility
- Literals
- Escape Sequences
- Naming Constants
- Introduction to the string class
- Arithmetic Operators and Expressions
- Integer and Floating-Point Division
- Type Casting
- Increment and Decrement Operators

1.3 Console Input/Output
- Output Using cout
- New Lines in Output
- Formatting for Numbers with a Decimal Point
- Output with cerr
- Input Using cin

1.4 Program Style
- Comments

1.5 Libraries and Namespaces
- Libraries and include Directives
- Namespaces

1. Introduction and Teaching Suggestions

This chapter introduces the students to the history of the C++ language and begins to tell them about what types of programs can be written in C++ as well as the basic structure of a C++ program. During the discussions on compilation and running a program, care should be taken to explain the process on the particular computer system that the students will be using, as different computing/development environments will each have their own specific directions that will need to be followed. In the development of this instructor's manual, a majority of the programs have been compiled using g++ 4.0.2 on Ubuntu Linux, g++ 3.4 on cygwin, and Visual Studio .NET 2008 using Windows Vista. There are significant differences between the development environments and sometimes on the compilers as well. Anyone that is still using Visual Studio 6 is strongly recommended to upgrade to the latest patch level, as the original compiler contained many errors that will prevent programs in this book from compiling.

Simple programming elements are then introduced, starting with simple variable declarations, data types, assignment statements, and eventually evolving into arithmetic expressions. String variables are not introduced in detail until Chapter 9, but an introduction is given and could be elaborating upon if desired. If time allows, a discussion of how the computer stores data is appropriate. While some of the operations on the primitives are familiar to students, operations like modulus (%) are usually not and require additional explanation. Also, the functionality of the increment and decrement operators requires attention. The issue of type casting is also introduced, which syntactically as well as conceptually can be difficult for students. Some students that have previously learned C may use the old form of type casting (e.g. (int)), but should be encouraged to use the newer form (e.g. static_cast<int>).

The section on programming style further introduces the ideas of conventions for naming of programmatic entities and the use and importance of commenting source code. Commenting is a skill that students will need to develop and they should begin commenting their code from the first program that they complete. Indentation is also discussed. However, many development environments actually handle this automatically.

2. Key Points

Compiler. The compiler is the program that translates source code into a language that a computer can understand. Students should be exposed to how compiling works in their particular development environment. If using an IDE, it is often instructive to show command-line compiling so students can a sense of a separate program being invoked to translate their code into machine code. This process can seem "magical" when a button is simply pressed in an IDE to compile a program.

Syntax and Semantics. When discussing any programming language, we describe both the rules for writing the language, i.e. its grammar, as well as the interpretation of what has been written, i.e. its semantics. For syntax, we have a compiler that will tell us when we have made a mistake. We can correct the error and try compiling again. However, the bigger challenge may lie in the understanding of what the code actually means. There is no "compiler" for telling us if

the code that is written will do what we want it to, and this is when the code does not do what we want, it most often takes longer to fix than a simple syntax error.

Names (Identifiers). C++ has specific rules for how you can name an entity in a program. These rules are compiler enforced, but students should be able to recognize a correct or incorrect identifier. Also, there are common conventions for how C++ names its programming entities. Variable names begin with a lower case letter while constants are in all upper case. However, these conventions are not compiler enforced. The book and the source code for C++ itself use these conventions and it is helpful for students to understand that if they follow them, their code is easier for others to read.

Variable Declarations. C++ requires that all variables be declared before they are used. The declaration consists of the type of the variable as well as the name. You can declare more than one variable per line.

Assignment Statements with Primitive Types. To assign a value to a variable whose type is a primitive, we use the assignment operator, which is the equals (=) sign. Assignment occurs by first evaluating the expression on the right hand side of the equals sign and then assigning the value to the variable on the left. Confusion usually arises for students when assigning the value of one variable to another. Showing that x = y is not the same as y = x is helpful when trying to clear up this confusion.

Initializing a Variable in a Declaration. We can and should give our variables an initial value when they are declared. This is achieved through the use of the assignment operator. We can assign each variable a value on separate lines or we can do multiple assignments in one line.

Assignment Compatibility. Normally, we can only assign values to a variable that are of the same type as we declared the variable to be. For example, we can assign an integer value to an integer variable. However, we can also assign a char value to an integer due to the following ordering:

$$\text{char} \rightarrow \text{short} \rightarrow \text{int} \rightarrow \text{long} \rightarrow \text{float} \rightarrow \text{double}$$

Values on the left can be assigned to variables whose types are to the right. You cannot go in the other direction. In fact, the compiler will give an error if you do. However, you may receive a compiler warning message about loss of precision.

