**Chapter 12: Fundamental Data Types**

**Numbers in General**

* Avoid “magic numbers” (literal numbers)
  + Changes can be made more reliably
  + Changes can be made more easily
  + Code is more readable
* CAN use 0, 1 no problem
* Anticipate divide by 0 errors
* Make type conversions obvious
* Avoid mixed type comparison

**Integers**

* Check for integer division 🡪 nearest integer
* Check for integer overflow
  + Usually comes about from multiplication

Floating-Point Numbers

* Avoid additions and subtractions on numbers that have greatly different magnitudes
  + 1,000,000.00 + .01 wont encompass the last part
* Avoid equality comparisons
  + Floats that should be equal are not always equal
* Rounding problems
  + Change to a variable type that has greater precision
  + Change to binary coded decimal variables
  + Change from floating point to integer variables
    - Similar to BCD

**Characters and Strings**

* Avoid magic characters and strings
  + - ‘A’ or “Gigamatic Accounting Program”
    - Literal values like magic numbers
  + Set as variables
  + International markets are important so translating a single variable is better than an entire program
  + String literals take up space
  + String and character literals are cryptic

**Boolean Variables**

* Use Boolean variables to document your program
  + Make the implication of the test unmistakable

**Named Constants**

* Like a variable but one that cant be changed once assigned
* This is a way of “parameterizing” a program
  + Putting an aspect of the program that might change into a parameter that can be changed in one place instead of having to make changes all over the system
* Use named constants in data declarations
* Be sure to use consistently too

**Arrays**

* Try to think of arrays as sequential structures
  + Some smart folk have suggested that arrays never be accessed randomly, only sequentially
  + Sets, stacks and queues are better
  + Testing found that designs created this way resulted in
    - Fewer variables
    - Fewer variable references
    - Efficient designs
    - Highly reliable software
* Consider using sets, stacks queues as alternatives before an array
* If multidimensional array
  + Make sure indexes are used in the correct order
  + Watch out for index cross-talk in nested loops

**Creating Your Own Types (Type Aliasing)**

* Good for clarifying the understanding of a program
* Protect program against unforeseen changes and make it easier to read
* Why to create
  + Make modifications easier
  + Avoid excessive information distribution
  + Increase reliability
  + Make up for language weakness

Guidelines for Creating Your Own Types

* Create types with functionally oriented names
  + Avoid names that refer to underlying data types
* Be wary of name types that refer to predefined types
* Avoid redefined types
* Don’t use a predefined type
  + Use your own types as much as possible
* Define substitute types for portability
* Be sure not to define types that are easily mistaken for predefined types
* Consider creating a class rather than using a typedef