CS1010E Programming Methodology

Joxan Jaffar

Block COM1, Room 3-11, +65 65167346
www.comp.nus.edu.sg/~joxan
cs1010e@comp.nus.edu.sg

Semester II, 2016/2017

Lecture 1:

Engineering Problem Solving, An Overview

Lecture Outline

- Engineering in the 21st Century
- Computing Systems:
  Hardware and *Software*
- An Engineering Problem-Solving *Methodology*
  A five-step problem-solving technique that we will use throughout this course

Engineering in the 21^{st} Century

- Engineers solve real-world problems using scientific principles from many disciplines
- These disciplines include computer science, mathematics, physics, biology, and chemistry
- Large variety of subjects and challenge of real problems makes engineering so interesting and rewarding

Engineering Achievements

- Since the development of the computer in the 1950s, many significant engineering achievements have occurred
- In 1989, the National Academy of Engineering selected the 10 engineering achievements from the previous 25 years
- They illustrate the multidisciplinary nature of engineering and how engineering has improved our lives.

Engineering Achievements (Older)

- Microprocessor
- Moon landing
- Application satellites
- Computer-aided design and manufacturing
- Jumbo jet
- Advanced composite materials
- Computerized axial tomography
- Genetic engineering
- Lasers
- Optical fiber

Other Achievements (post 1990)

- The Internet
- Mobile and Embedded computing
- Games and Movies
- Social Media and Networking
- Cloud computing
- Artifical Intelligence

Computing is now ubiquitous!

Why Should Engineers study Programming?

Computational Thinking:

- Logically organizing and analyzing data
- Representing data through abstractions such as models and simulations
- Automating solutions through algorithmic thinking (a series of ordered steps)
- Identifying, analyzing, and implementing possible solutions to achieve the most effective combination of steps and resources
- Generalizing and transferring this problem solving process to a wide variety of problems

Jeannette M. Wing, Computational Thinking, CACM Viewpoint, March 2006
`http://www.cs.cmu.edu/~wing/`

Infomation Technology and Computational Thinking

We are now very familiar with I.T.:

email, Networks, MPG, LAN, modem, blogs, WYSIWYG, desktop publishing, File, icon, chip, RFID, USB, DOS, RAM, GUI, etc ...

But C.T. is *beyond* I.T.

- Knowing about data and ideas and using/combining these resources to solve problems.
- Go beyond using tools and information to creating tools and information
- These raw materials require thought processes about manipulating data, using abstractions, and lots of Computer Science concepts

Computational Thinking Concepts

- Algorithm: the most important thing
- Data: variables, data bases, Queue
- Abstraction: conceptualizing, modularizing
- Query: search, conditionals, Boolean
- Sensing and Feedback: robotics
- Iterations: loops, recursion

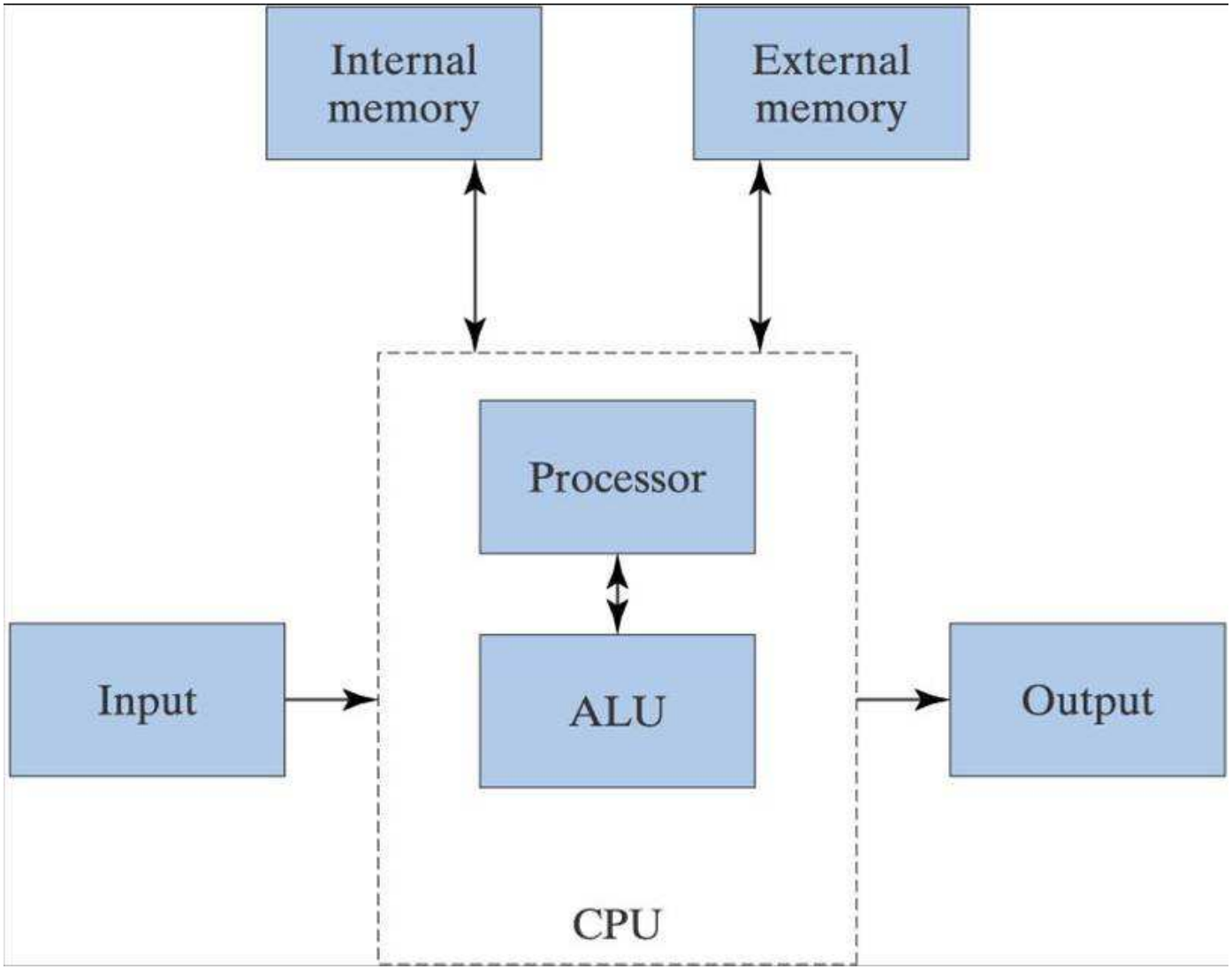
Why Should Engineers study Programming in C?