What is Doubled? This discussion concerns how floating-point numbers are stored inside the computer. A related topic would be to show the conversion of these numbers into the format (e.g. IEEE 754 into two's complement) that the computer uses.

Escape Sequences. When outputting strings, the \ character is used to escape the following character and interpret it literally. It is useful to use this to show how to output " or \ along with untypable characters, such as newlines or tabs.

I/O with cin, cout, cerr. This discussion shows how to input values from the keyboard and output them to the screen. Under a Unix system it is easy to show the difference between cout and cerr by redirecting the output.

Namespaces. This section illustrates how functions can exist within different namespaces. Students that have previously learned C sometimes have difficulty with namespaces. Some instructors wish to avoid "using namespace std;" and instead prefer to start their programs only using the constructs that are actually used (e.g. using std::cin;). The book starts using the entire namespace but gravitates toward the latter toward the end of the book.

Naming Constants. Program style is important and varies from one person to another. However, having programmatic style standards makes programs easier to read for everyone. One of these style points can be the naming of constants in the program. The convention that is introduced is the one that is common to C++ and the text.

3. Tips

Error Messages. One of the most frustrating parts of learning how to program is learning to understand the error messages of the compiler. These errors, which are commonly called syntax errors, frustrate students. It is helpful to show some common error messages from the compiler so that students have a frame of reference when they see the errors again themselves. Also important for students to note is that even though C++ states the line number that the error occurred on, it is not always accurate. Run-time errors occur when the program has been run. For this section, creating a simple statement that divides by zero can generate one such error. The last type of error, a logic error is one that is hardest to spot because on the surface the program runs fine, but does not produce the correct result. The difference between x++ and ++x could be used here to illustrate a logic error.

4. Pitfalls

Uninitialized Variables. Variables that are declared but not assigned a value are uninitialized. It is good programming practice to always initialize your variables. It may be instructive to output the contents of uninitialized variables to show the unpredictable values they may contain. Uninitialized variables used in computation can cause errors in your program and the best way to avoid these errors is to always give variables an initial value. This can most easily be done right when the variable is declared.

Round-off Errors in Floating-Point Numbers. One of the places to show the fallibility of computers is in the round-off errors that we experience when using floating point numbers. This topic relates to why the type is named double and also deals with the representation of floating point numbers in the system. This problem occurs because not all floating-point numbers are finite, a common example being the decimal representation of the fraction 1/3. Because we can only store so many digits after the decimal points, our computation is not always accurate. A discussion of when this type of round off error could be a problem would be appropriate to highlight some of the shortcomings of computing.

Division with Whole Numbers. In C++, all of the arithmetic operations are closed within their types. Therefore, division of two integers will produce an integer, which will produce an answer that most students do not expect. For example, the integer 1 divided by the integer 2 will produce the integer 0. Students will expect 0.5. One way to get a floating-point answer out of integer division is to use typecasting. Another way is to force floating-point division by making one of the integers a floating-point number by placing a ".0" at the end. Experimentation with this issue is important to show the different results that can be obtained from integer division.

Order of Evaluation. The order of evaluation of subexpressions is not guaranteed. For example in (n + (++n)) there will be a different result if ++n is evaluated first or second. The best advice is to avoid such scenarios. Precedence of operators is discussed in chapter 2.

5. Programming Projects Answers

1. Metric - English units Conversion

A metric ton is `35,273.92` ounces. Write a C++ program to read the weight of a box of cereal in ounces then output this weight in metric tons, along with the number of boxes to yield a metric ton of cereal.

Design: To convert `14` ounces (of cereal) to metric tons, we use the 'ratio of units' to tell us whether to divide or multiply:

$$14 \text{ ounces} * \frac{1}{35{,}273.92} \frac{\text{metric tons}}{\text{ounce}} = 0.000397 \text{ metric tons}$$

The *explicit* use of units will simplify the determination of whether to divide or to multiply in making a conversion. Notice that ounces/ounce becomes unit-less, so that we are left with metric ton units. The number of ounces will be very, very much larger than the number of metric tons. It is then reasonable to divide the number of ounces by the number of ounces in a metric ton to get the number of metric tons.

