COMSC-110 Chapter 4

Doing the math: Libraries

Valerie Colber, MBA, PMP, SCPM

Programming conventions

- Write a 4 part (1. objective; 2. requirements for INPUT, PROCESSING, OUTPUT, DATA; 3. algorithm instructions; 4. test cases) algorithm for every program, and perform the test cases showing the intermediate result of executing each instruction!
- Required comments at the beginning of your program
- Use proper indentation and alignment of { }

Programming conventions

- Always have a commented data declaration code block at the beginning of every function including the main program
- Always first output is the user introduction
- Always label all output data
- Always output an input prompt for all user input from the console

Parts of an algorithm

- What is the objective or purpose of this algorithm?
- What are the requirements or specifications for INPUT, PROCESSING (, OUTPUT, DATA in this algorithm, including any necessary formulas for processing/calculating, input values and their format, output values and their format, etc.
- What way are you going to use to accomplish the objective and describe it as algorithm instructions in English that anyone can do without any further information or support from you, each English instruction has a unique identification number for reference (10, 20, 30….), that describes a way (usually there is more than one way for doing any objective) for doing the objective that does NOT say how to write a computer program to do this objective. Each instruction can only be one instruction and produce one result.
- Test cases that demonstrates the algorithm works under any normal use of the algorithm by any user, including an example obvious test case for what the algorithm was designed for, and then think about test cases for unusual circumstances that will make your algorithm work even if the user input or actions are not what you were expecting.

Program file organization

- Programmer remarks including program objective, programmer, compiler
- Libraries
- Special compiler dependent definitions
- Global data
- Programmer defined functions (subprograms)
- Main program
- data declaration code block with programmer remarks for each variable purpose and valid values

Libraries

- Libraries are collections of useful reusable programs called functions.
- Libraries add functionality to a computer language -- they add features that are not part of the base language.
- Programmers that develop useful reusable programs, put those programs into libraries for other programmers to use in their programs.

Libraries

- Some problems are too difficult to solve with math operations only.
- cmath declares a set of functions to compute common mathematical operations and transformations
- Libraries are attached to programs through the use of **#include** statements.

Trigonometric functions

- Cos Compute cosine
- Sin Compute sine
- Tan Compute tangent
- Acos Compute arc cosine
- Asin Compute arc sine
- Atan Compute arc tangent
- Atan2 Compute arc tangent with two parameters

Hyperbolic functions

- **Cosh Compute hyperbolic cosine**
- **Sinh Compute hyperbolic sine**
- **Tanh Compute hyperbolic tangent**

Exponential and logarithmic functions

- Exp Compute exponential function
- Frexp Get significand and exponent
- Ldexp Generate number from significand and exponent
- Log Compute natural logarithm
- Log10 Compute common logarithm
- Modf Break into fractional and integral parts

Power functions

- Pow Raise to power
- Sqrt Compute square root

Rounding, absolute value and remainder functions

- **Ceil Round up value**
- **Fabs Compute absolute value**
- **Floor Round down value**
- **Fmod Compute remainder of division**

Simple amortization example

- If you deposit $100 each month for 10 years, earning 7.5% interest per year compounded monthly, how much would you have at the end of those 10 years?
- We can use the following formula where D is the monthly deposit amount of $100, S is the sum at the end of 10 years, p is the *monthly* interest rate (0.075 / 12), and T is the number of months in 10 years (10 * 12).
- We can do these calculations with simple math operations, but we would have to multiply 120 identical expressions to get $(1+ p)^T$

$$S = D\left(\frac{(1+p)^T - 1}{p}\right)$$

Simple amortization example

- To solve this, we use the built-in math library!
- It contains useful expressions, like raise to a power, find the square root, trig, and exponentials.
- For example, to raise "x" to the power of "y", use this expression: **pow(x, y)** , but "x" and "y" must be floating point variables.

Functions

- Programmers refer to **pow(x, y)** as "a function call".
- "pow" is the function name.
- Function names follow the same naming rules as variables, which involved the concept of "identifiers".
- Since both variables and functions are identified by "identifiers", it makes it difficult to tell whether an identifier refers to a variable or a function.
- But there is an easy way to tell the difference -- functions always have parentheses, even if the parentheses are "empty".

mort.cpp

```
//Course:COMSC-110-8269
//Author:Teacher
//Compiler: MS VS 2010
//Editor: Notepad

//Libraries
#include <cmath>

int main()
{
// assignment upon declaration
double D = 100.0;
double p = 0.075 / 12.0;
double T = 10.0 * 12.0;

// the pow function (new)
double S = D * ((pow(1.0 + p, T) – 1.0) / p);
}
```

why is using namespace std; not used?