- Most widely used
- Closest to the computer's memory model
- Covers fundamental features that support understanding toward other programming languages

Computer Hardware and Software

- A computer is a machine that is designed to perform operations that are specified with a set of instructions called a program
- Computer hardware refers to computer equipment
- Computer software refers to the programs that describe the steps that we want the computer to perform.

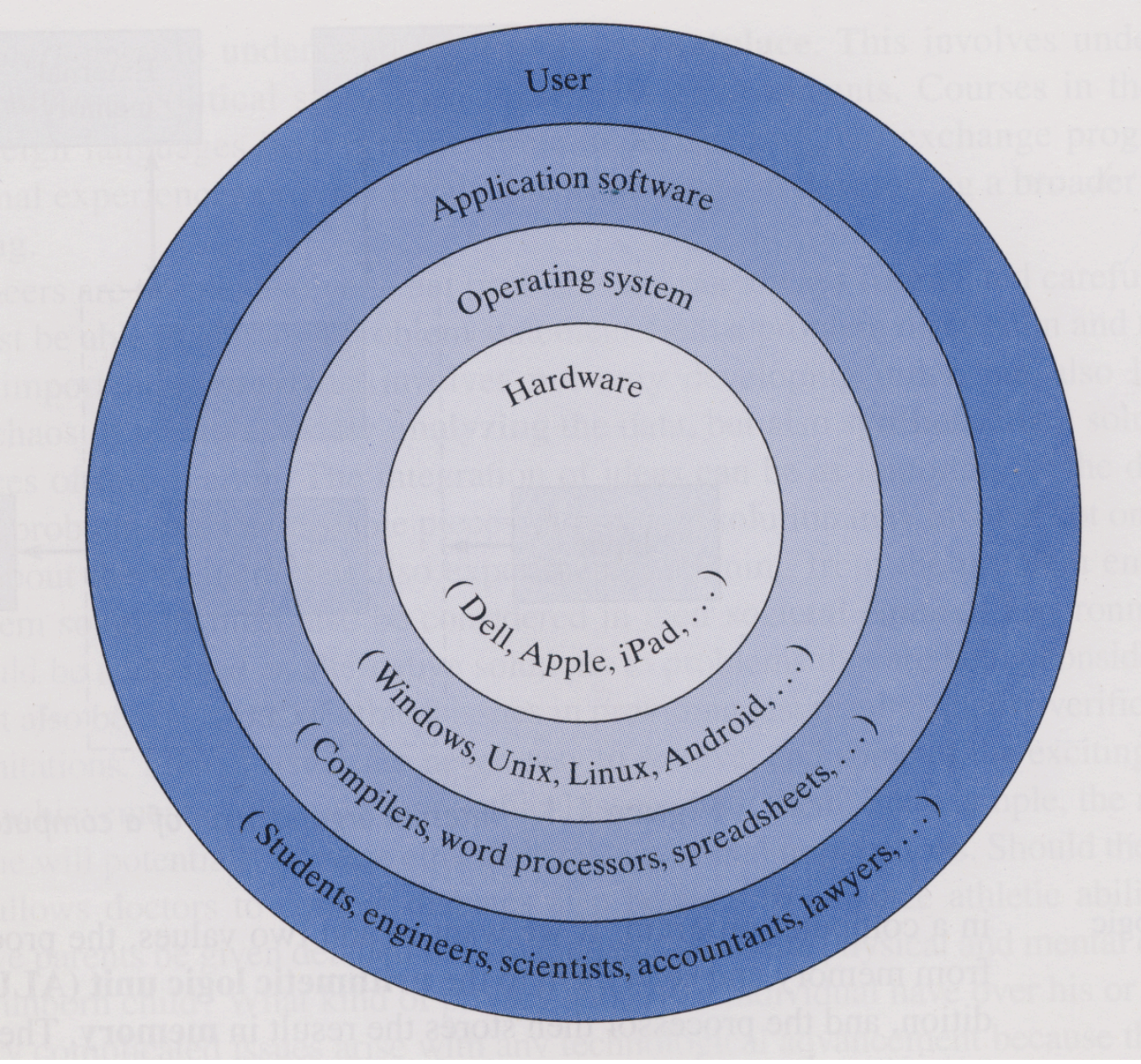
Internal Organization



More Computer Hardware

- Input and Output devices
- Storage
- Personal computers
- Workstations
- Supercomputers
- Networks

Software Interface



Computer Software

- Operating systems
  - User interface (windows, shells, ...)
  - Networking
  - Programming support and Utilities
  - In general, an OS performs resource management of a computer
- Software Tools
  - Word processors
  - Desktop publishing
  - Spreadsheet
  - Database management
  - Computer-aided design