Let `metricTonsPerBox` be the weight of the cereal contained in the box in metric tons, and let `ouncesPerBox` be the weight of the cereal contained in the box in ounces. Then in C++ the formula becomes:

```
const double ouncesPerMetricTon = 35272.92;
metricTonsPerBox = ouncesPerBox/ouncesPerMetricTon;
```

This is metric tons PER BOX, whence the number of BOX(es) PER metric ton should be the reciprocal of this number:

```
  boxesPerMetricTon = 1 / metricTonsPerBox;
```

Once this analysis is made, the code proceeds quickly:

```
// Purpose: To convert cereal box weight from ounces to
// metric tons and to compute number of boxes of cereal that
// constitute a metric ton of cereal.
#include <iostream>
using namespace std;
const double oncesPerMetricTon = 35272.92;

int main()
{
  double ouncesPerBox, metricTonsPerBox, boxesPerMetricTon;
  cout << "enter the weight in ounces of your"
       << "favorite cereal:" <<endl;
  cin >> ouncesPerBox;
  metricTonsPerBox =
                ouncesPerBox / oncesPerMetricTon;
  boxesPerMetricTon = 1 / metricTonsPerBox;
  cout << "metric tons per box = "
       << metricTonsPerBox << endl;
  cout << "boxes to yield a metric ton = "
       << boxesPerMetricTon << endl;

  return 0;
}
```

A sample run follows:

```
~/AW$ a.out
enter the weight in ounces of your favorite cereal:
14
metric tons per box = 0.000396905
boxes to yield a metric ton = 2519.49

enter the weight in ounces of your favorite cereal:
```

2. Lethal Dose

Certain artificial sweeteners are poisonous at some dosage level. It is desired to know how much soda a dieter can drink without dying. The problem statement gives no information about how to scale the amount of toxicity from the dimensions of the experimental mouse to the dimensions of the dieter. Hence the student must supply some more or less reasonable assumption as basis for the calculation.

This solution supposes the lethal dose is directly proportional to the weight of the subject, hence

$$\text{lethal\_dose\_dieter} = \text{lethal\_dose\_mouse} * \frac{\text{weight\_of\_dieter}}{\text{weight\_of\_mouse}}$$

This program accepts weight of a lethal dose of sweetener for a mouse, the weight of the mouse, and the weight of the dieter, then calculates the amount of sweetener that will just kill the dieter, based on the lethal dose for a mouse in the lab. If the student has problems with grams and pounds, a pound is 454 grams.
It is interesting that the result probably wanted is a *safe* number of cans, while all the data can provide is the minimum lethal number. Some students will probably realize this, but my experience is that most will not, or will not care. I just weighed a can of diet pop and subtracted the weight of an empty can. The result is about 350 grams. The label claims 355 ml, which weighs very nearly 355 grams. To get the lethal number of cans from the number of grams of sweetener, you need the number of grams of sweetener in a can of pop, and the concentration of sweetener, which the problem assumes 0.1% , that is, a conversion factor of 0.001.

```
gramsSweetenerPerCan = 350 * 0.001 = 0.35 grams/can
cans = lethalDoseForDieter / (0.35 grams / can)
```

```
//Input: lethal dose of sweetener for a lab mouse, weights
// of  mouse and dieter, and concentration of sweetener in a
// soda.
//Output: lethal dose of soda as a number of cans.
//Assumption: lethal dose proportional to weight of subject
//    concentration of sweetener in the soda is 1/10 percent

#include <iostream>
using namespace std;
const double concentration = .001; // 1/10 of 1 percent
const double canWeight = 350;
const double gramsSweetenerPerCan =
               canWeight * concentration; //units: grams/can
int main()
{
    double lethalDoseMouse, lethalDoseDieter,
           weightMouse, weightDieter; //units: grams
    double cans;
    cout << "Enter the weight of the mouse in grams"
         << endl;
    cin >> weightMouse;
    cout << "Enter the lethal dose for the mouse in‟
         << ” grams " << endl;
    cin >> lethalDoseMouse;
    cout << "Enter the desired weight of the dieter in”
         <<‟ grams " << endl;
```

```
    cin >> weightDieter;
    lethalDoseDieter =
    lethalDoseMouse * weightDieter/weightMouse;
    cout << "For these parameters:\nmouse weight: "
         << weightMouse
         << " grams " << endl
         << "lethal dose for the mouse: "
         << lethalDoseMouse
         << " grams" << endl
         << "Dieter weight: " << weightDieter
         << " grams " << endl
         << "The lethal dose in grams of sweetener is: "
         << lethalDoseDieter << endl;
    cans = lethalDoseDieter / gramsSweetenerPerCan;
    cout << "Lethal number of cans of pop: "
         << cans << endl;

    return 0;
}
```

```
A typical run follows:
Enter the weight of the mouse in grams
15
Enter the lethal dose for the mouse in grams
100
Enter the desired weight of the dieter, in grams
45400
For these parameters:
mouse weight: 15 grams
lethal dose for the mouse: 100 grams
Dieter weight: 45400 grams
The lethal dose in grams of sweetener is: 302667
Lethal number of cans of pop: 864762
```