Is this the only way to write this program?

Programmer Comments

- Note the use of ***//*** in the mort.cpp example.
- These denote "comments“ or remarks -- programmer's notes that are included in code but are ignored by the compiler.

How can we use remarks in a program?

Integer division and truncation

- There is an inherent inconsistency when using division and integers, because integers are whole numbers and cannot have fractions.
- To solve this, C++ simply throws away, or *truncates* any fractional value.
- It does not round to the nearest -- it truncates.
- So **99 / 100** resolves to zero!

Integer division and truncation

- When using numbers in calculations, whole numbers without decimals are treated as integers, and numbers with decimals are treated as floating point numbers!

Conversion of temperature from Celsius to Fahrenheit

- Formula is f = (9 / 5) * c + 32
- If we were to program this as

f = (9 / 5) * c + 32;

we would have a problem because **9 / 5** resolves to one!

- But if either of the 2 numbers in an operation is a floating point number, then the operation is resolved as if both were floating point.

Conversion of temperature from Celsius to Fahrenheit

- So one solution is to write the statement this way: **f = (9.0 / 5) * c + 32;**
- And it really does not matter how "c" is declared -- integer or floating point -- because it gets multiplied by the floating point result, 1.8 (that is, 9.0 / 5), and is thus temporarily "promoted" to floating point for purposes of the calculation.

Converting integers to floating point numbers

- Most of the expressions in the library require numbers of data type "double".
- But for various reasons, we may have numbers of data type "int" that we want to use with one of these expressions.
- So for an integer number or expression, use **sqrt(double(x))** if "x" is an integer.
- This uses *type casting*, which is explained in chapter 9.

Formatting output

- Computer calculations are accurate.
- They are so accurate that computers just have to show all the digits behind the decimal.
- Consider 10 / 3 -- the result is 3.33333....
- The computer likes to show all the 3's that it has.
- But you can force the computer to round off numbers and show them with a desired number of digits after the decimal.

Formatting output

- Use the include statement **#include <iomanip>** , to attach the "iomanip" expansion library.
- Then use the statement **cout << fixed;** before the first rounded-off value appears in a "cout" statement

OR

cout.setf(ios::fixed|ios::showpoint);

- Do this just once -- once it appears in a "cout" statement, it is a permanent change for the rest of the program.
- Finally, use the statement **cout << setprecision(2);** so that all values printed *after* "setprecision" will be rounded to 2 decimal digits.
- You can change this at any time by using another "setprecision" with another digit in the parentheses.
- Note that formatting applies only to the *appearance* of a number when it appears on the display screen.
- Formatting does *not* affect its actual value as stored in the program's memory.

mort2.cpp

```
//Course: COMSC-110-5003
//Author: Teacher
//Compiler: TDM MinGW
//Editor: Notepad

//Libraries
#include <iomanip>  //formatting
#include <iostream> //console output
using namespace std;
#include <cmath>  //pow function

int main()
{
  //data
  double years = 10.0; //number of years
  double D = 100.0;  //deposit amount
  double p = 0.075 / 12.0;  //monthly interest rate
  double T = years * 12.0; //number of months
  double S = D * ((pow(1.0 + p, T) – 1.0) / p); //amortization formula

  // formatting output (see chapter 4 section 4.2)
  cout.setf(ios::fixed|ios::showpoint);
  cout << setprecision(0);

  cout << "In " << years << " years, $";
  cout << D << " deposited per month will grow to $";
  cout << setprecision(2) << S << "." << endl;
} //main
```

Copy, compile, and run this example. The result should be **In 10 years, $100 deposited per month will grow to $17793.03.**

```
//example program for chapter 4
//programmer: teacher
//editor: notepad
//compiler: MS Visual Studio 2010
//objective: calculate the sum of money from a deposit over time

#include <iomanip>
#include <iostream>
using namespace std;
#include <cmath>

int main()
{
  //variables
  double years = 10.0; //number of years
  double D = 100.0; //amount of deposit
  double p = 0.075 / 12.0; //monthly interest rate
  double T = years * 12.0; //number of months
  double S = D * ((pow(1.0 + p, T) – 1.0) / p); //sum of money over time

// formatting output
  cout << fixed;
  cout << setprecision(0);

  //output
  cout << “This program will calculate the sum of money from a deposit over time”
<< endl << endl;
  cout << "In " << years << " years, $";
  cout << D << " deposited per month will grow to $";
  cout << setprecision(2) << S << "." << endl;
}
```

Copy, compile, and run this example. The result should be **In 10 years, $100 deposited per month will grow to $17793.03.**