  - Mathematical computation tools

Computer Languages

- **Machine language** (bits: 0's and 1's)
- **Assembly language** (mnemonics, jumps)
- **High-level language** (C, Java, ...)
- **Natural language**

High-Level Languages

- 1950s/60s - At the Beginning
  FORTRAN, LISP, COBOL
  APL, Simula, BASIC, PL/I, BCPL
- 1970s: Structured and Declarative Programming
  C, Smalltalk, PROLOG, ML, B, Scheme
- 1980s: Modules
  C++, Ada, Erlang, Perl
- 1990s - the Internet age:
  Haskell, Python, Visual Basic, Java, Javascript, PHP
- 2000s - the Latest
  C#, Scala, Visual Basic .NET, F#

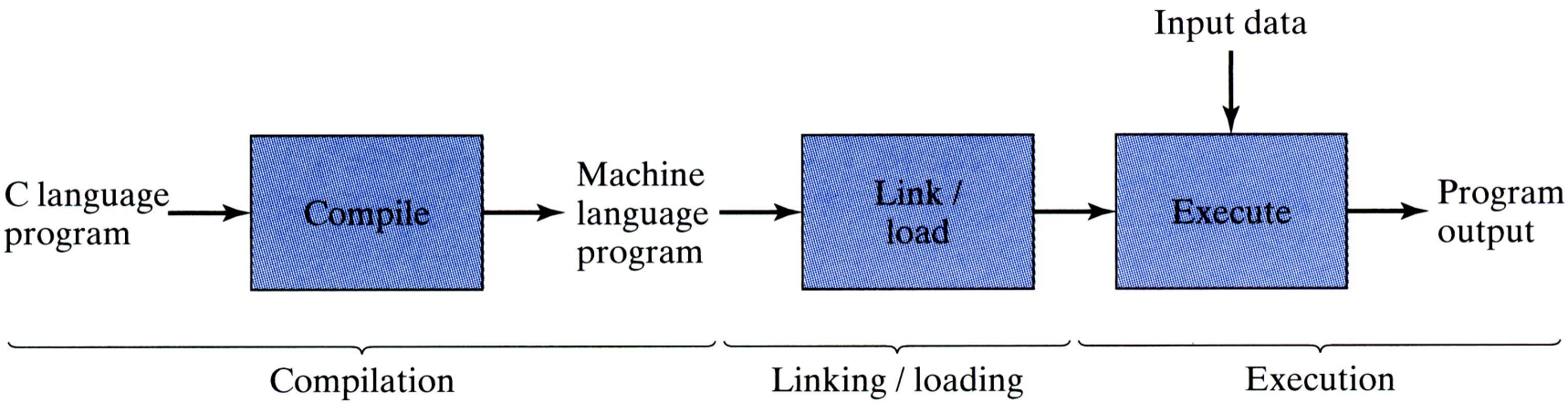
More about C

- General-purpose language
- Evolved from two languages, BCPL and B
- Designed by Dennis Ritchie of Bell Laboratories and implemented on a PDP-11 in 1972
- Used as the systems language for the UNIX operating system
- Standard definition needed: ANSI C standard approved in 1989
- Standard text: B. Kernighan and D. Ritchie, *The C Programming Language*, 2nd Ed, Prentice-Hall, 1988.