3. Pay Increase

The workers have won a 7.6% pay increase, effective 6 months retroactively. This program is to accept the previous salary, then outputs the retroactive pay due the employee, the new annual salary, and the new monthly salary. Allow user to repeat as desired. The appropriate formulae are:

```
newSalary = salary * ( 1 + INCREASE );
monthly = salary / 12;
retroactive = (newSalary – salary) / 2;
```

The code follows:

```
//Given 6 months retroactive, 7.6% pay increase,
//Input: old salary
//Output: new annual and monthly salary, retroactive pay due
#include <iostream>
using namespace std;
const double INCREASE = 0.076;

int main()
{
   double oldSalary, salary, monthly, retroactive;
   char ans;
   cout << "Enter current annual salary." << endl
        << "I'll return new annual salary, monthly "
        << "salary, and retroactive pay." << endl;
   cin >> oldSalary;
   salary = oldSalary * ( 1 + INCREASE );
   monthly = salary / 12;
   retroactive = (salary - oldSalary) / 2;
   cout << "new annual salary " << salary << endl;
   cout << "new monthly salary " << monthly << endl;
   cout << "retroactive salary due: "
        << retroactive << endl;
   return 0;
}
17:50:12:~/AW$ a.out
Enter current annual salary.
100000
I'll return new annual salary, monthly salary, and retroactive
pay.
new annual salary 107600
new monthly salary 8966.67
retroactive salary due: 53800
```

4. Consumer Loan

This problem is an example where the student needs to have a solution in hand before going to the computer. Algebra will solve the problem based on what is given. First compute the "discounted loan face value" in terms of the other things known:

```
InterestRate and NoYearsToRun:
AmountRecevied = FaceValue - discount
Discount = FaceValue * AnnualInterestRate * NoYearsToRun;
AmountReceived =
     FaceValue - FaceValue*AnnualInterestRate*NoYearsToRun
```

Then ask them to solve for `FaceValue`:

```
FaceValue =  AmountReceived / (1
      AnnualInterestRate*NoYearsToRun)
```

Writing a program for this will be easy.

5. Occupancy of Meeting Room beyond file-law limits.

Program reads the maximum occupancy, then the then the number of attendees. If the maximum has been exceeded, announces either that the room has its maximum occupancy exceeded and how many must leave, else it announces that the maximum has not been exceeded.
This problem uses the `if-else` illustrated in Display 1.1, and a one-sided `if` statement,

```
//Maximum Occupancy
#include <iostream>
using namespace std;
int main()
{
  int maxOccupancy;
  int numberOccupants;
  cout << "Enter the Maximum occupancy for the room.\n";
  cin >> maxOccupancy;
  cout << maxOccupancy << endl;
  cout << "Enter the number of occupants of the room.\n";
  cin >> numberOccupants;
  cout << numberOccupants << endl;

  if(numberOccupants > maxOccupancy)
    cout << "ATTENTION: MAXIMUM OCCUPANCY EXCEEDED. \n"
         << "THE LAW REQUIRES "
         << numberOccupants - maxOccupancy
         << " PERSONS TO LEAVE THE ROOM IMMEDIATELY\n";
  else
    cout <<"The number of occupants does not exceed "
         << "the legal maximum.\n";
  return 0;
}
/*
A typical run is:

Enter the Maximum occupancy for the room.
250
Enter the number of occupants of the room.
375
ATTENTION: MAXIMUM OCCUPANCY EXCEEDED.
THE LAW REQUIRES 125 PERSONS TO LEAVE THE ROOM IMMEDIATELY

Another run:

Enter the Maximum occupancy for the room.
250
Enter the number of occupants of the room.
225
The number of occupants does not exceed the legal maximum.

*/
```

