

BA272 Manipulating Data

Manipulating Data (p. 32-44)

- A **Variable** is a named location where you can store something (p.32).
 - You can think of them as boxes. Each has a particular size and shape and has a name painted on it. It is important to use a box of the right size and shape and give it a good name. Otherwise, you will have a cluttered house, you will waste lots of space, your stuff will get broken, and you will find yourself handling things that that don't work.
- Three kinds of variables:
 - **Numeric variables** are differentiated by the size number they can hold and their precision (how decimal values are stored)
 - Although we may get along just fine for BA272 using only **int** and **double**, you need to understand a variety of numeric variable sizes.
 - It is good to use **integer** variables when we can (see the list on p. 33)
 - They take up less memory
 - They are precise – 1 is always 1, never .999998
 - The sizes are a power of two thing 2^8 is 256. So if we allow negatives (**sbyte**) we can store -128 through 127 - that's 256 values because 0 counts too. So it takes 8 bits (each bit a one or zero) to store an sbyte. A **byte** variable uses the same 8 bits but can't be negative so it ranges from 0 to 255. We say that there are 8 bits in a byte in general, and in C# we call an integer that takes up 8 bits a variable of type byte.
 - Similarly **short** and **ushort** use up two bytes (16 bits) and can have 2^{16} , or 65,536 different values. **int** and **uint** are bigger (in space used and range of value stored), and **long** and **ulong** are longer still.
 - I'm not quite sure why Rob Miles added **char** to the integer list. But it does make some sense. Each char value represents a single unicode character. Because it takes up 16 bits, we can identify up to 65,536 different values in Unicode. For a long time almost everything was stored in ASCII which uses only one byte to store a letter. That allowed us 256 ASCII characters: more than enough for English even with both lower and upper case letters plus numbers. But using Unicode we can store languages like Chinese which has thousands of individual "letters."
 - Storing numbers with decimal places is harder because we don't know when we have enough decimal places. Consider, how would you store the value of the fraction $2/3$? That is one good way to think about storing real values. We will just use **doubles** in BA272. They work very nicely for many situations.
 - Choosing the right variable type can be really important when you do dollar amounts, since you will have to deal with rounding issues. Rob's idea of doing it in cents instead of dollars is a great solution.
 - Here are some rules of thumb to help you build reliable and maintainable systems despite this sort of problem.
 - Truncation and rounding errors should be explicitly avoided. Clever solutions that manage to get the correct answer because of situational subtleties make it difficult to debug and maintain code. Sometimes an implicit truncation, e.g., `int i = intCountOfSheep / 2`, is the right

thing to do. But a programmer who comes through later may not know if it is an error or a feature. So provide comments.

- Sometimes adding extra variables, extra casts, extra parentheses, extra computations, and certainly extra comments can make a block of code easier to understand and therefore easier to maintain. On the other hand extra code (for variables or computations) and extra processing can slow a system down, make it use more resources, or make things more confusing. There is a balance here but many programmers worry way too much about avoiding an extra variable or extra line of code and cause headaches for the organizations that need to use the system over time. Strive for clarity.
 - Be sure and read the comments in the text on widening, narrowing and casting. These concepts are explored a bit more below. *Hint: I will try to identify students who master the material by dealing with these issues on exams!*
- **Text**
 - A value that is provided in the code like 'Q' is called a **literal**. This is true for both numbers (in `l = 25`) and text.
 - Understand **character codes** and **escape sequences**.
 - Practice writing tabs and new line characters. And use the **verbatim character @**.
- **Other stuff** – learning about numbers and text variables sets the stage for lots of other kinds of things we will use in our programs.
 - **bool** – a variable that can be only true or false.
 - **Identifiers** - know and apply the naming rules as per pg. 38
 - **Remember: = is a gozzinta instruction, not a question**
 - **Expressions: Operators + Operands** (p. 39-40)
- **Casting, Widening, Narrowing**
 - Types of data in expressions (p. 42). Do not rely on chance. Cast things when you mean to, don't get caught out by something like the difference between `d = 1/2` and `d = 1/2.0` !
 - Take note that the plus sign, when used with strings, concatenates the strings.

This material wraps up what you need to know to complete the first assignment. Good luck!

Explore numeric type errors and casting:

- Given that mathematically $50 / 2 = (25 + 25) / 2 = (25/2) + (25/2) = 25/2 + 25/2 = 25$. Consider the following 6 statements.

```
Line A: int i = (int)((25f / 2f) + (25 / 2));
Line B: int i = (25 + 25) / 2;
Line C: int i = (25f / 2f) + (25f / 2f);
Line D: int i = (int)((25f / 2f) + (25f / 2f));
Line E: int i = (25 / 2) + (25 / 2);
Line F: int i = (int)(25f / 2f + 25f / 2f);
Line G: double d = (25 / 2) + (25 / 2);
```

1. One is an error that would be rejected by the compiler. Which one? What is the error?
2. Only three of the computations return the correct answer. Which ones?
3. Explain what the code `(int)` does. *Hint: include a word you might hear on a fishing trip.*
4. Explain how 25 works differently from 25f.
5. Explain what is wrong with each of the ones that do not work correctly.

6. Identify an example of widening in the statements. Explain why it is widening.
7. Identify an example of narrowing in the statements. Explain why it is narrowing.
8. What are the implications of narrowing as opposed to widening?
9. Some might say that all the commands (except the one that has a syntax error) “work.” The computer is just doing what we told it to do. And actually, business rules sometimes require numbers to be rounded or truncated in different (even weird!) ways. Write a few sentences to describe how a programmer can approach these sorts of issues given the goal of creating reliable, maintainable systems.

Compare these two blocks of code.

- Block one:

```
int iFeetPerMile = 5280; // Everyone knows that!
double dAvgSpeedFeetPerHour = 2427; // Computed avg for the job
double dHoursSpent = 22.6; // Billable time
int iRatePerMileInCents = 5435; // Negotiated rate or $54.35 / mile

double dFeetDone = dAvgSpeedFeetPerHour * dHoursSpent;
int iMilesDone = (int)( dFeetDone / iFeetPerMile); // only complete miles
int iAmountPaidInCents = iMilesDone * iRatePerMileInCents;
double dAmountPaidInDollars = (double) iAmountPaidInCents / 100;
Console.WriteLine("Charge: " + dAmountPaidInDollars);
```

- Block two:

```
double avgSpeedFeetPerHour = 2427; // Computed avg for the job
double hoursSpent = 22.6; // Billable time
int ratePerMileInCents = 5435; // Negotiated rate or $54.35 / mile
Console.WriteLine("Charge: " + (double)((double)(
    (int)((avgSpeedFeetPerHour * hoursSpent) / 5280)
    ) * ratePerMileInCents) / 100));
```

1. Do they accomplish the same thing? (Okay, so that would take a while to figure out by inspection and you still might not be sure, so make a program and test the two.)
2. Which is easier to read?
3. Discuss why one approach might be better or worse than the other.