Executing a Computer Program

- A program written in a high-level language such as C must be translated into machine language before the instructions can be executed by the computer
- A special program called a **compiler** is used to perform this translation
- The C compiler we will use is called **gcc** (GNU Compiler for C)
- The **gcc** command does the compilation.

From Compiling to Executing



Compiling

- If errors (often called **bugs**) are detected by the compiler, corresponding error messages are printed
- Program statements must be corrected and we must compile our program again
- Errors detected during this stage are called **compiler errors** or **compile-time errors**
- Process of compiling, correcting statements, and recompiling is often repeated several times before all compiler errors are removed

Compiling

- When there are no more errors, the compiler generates the equivalent program in machine language
- The original C program is referred to as the **source program**
- The machine language version is called an **object program**
- The source program and the object program specify the same steps, but the source program is specified in a high-level language and the object program is specified in machine language

Linking and Loading

- Once the program has compiled correctly, we can **execute** the object program. This requires two preparation steps
- This preparation involves **linking** other machine language statements to the object program and then **loading** the entire program into memory
- After this linking / loading, the program steps are executed by the computer

Executing

- Errors can occur during the execution of a program.
  - An error that causes the program to stop executing abnormally (halfway) is called a **run-time error**
    Example: trying to divide a number by zero
  - An error that results in incorrect results to be computed is called a **logic error**

- These errors must be fixed by modifying the source program, compiling, linking / loading, and executing the executable program again

Software Life Cycle

- In 1955, the cost of a computer solution was estimated to be 15% for software development and 85% for computer hardware
- In 1985 these numbers had switched
- Nowadays, the majority of the cost of a computer solution is invested in software development
- The development of a software project follows definite steps or cycles called a **software life cycle**

Software Life Cycle Phases

Life Cycle	Percent of Effort
Definition	3
Specification	15
Coding and modular testing	14
Integrated testing	8
Maintainence	60

Software Life Cycle

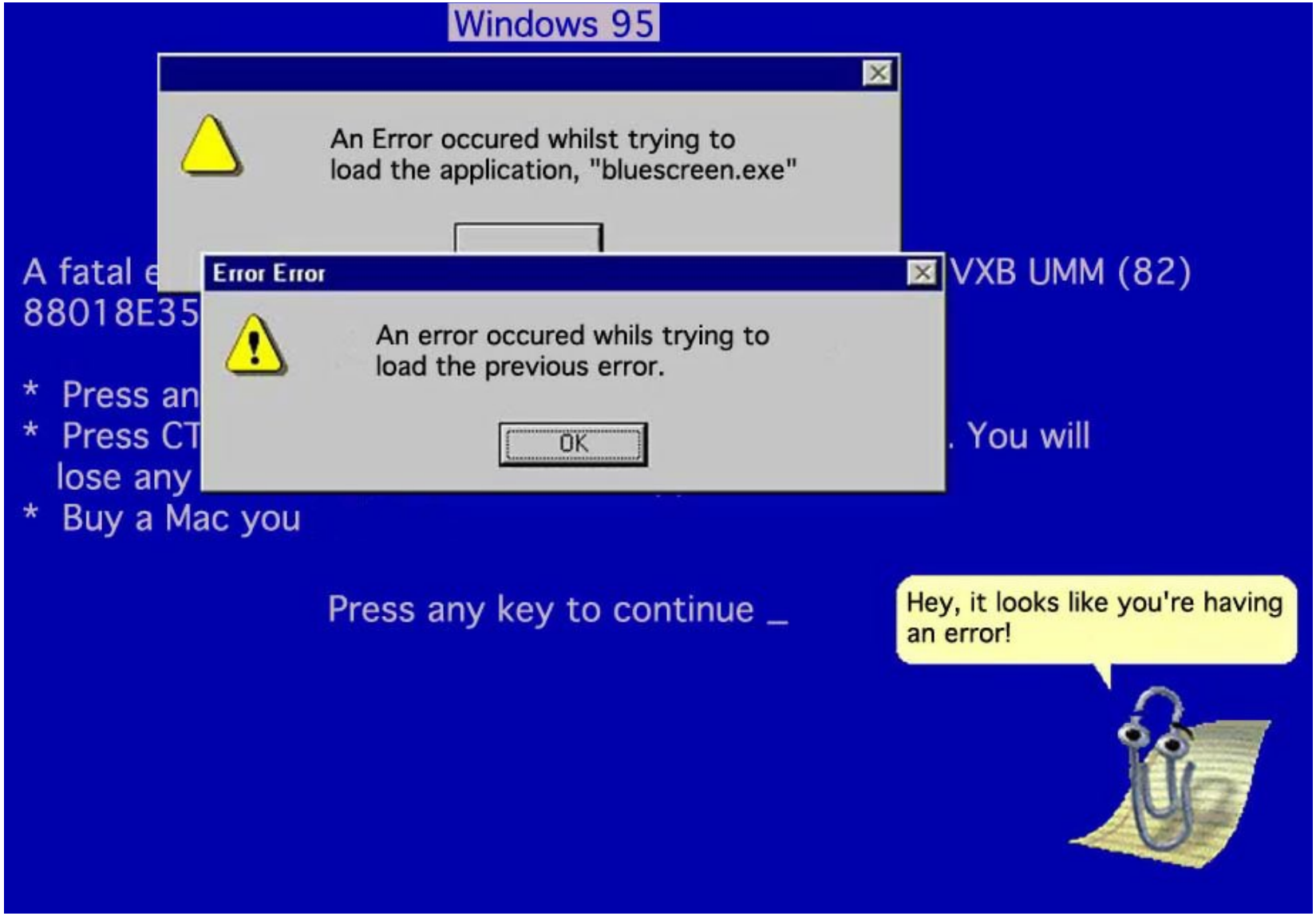
- **Software maintenance** is a significant part of the total cost
- Maintenance includes adding enhancements to the software, fixing errors, adapting the software to work with new hardware and software
- The ease of providing maintenance is directly related to the original definition and specification
- We must define and specify the solution carefully before beginning to code or test it