6. Overtime Pay

This problem uses the `if-else` illustrated in Display 1.1, and a one-sided `if` statement, which is just an `if-else` with the `else` clause omitted.

```
//pay roll problem:
//Inputs: hoursWorked, number of dependents
//Outputs: gross pay, each deduction, net pay
//
//Outline:
//
//regular payRate = $16.78/hour for hoursWorked <= 40 //hours.
//For hoursWorked > 40 hours,
//   overtimePay = (hoursWorked - 40) * 1.5 * payRate.
//FICA (social security) tax rate is 6%
//Federal income tax rate is 14%.
//Union dues = $10/week .
//If number of dependents >= 3
//    $35 more is withheld for dependent health insurance.
//
#include <iostream>
using namespace std;

const double PAYRATE = 16.78;

const double SS_TAX_RATE = 0.06;
const double FedIRS_RATE = 0.14;
const double STATETAX_RATE = 0.05;
const double UNION_DUES = 10.0;
const double OVERTIME_FACTOR = 1.5;
const double HEALTH_INSURANCE = 35.0;

int main()
{
  double hoursWorked, grossPay, overTime, fica,
                      incomeTax, stateTax, netPay;
  int numberDependents, employeeNumber;

  //set the output to two places, and force .00 for cents
  cout.setf(ios::showpoint);
  cout.setf(ios::fixed);
  cout.precision(2);

  // compute payroll
  cout << "Enter employee SSN (digits only,"
       << " no spaces or dashes) \n";
  cin >> employeeNumber ;
  cout << endl << employeeNumber << endl;
  cout << "Please enter hours worked \n";
  cin >> hoursWorked;
  cout << endl << hoursWorked << endl;
  cout << "Please enter number of dependants." << endl;
  cin >> numberDependents;
  cout << endl << numberDependents << endl << endl;

  if (hoursWorked <= 40 )
    grossPay = hoursWorked * PAYRATE;
  else
```

```
    overTime =
       (hoursWorked - 40) * PAYRATE * OVERTIME_FACTOR;

  if (hoursWorked > 40)
    grossPay = 40 * PAYRATE + overTime;

  fica = grossPay * SS_TAX_RATE;
  incomeTax = grossPay * FedIRS_RATE;
  stateTax = grossPay * STATETAX_RATE;
  netPay = grossPay - fica - incomeTax
                          - UNION_DUES - stateTax;

  if (numberDependents >= 3)
    netPay = netPay - HEALTH_INSURANCE;

  //now print report for this employee:
  cout << "Employee number: "
       << employeeNumber << endl;
  cout << "hours worked: " << hoursWorked << endl;
  cout << "regular pay rate: " << PAYRATE << endl;

  if (hoursWorked > 40)
    cout << "overtime hours worked: "
         << hoursWorked - 40 << endl;

  if (hoursWorked > 40)
    cout << "with overtime premium: "
         << OVERTIME_FACTOR << endl;

  cout << "gross pay: " << grossPay << endl;
  cout << "FICA tax withheld: " << fica << endl;
  cout << "Federal Income Tax withheld: "
       << incomeTax << endl;
  cout << "State Tax withheld: " << stateTax << endl;

  if (numberDependents >= 3)
    cout << "Health Insurance Premium withheld: "
         << HEALTH_INSURANCE << endl;

  cout << "Flabbergaster's Union Dues withheld: "
       << UNION_DUES << endl;
  cout << "Net Pay: " << netPay << endl << endl;

  return 0;
}
/*
Two typical runs follow:

Enter employee SSN (digits only, no spaces or dashes)
234567890
234567890
Please enter hours worked
37.00
37.00
Please enter number of dependants.
1
1
```

```
Employee number: 234567890
hours worked: 37.00
regular pay rate: 16.78
gross pay: 620.86
FICA tax withheld: 37.25
Federal Income Tax withheld: 86.92
State Tax withheld: 31.04
Flabbergaster's Union Dues withheld: 10.00
Net Pay: 455.64

Enter employee SSN (digits only, no spaces or dashes)
234567890
234567890
Please enter hours worked
54.00
54.00
Please enter number of dependants.
4
4

Employee number: 234567890
hours worked: 54.00
regular pay rate: 16.78
overtime hours worked: 14.00
with overtime premium: 1.50
gross pay: 1023.58
FICA tax withheld: 61.41
Federal Income Tax withheld: 143.30
State Tax withheld: 51.18
Health Insurance Premium withheld: 35.00
Flabbergaster's Union Dues withheld: 10.00
Net Pay: 722.69

/*
```