One million and counting

- If you print the value of a variable whose data type is "double", and do not apply any formatting, you will have no problem -- unless it's value is 1 million or more!
- To fix this problem, use **setprecision(15)** before sending the value to **cout** , *without* using **fixed**.
- *If you use **fixed** before **setprecision(15)** , then you will get 15 decimal digits!*

Turning off formatting

- Once you send **fixed** to **cout** , you are committed to formatting.
- All the rest of the numeric values that you send will be affected.
- But there is a way to turn off formatting, and go back to the way things were when your program first started.

unformatting.cpp

```
//example program for chapter 4
//programmer: teacher
//editor: notepad
//compiler: TDM MinGW
//objective: calculate the sum of money from a deposit over time

//libraries
#include <iomanip> //formatting
#include <iostream> //console output
using namespace std;

// system specific logic for use in unformatting
#ifdef __GNUC__
const std::_Ios_Fmtflags UNFORMAT = cout.flags();
#elif defined _WIN32
const int UNFORMAT = cout.flags();
#endif

int main()
{
  //data
  double a = 54.214; //actual initial value in memory

  // set to 2 digits after the decimal (see 4.2)
  cout << fixed << setprecision(2); // applies until set to something else

  // print with current formatting (see 3.4.1)
  cout << "formatted a = " << a << endl; // 54.21 (rounded off!)

  // unformat, so show a number "as is" (new)
  cout << setprecision(6); // resets precision to its default
  cout.flags(UNFORMAT); // turns off "fixed"

  // print unformatted, with a label (see 3.4.1)
  cout << "unformatted a = " << a << endl; // 54.214 (original)
}
```

- Note that this code includes some special instructions to determine if the compiler is Windows or GNU.

- The required code is different, depending on which compiler is being used.

- So our example will work, as is, in either compiler version!

- Note the alternate use of ***//***, where it appears on the same line as a C++ statement.

- The statement is read by the compiler as it normally would be, but the compiler ignores the ***//*** and everything that follows it on the same line.

Code blocks

- Usually it takes more that a single statement to do one thing in a program.
- Similarly in written language it can take several sentences to complete a thought.
- In written English we use paragraphs to group related sentences together to form a thought.
- In programming we use "code blocks".

Code blocks

- Code blocks are written in programs purely for organizational purposes.
- Their use, correct or otherwise, has no impact on a program's operation.
- Code blocks usually contain a comment line that explains the purpose of the code block, so that a reader of the program does not have to read all the statements in the code block in order to figure out what it does.
- These leading code block comments come from the processing section in your algorithm.
- To make code blocks easier to spot in a long program, they are usually separated by one blank line.

More handy libraries

Function	Explanation	Required #include statement
`toupper(x)`	uppercase char version of x, where x is a char	`#include <cctype>`
`tolower(x)`	lowercase char version of x, where x is a char	`#include <cctype>`
`abs(x)`	absolute value of x, where x is an integer	`#include <cmath>`
`fabs(x)`	absolute value of x, where x is a floating point number	`#include <cmath>`

See Appendix The C++ Library

- Page 235 in the workbook
- Shows each function in alphabetical order that we will be using in this course. There may be many more, but we won't need them.
- Shows the library they are in and whether they need the statement

  using namespace std;

- C++ libraries are above this statement and C libraries are directly below this statement in the libraries code block.

Expressions

- Note that the function calls in the previous table are all *expressions*.
- They resolve to *new* values -- they do not modify the original value stored in the variable **x**.
- In order to change a stored value, you need to use an expression in an assignment statement.

Conversion statements

Statement	Explanation
x = toupper(x);	replaces the char stored in x with its uppercase version; no effect if x's value is not a letter
x = tolower(x);	replaces the char stored in x with its lowercase version; no effect if x's value is not a letter
x = abs(x);	replaces the int stored in x with its absolute value; no effect if x's value already non-negative
x = fabs(x);	replaces the double stored in x with its absolute value; no effect if x's value already non-negative

Notes on conversion statements

- Note that **toupper(x)** and **tolower(x)** work only for **char** values and variables.
- They do *not* work for **string** values or variables.

Using multiple #include statements

- Separate the libraries beginning with the letter "c" from the others -- these are the libraries that C++ inherited from the C language, and they do not require the **using namespace std;** statement.
- Put all of the C++ libraries, like **iostream**, *above* the **using namespace std;** statement, in alphabetical order.
- Put all of the C libraries, like **cmath**, *below* the **using namespace std;** statement, also in alphabetical order.
- If there are no C++ libraries at all, then exclude the **using namespace std;** statement -- some compilers do not allow the use of that statement when no C++ libraries are included.
- Only use libraries as necessary!

References

- INTRODUCTION TO PROGRAMMING USING C++ by Robert Burns