Software Life Cycle

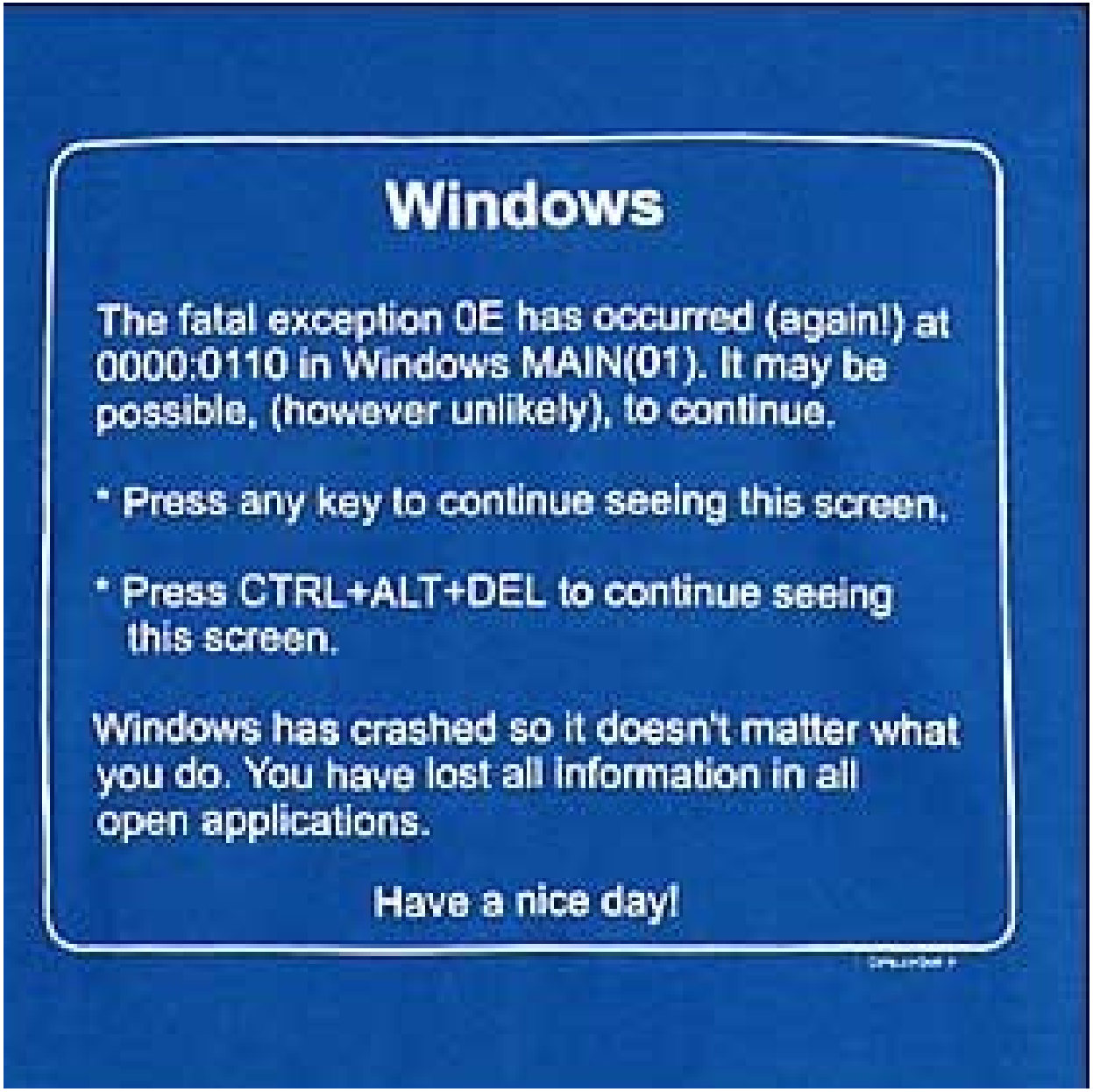
- As an engineer, it is likely that you need to modify or add additional capabilities to existing software
- Such modifications are much simpler if the existing software is well-structured, readable, and the documentation is up-to-date and clearly written
- We stress good habits that make programs more readable and self-documenting
- Following **style guidelines** will make your programming easier!