7. Calories

One way to measure the amount of energy that is expended during exercise is to use metabolic equivalents (MET). Here are some METS for various activities:

Running 6 MPH:	10 METS
Basketball:	8 METS
Sleeping:	1 MET

The number of calories burned per minute may be estimated using the formula:

$$\text{Calories/Minute} = 0.0175 \times \text{MET} \times \text{Weight(Kg)}$$

Write a program that inputs a subject's weight in pounds, the number of METS for an activity, and the number of minutes spent in that activity, and then outputs the estimate for total number of calories burned. One kilogram is equal to 2.2 pounds.

```
//calories.cpp
//This program calculates the amount of energy expended
//  using the concept of metabolic equivalents.  The formula
//  used is Calories/Minute = 0.0175 * MET * WeightInKilos
//One Kg = 2.2 Pounds

#include <iostream>
#include <cstdlib>

using namespace std;

const double POUNDS_TO_KG = 1 / 2.2;

// ======================
//     main function
// ======================
int main()
{
 //
 // Variable declarations
 double mets;
 double weight_lb;
 double mins;
 double weight_kg;
 double calories;

 cout << endl << "Welcome to the calorie calculator." << endl;

 // --------------------------------
 // ----- ENTER YOUR CODE HERE -----
 // --------------------------------

 cout << "Enter your weight in pounds: " << endl;
 cin >> weight_lb;

 cout << "Enter the number of METS for the activity: " << endl;
 cin >> mets;

 cout << "Enter the number of minutes spent exercising: " << endl;
 cin >> mins;

 // Convert from pounds to kilograms
 weight_kg = POUNDS_TO_KG * weight_lb;

 // Formula to compute calories
 calories = 0.0175 * mets * weight_kg * mins;

 // Output total calories
 cout << "You burned an estimated " << calories << " calories." << endl <<
endl;

 // --------------------------------
 // --------- END USER CODE --------
 // --------------------------------
```

8. Babylonian

The Babylonian algorithm to compute the square root of a number ***n*** is as follows:

1. Make a ***guess*** at the answer (you can pick n/2 as your initial guess).
2. Compute ***r = n / guess***
3. Set ***guess = (guess +r) / 2***
4. Go back to step 2 for as many iterations as necessary. The more that steps 2 and 3 are repeated, the closer ***guess*** will become to the square root of ***n***.

Write a program that inputs an integer for ***n***, iterates through the Babylonian algorithm five times, and outputs the answer as a double to two decimal places. Your answer will be most accurate for small values of ***n***.

CodeMate Hints: Make guess a double
CodeMate Hints: Typecast using static_cast<double>(n) described on page 23
CodeMate Hints: See page 31 for the magic formula on setting the precision for outputting doubles.

```
//babylonian.cpp
//
//This program uses the Babylonian algorithm, using five
//  iterations, to estimate the square root of a number n.

#include <iostream>
#include <cstdlib>

using namespace std;

// ======================
//     main function
// ======================
int main()
{
 // --------------------------------
 // ----- ENTER YOUR CODE HERE -----
 // --------------------------------

 // Variable declarations
 double guess;
 int n;
 double r;

 cout << endl << "This program makes a rough estimate for square roots."
      << endl;

 cout << "Enter an integer to estimate the square root of: " << endl;
 cin >> n;

 // Initial guess
 guess = static_cast<double>(n)/2;

 // First guess
```

```
 r = static_cast<double>(n)/ guess;
 guess = (guess+r)/2;

 // Second guess
 r = static_cast<double>(n)/ guess;
 guess = (guess+r)/2;

 // Third guess
 r = static_cast<double>(n)/ guess;
 guess = (guess+r)/2;

 // Fourth guess
 r = static_cast<double>(n)/ guess;
 guess = (guess+r)/2;

 // Fifth guess
 r = static_cast<double>(n)/ guess;
 guess = (guess+r)/2;

 // Code to set the precision to 2 decimal places
 cout.setf(ios::fixed);
 cout.setf(ios::showpoint);
 cout.precision(2);

 // Output the fifth guess
 cout << "The estimated square root of " << n << " is " << guess
       << endl << endl;

 // --------------------------------
 // --------- END USER CODE --------
 // --------------------------------
```

9. Coupons

The video game machines at your local arcade output coupons depending upon how well you play the game. You can redeem 10 coupons for a candy bar or 3 coupons for a gumball. You prefer candy bars to gumballs. Write a program that inputs the number of coupons you win and outputs how many candy bars and gumballs you can get if you spend all of your coupons on candy bars first, and any remaining coupons on gumballs.

