Keys to GOOD Programming

Main Programming Challenge: "**Manage Complexity**"

Modularity

- "Divide and Conquer"
- Design and implement programs in small (independent) segments
    - Easier to **create** the code
      → Collection of "smaller" modules
    - Easier to **debug** the code
      → Isolates ERRORS
    - Easier to **read** the code
    - Easier to **modify** the code
    - Helps eliminate **redundant** code

Modular Design Alternatives

Key Concept: Abstraction and Information Hiding

- Separate "WHAT" from "HOW"

- **Abstraction** separates the **purpose** of a module from its **implementation**
    - Information Hiding: **Public** vs. **Private** information

- **Functional abstraction** (procedural)
    - Purpose
    - Parameters

- **Data Abstraction**
    - Collection of Data
    - Set of operations on that Data
    - Examples:
        - Database
        - Floating point numbers
        - List
    - Implementation: using "data structures" in the programming language

- **ADT: Abstract Data Type**
    - **Set of Values (Collection of Data)**
    - **Set of Operations defined on these values**
    - **Only the "defined" operations are allowed or possible**

Modular Design Approaches

Top-Down Design (TDD)

- Focus on "**verbs**" in the problem statement
- Emphasis of the problem is on **algorithms** (Functional Abstraction)
- Frequently uses a Structure chart or Flowchart
- Stepwise Refinement
  - Start with "high level" overview of the main task(s) to be accomplished
  - Examine each step and partition it into smaller, more focused modules
  - Continue through successively lower levels of detail until possible to code directly

Object-Oriented Design (OOD)

- Focus on "**nouns**" in the problem statement
- Emphasis of the problem is on "**OBJECTS**", that combine data and operations
- General concept of "objects" can embody most aspects of a program
  - Traditional "data" items
  - "Report" object
  - "Input" object
  - "Compute the answers" object

- **Class**: A "type" for a group of similar objects
- **Instance**: A particular object of some Class type
- **Method**: An operation for a Class type

- Overall program solution creates various **instances** of objects and invokes their methods to achieve the desired results.
- Three Principles of Object-Oriented Programming
  - **Encapsulation**: Objects combine data and operations
  - **Inheritance**: Classes can inherit properties from other classes
  - **Polymorphism**: Objects can determine appropriate operations at execution time
    Operator Overloading

Typical Program Design combines OOD, TDD, abstraction, and information hiding

Keys to GOOD Programming

Modularity

Modifiability

Create code that is "easy" to

- correct
- adapt to new system environment (different machine, O/S, etc)
- adapt to new external environment (e.g., change in tax laws)
- enhance with new features

- Use **Named Constants**

```
const   float  TAX_RATE = 0.086;
```

- Use **Functions**, parameters, and local variables

- Use **Object-Oriented Encapsulation** and other Modularity techniques

Ease of Use

Don't forget the USER !!!

- Interactive input?
  - Prompt the user for all input
  - Clear, precise
  - Echo all input so that user can verify

- Label all output neatly and clearly
  - Easy to read
  - Avoid misinterpretation
  - Neatness Counts!!

Fail-Safe Programming

A fail-safe program is one that will perform reasonably no matter how anyone uses it

- Validate input data. Detect errors in input values.
  - Problem specification should be precise in describing acceptable inputs
    "Reasonable" error checking depends on the program and users
  - Use "built-in" I/O-conversions cautiously

- Anticipate computational or data manipulation errors
  - Failed calculations (e.g., divide by zero)
  - Overflow/Underflow of a list
  - System failures (e.g. failure to open a file)

Debugging

- Learn to use the Microsoft Visual C/C++ DEBUGGER
  - Breakpoints
  - Single-step
    - Step-In Step-Over Step-Out
  - Examining values of variables
  - Watch Window
  - Call Stack

- Insert cout << debugging code

- Write "debugging" functions that output useful debugging data
  - Makes it easy to call from various locations in your code

- Debug each function individually
  - Verify function arguments at the beginning of a function
  - Verify function return-values at end of function or in calling routine

- Debug loops carefully
  - Particularly validate the first and last times through a loop
  - Be careful with **boundary conditions**

Style Counts

Refer to "Programming Assignments and Style" handout for guidelines

Summary:

- Use functions

- **Avoid** the use of **global variables** (unless your instructor says otherwise!) Instead, use **local variables** and pass all necessary **parameters to functions** using parameter lists.

- Know when to use **value** arguments and **reference** arguments in functions

- Know when to use **void** functions and when to use **valued** functions

- Always use **meaningful identifier names**
  SearchAndReplace(…) or search_and_replace(…)

- Always use **indentation** to make code more readable

- **Document** your code