Seen This Before?

Friendly Version



What it Really Means



BUGS are a REAL Problem

- **Large applications** (global trading, ...)
- **Critical applications** (weapons control, ...)
- **Real-time performance** (car brakes, ...)
- **Security**

Some Famous Bugs

- July 28, 1962: Mariner I space probe (rocket lost)
- 1982: Soviet gas pipeline (boom! ... sabotage?)
- 1985-1987: Therac-25 medical accelerator ($\geq$ 5 people dead)
- 1988: Buffer overflow in Berkeley Unix finger daemon (first “worm”)
- 1988-1996: Kerberos Random Number Generator (allows intrusion)
- January 15, 1990
  AT&T Network Outage (60K people affected for 9 hours)
- 1993: Intel Pentium floating point divide (cost Intel $475m)
- 1995/1996: The Ping of Death (malicious crash)
- June 4, 1996: Ariane 5 Flight 501 (boom)
- November 2000
  National Cancer Institute, Panama City ($\geq$ 8 people die)

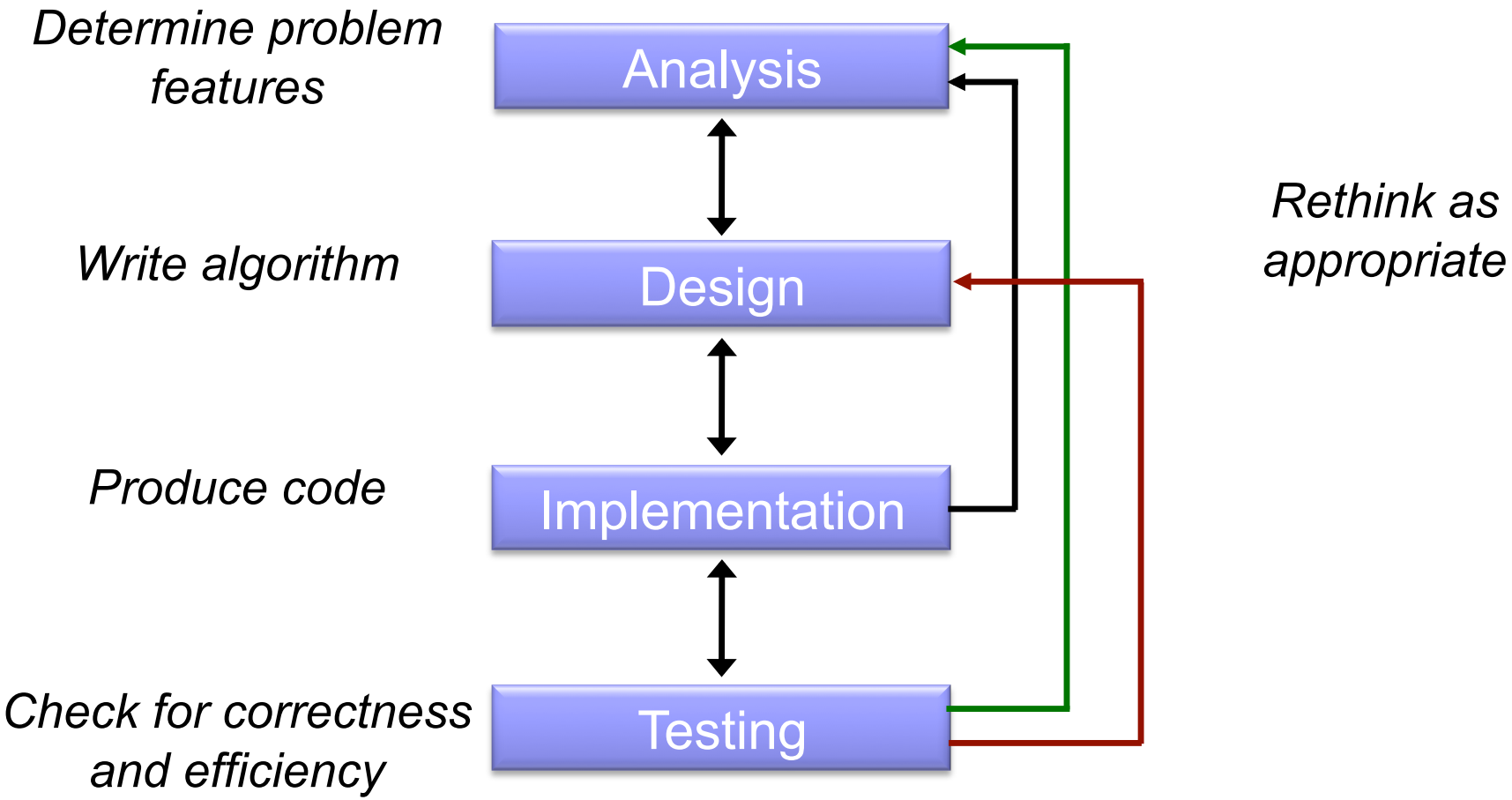
Some Famous Bugs (2)