CodeMate Hints: Use the % operator to compute the number of coupons remaining

```
//coupons.cpp
//
//  This program computes the number of candy bars and gumballs you
//  can get by redeeming coupons at an arcade.  10 coupons is
//  redeemable for candy bars and 3 coupons for gumballs.  You
//  would like as many candy bars as possible and only use
//  remaining coupons on gumballs.

#include <iostream>
#include <cstdlib>

using namespace std;
```

```
// ======================
//     main function
// ======================
int main()
{

 // Variable declarations
 int num_coupons = 0;
 int num_coupons_after_candybars = 0;
 int num_coupons_after_gumballs = 0;
 int num_candy_bars = 0;
 int num_gumballs = 0;

 cout << endl << "Candy calculator.  Enter number of coupons to redeem:"
      << endl;
 //
 // --------------------------------
 // ----- ENTER YOUR CODE HERE -----
 // --------------------------------

 cin >> num_coupons;

 // Integer division below discards any remainder
 num_candy_bars = num_coupons / 10;

 // Calculate remaining coupons
 num_coupons_after_candybars = num_coupons % 10;

 // Calculate gumballs
 num_gumballs = num_coupons_after_candybars / 3;

 // Calculate any leftover coupons
 num_coupons_after_gumballs = num_coupons_after_candybars % 3;

 // Output the number of candy bars and gumballs
 cout << "Your " << num_coupons << " coupons can be redeemed for " <<
      num_candy_bars << " candy bars and " <<
      num_gumballs << " gumballs with " <<
      num_coupons_after_gumballs << " coupons leftover."
      << endl << endl;

 // --------------------------------
 // --------- END USER CODE --------
 // --------------------------------
```

10. Freefall

Write a program that allows the user to enter a time in seconds and then outputs how far an object would drop if it is in freefall for that length of time. Assume no friction or resistance from air and a constant acceleration of 32 feet per second due to gravity. Use the equation:

$$\text{distance} = \frac{1}{2} \times acceleration \times time^2$$

```
// Programming Project 1.10
#include <iostream>

using namespace std;

int main()
{
      cout << "Enter a time in seconds." << endl;
      int s;
      cin >> s;

      int distance;
      distance = (32 / 2) * (s * s);

      cout << "An object in freefall for " << s <<
            " seconds will fall " << distance <<
            " feet." << endl;

      // Type a key and enter to close the program
      char c;
      cin >> c;
}
```

11. Time Conversion

Write a program that inputs an integer that represents a length of time in seconds. The program should then output the number of hours, minutes, and seconds that corresponds to that number of seconds. For example, if the user inputs 50391 total seconds then the program should output 13 hours, 59 minutes, and 51 seconds.

```
// Programming Project 1.11
#include <iostream>

using namespace std;

int main()
{
      cout << "Enter a time in seconds." << endl;
      int s;
      cin >> s;

      int hours, minutes, seconds;
      hours = s / 3600;
      minutes = (s % 3600) / 60;
      seconds = (s % 3600) % 60;

      cout << s << " total seconds is equivalent to " <<
            hours << " hours, " << minutes <<
            " minutes, and " << seconds <<
            " seconds. "<< endl;

      // Type a key and enter to close the program
      char c;
      cin >> c;
}
```

12. Ideal Body Weight Estimation

A simple rule to estimate your ideal body weight is to allow 110 pounds for the first 5 feet of height and 5 pounds for each additional inch. Write a program with a variable for the height of a person in feet and another variable for the additional inches and input values for these variables from the keyboard. Assume the person is at least 5 feet tall. For example, a person that is 6 feet and 3 inches tall would be represented with a variable that stores the number 6 and another variable that stores the number 3. Based on these values calculate and output the ideal body weight.

```
// Programming Project 1.12
#include <iostream>
using namespace std;

int main()
{
        int heightFeet = 5;
        int heightInches = 5;
        cout << "Your ideal body weight is ";
        int weight = 110 + ((heightFeet - 5)*12 + heightInches) * 5;
        cout << weight << " pounds." << endl;

        return 0;
}
```