- April 2008
  Valgrind, a maintainer of Debian patched OpenSSL and broke the random number generator in the process. Every key generated with the broken version is compromised threatening many applications that rely on encryption such as S/MIME, Tor, SSL or TLS protected connections and SSH.

- August 14, 2008
  Power blackout in Northern America affecting 55 million people. Cause: a “race condition” where two parts of the system were competing over the same resource and were unable to resolve the conflict.

- April 2014
  Heartbleed, an OpenSSL vulnerability. Shut down the Canada Revenue Agency’s public online access to online filing following the theft of social insurance numbers. As of May 20, 2014, 1.5% of the 800,000 most popular TLS-enabled websites were still vulnerable to Heartbleed.

Problem Solving Process



Pólya: How to Solve It (1/5)

A great discovery solves a great problem but there is a grain of discovery in the solution of *any* problem.

Your problem may be modest; but if it challenges your curiosity and brings into play your inventive faculties, and if you solve it by ***your own means***, you may experience the tension and enjoy the triumph of discovery.

Such experiences at a susceptible age may create a taste for mental work and leave their imprint on mind and character for a lifetime.

– George Pólya

Pólya: How to Solve It (2/5)

- **Phase 1: Understanding the problem**
- Phase 2: Devising a plan
- Phase 3: Carrying out the plan
- Phase 4: Looking back
  - What is the unknown? What are the data?
  - What is the condition? Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown?
  - Draw a figure. Introduce suitable notation.

Pólya: How to Solve It (3/5)

- Phase 1: Understanding the problem
- **Phase 2: Devising a plan**
- Phase 3: Carrying out the plan
- Phase 4: Looking back
  - Have you seen the problem before? Do you know a related problem?
  - Look at the unknown. Think of a problem having the same or similar unknown.
  - Split the problem into smaller sub-problems.

Pólya: How to Solve It (4/5)

- Phase 1: Understanding the problem
- Phase 2: Devising a plan
- **Phase 3: Carrying out the plan**
- Phase 4: Looking back
  - Carry out your plan of the solution. Check each step.
  - Can you see clearly that the step is correct?
  - Can you prove that it is correct?

Pólya: How to Solve It (5/5)

- Phase 1: Understanding the problem
- Phase 2: Devising a plan
- Phase 3: Carrying out the plan
- **Phase 4: Looking back**
  - Can you check the result?
  - Can you derive the result differently?
  - Can you use the result, or the method, for some other problem?

Algorithmic Problem Solving

- An **algorithm** is a well-defined computational procedure consisting of *a set of instructions*, that takes some value or set of values, as *input*, and produces some value or set of values, as *output*.

Algorithm

- Each step of an algorithm must be **exact**.
- An algorithm must **terminate**.
- An algorithm must be **effective**.
- An algorithm must be **general**.
- Can be presented in *pseudo-code* or *flowchart*.

Euclidean algorithm

- First documented algorithm by Greek mathematician Euclid in 300 B.C.
  - To compute the **GCD** (greatest common divisor) of two integers.

1. Let A and B be integers with $A > B \geq 0$.
2. If $B = 0$, then the GCD is A and algorithm ends.
3. Otherwise, find q and r such that

   $$A = q.B + r \quad \text{where } 0 \leq r < B$$

4. Replace A by B, and B by r. Go to step 2.

Euclid in C

```
#include <stdio.h>

void main() {
   int a, b;
   printf("Input two positive numbers:  ");
   scanf("%d %d", &a, &b);
   while (a != b) {
      if (a > b) a = a - b;
      else b = b - a;
   }
   printf("The GCD is %d \n", a);
}
```

NOTE: there is some distance between the English version above and this one in C

Find maximum and average of a list of numbers (1/3)

- Version 1

Declare variables *sum*, *count* and *max*.

First, you initialise *sum*, *count* and *max* to zero.

Then, you enter the input numbers, one by one.

For each number that you have entered, assign it to *num* and add it to the *sum*.

Increase *count* by 1.

At the same time, you compare *num* with *max*. If *num* is larger than *max*, let *max* be *num* instead.

After all the numbers have been entered, you divide *sum* by the numbers of items entered, and let *ave* be this result.

Print *max* and *ave*.

End of algorithm.

Find maximum and average of a list of numbers (2/3)

- Version 2

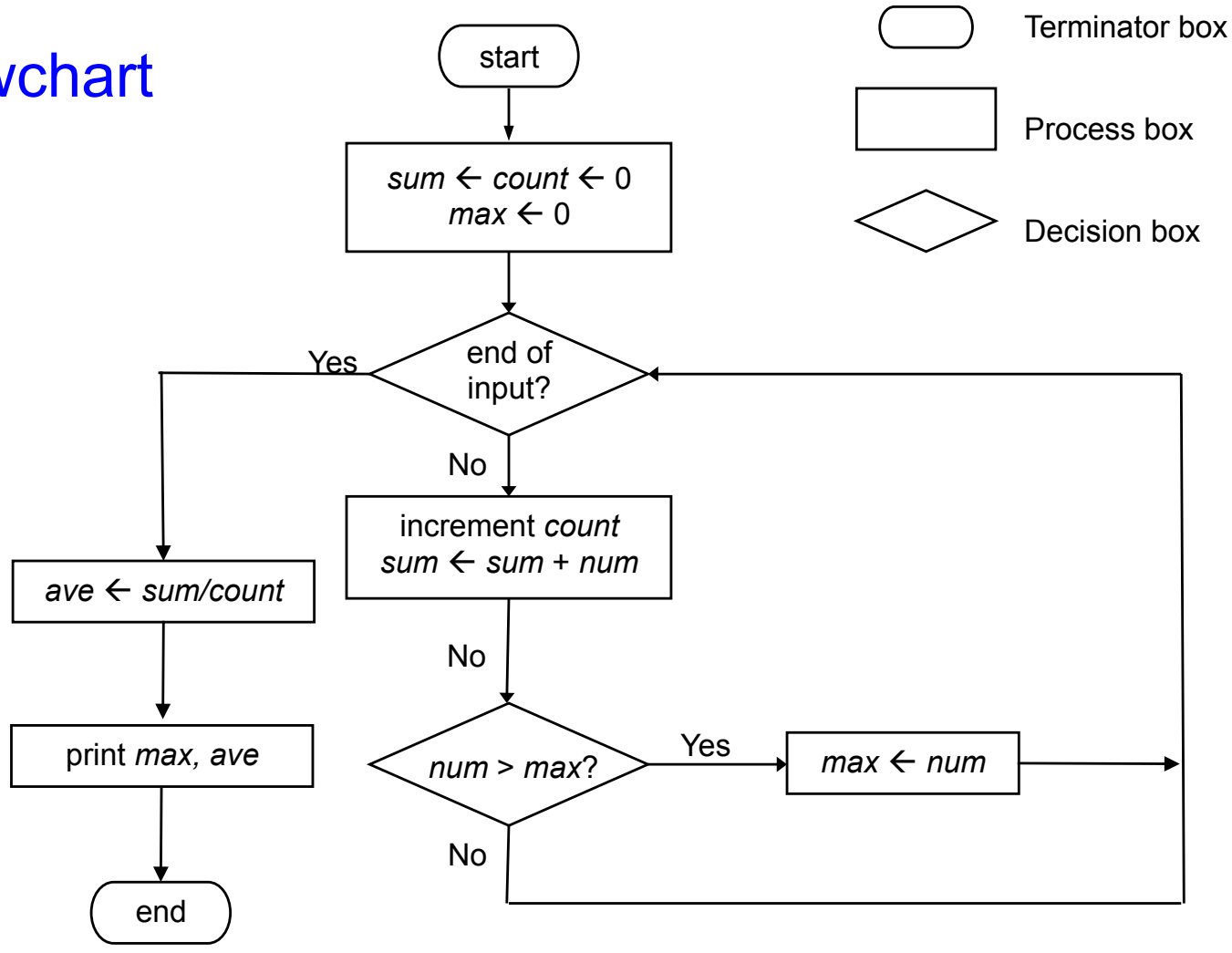
```
sum ← count ← 0      // sum = sum of numbers
                     // count = how many numbers are entered?
max ← 0              // max to hold the largest value eventually
for each num entered,
        count ← count + 1
        sum ← sum + num
        if num > max
                then max ← num

ave ← sum / count

print max, ave
```

Find maximum and average of a list of numbers (3/3)

- Flowchart



Testing

- The final step is testing the solution to make sure it is completely correct
- First test with data from a *hand example* for which we know the solution
- Once the solution works for the hand-calculated example, we should also test it with additional sets of data to be sure that the solution works for other valid sets of data
- Howover, we often cannot cover all possible sets of data

Finally ... Why Should I take CS1010E ?

- General Problem Solving via *Computational Thinking*
- Intro to a number of *fundamental algorithms*
- C exposes clearly the link between software and hardware; hence easier to *master new languages* starting from C
- Be able to build tools for general professional use beyond off-the-shelf productivity software

Next Lecture

Simple C Programs