Introduction to Computer Programming

Mr. Dave Clausen
La Cañada High School

Computer Programs

- Software refers to programs that make the computer perform some task.
- A *program* is a set of instructions that tells the computer what to do.
- When you have written a program, the computer will behave exactly as you have instructed it. It will do no more or no less than what is contained in your specific instructions.

Writing Programs

- Learning to write programs requires two skills.
  - You need to use specific terminology and punctuation that can be understood by the machine; that is, you need to learn a programming language.
  - You need to develop a plan for solving a particular problem. This plan–or algorithm–is a sequence of steps that, when followed, will lead to a solution of the problem.

Solving Problems

- Initially, you may think that learning a language is the more difficult task because your problems will have relatively easy solutions. Nothing could be further from the truth!
- **The single most important thing you can do as a student of computer science is to develop the skill to solve problems.**
- Once you have this skill, you can learn to write programs in several different languages.

What Is a Computer Language?

- A microprocessor is designed to "understand" a set of commands called an "*instruction set*"
- All instructions must be provided to the CPU in its native language, called *machine language*.
- All data transmission, manipulation, storage, and retrieval is done by the machine using electrical pulses representing sequences of binary digits.
- If eight-digit binary codes are used, there are 256 numbered instructions from 0000000 to 1111111.

Machine Language

- Instructions for adding two numbers would consist of a sequence of these eight-digit codes from 00000000 to 11111111.
- Instructions written in this form are referred to as *machine language.*
- It is the native language that the CPU "speaks" and "understands".
- It is possible to write an entire program in machine language. However, this is very time consuming and difficult to read and understand.

Programming Languages

- Fortunately, special languages have been developed that are more easily understood (than machine language).
- These special languages are called *programming languages*.
- These languages provide a way to write computer programs that are understood by both computers and people.
- Programming languages have their own vocabulary and rules of usage.
- Some languages are very technical, while others are similar to English.

Assembly Language

- The programming language that is most like machine language is *assembly language*.
- Assembly language uses letters and numbers to represent machine language instructions.
- An *assembler* is a program that reads the codes the programmer writes in assembly language and "assembles" a machine language program based on those codes.
- However, assembly language is still difficult to read.

Comparing Machine Language & Assembly Language

- For example, the machine code for adding two integers might be:

0100001100111010001111010100000100101011010000010

- While the assembly language code might be:

  LOAD A

  ADD B

  STORE C

  - This causes the number in A to be added to the number in B, and the result is stored for later use in C.

Low Level Languages

- Machine Language and Assembly Language are both called *low-level languages*.
- In a low-level language, it is necessary for the programmer to know the instruction set of the CPU in order to program the computer.
- Each instruction in a low-level language corresponds to **one** or only a **few** microprocessor instructions.

High Level Languages

- A *high-level language* is any programming language that uses words and symbols to make it relatively easy to read and write a computer program.
- In a high-level language, instructions do not necessarily correspond one-to-one with the instruction set of the CPU.
- One command in a high-level language may correspond to **many** microprocessor instructions.

High Level Languages 2

- Many *high-level languages* have been developed. These include:
- FORTRAN, COBOL, BASIC, Logo, Pascal, C, C++, Java, and others.
- These languages simplify even further the terminology and symbolism necessary for directing the machine to perform various manipulations of data.

Advantages Of High Level Languages

- High Level Languages:
  - Reduce the number of instructions that must be written.
  - Allow programs to be written in a shorter amount of time than a low-level language would take.
  - Reduce the number of errors that are made, because…
    - The instructions are easier to read.
  - Are more portable (the programs are easier to move among computers with different microprocessors).

Advantages Of Low Level Languages

- Low Level Languages:
  - Instructions can be written to enable the computer to do anything that the hardware will follow.
  - Require less memory
  - Run more quickly

High Level Language Examples

- Consider the following programs that add two numbers together:

BASIC

```
10 I = 3
20 J = 2
30 K = I + J
```

Pascal

```
program AddIt;
var
  i, j, k : integer;
  begin
   i := 3;
   j := 2;
   k := i + j;
  end.
```

C++

```
int main( )
{
   int i, j, k;
   i = 3;
   j = 2;
   k = i + j;
   return 0;
}
```

LOGO

```
to add :I :J :K
MAKE “I :3
MAKE “J :2
MAKE “K :I + :J
end
```

Interpreters and Compilers

- Programmers writing in a high-level language enter the program's instructions into a text editor.
- The files saved in this format are called *text files*.
- A program written in a high-level language is called *source code*.
- The programs are translated into machine language by interpreters or compilers.
- The resulting machine language code is known as *object code*.

Interpreters

- An interpreter is a program that translates the source code of a high-level language into machine language.
- Each instruction is interpreted from the programming language as needed (line by line of code).
- Every time the program is run, the interpreter must translate each instruction again.
- In order to "run" the program, the interpreter must be loaded into the computer's memory.

Compilers

- A compiler is another program that translates a high-level language into machine language.
- A compiler makes the translation once so that the source code don't have to be translated each time the program is run.
  - The source code is translated into a file called an *object file*.
  - A program called a *linker* is used to create an *executable program*.
  - Most modern compilers let you compile and link in a single operation, and have an "IDE" (Integrated Development Environment) to enter text, debug, compile, link, and run programs.

Debug

- Bug: An error in coding or logic that causes a program to malfunction or to produce incorrect results.
- Debug: To detect, locate, and correct logical or syntactical errors in a program.
- Folklore attributes the first use of the term "bug" to a problem in one of the first electronic computers that was traced to a moth caught between the contacts of a relay in the machine.

http://www.microsoft.com/canada/home/terms/2.7.1.1_B.asp

Programming Languages: First Generation

- Generation 1 – Late 1940s to Early 1950s: Machine Languages
  - Programmers entered programs and data directly into RAM using 1s and 0s
  - Several disadvantages existed:
    - Coding was error prone, tedious, and slow
    - Modifying programs was extremely difficult
    - It was nearly impossible for a person to decipher someone else's program
    - Programs were not portable

Programming Languages: Second Generation

- Generation 2 – Early 1950s to Present: Assembly Languages
  - Uses mnemonic symbols to represent instructions and data
  - Assembly language is:
    - More programmer friendly than machine language
    - Tedious to use and difficult to modify
    - Since each type of computer has its own unique assembly language, it is not portable

Programming Languages: Third Generation

- Generation 3 – Mid-1950s to Present: High-Level Languages
  - Designed to be human friendly – easy to read, write, and understand
  - Each instruction corresponds to many instructions in machine language
  - Translation to machine language occurs through a program called a 'compiler'
  - Examples: FORTRAN, COBOL, BASIC, C, Pascal, C++, and Java

Basic Approaches of Programming

- High-level programming languages utilize two different approaches
  - Procedural approach
    - Examples: COBOL, FORTRAN, BASIC, C, C++, LOGO, and Pascal
  - Object-Oriented Programming approach
    - Examples: Smalltalk, C++, and Java

What Is a Program?

- Program
  - A list of instructions written in a special code, or language.
  - The program tells the computer which operations to perform,
  - and in what sequence to perform them.
  - Garbage In, Garbage Out (G.I.G.O.)
  - Get what you asked for, not necessarily what you want.

Why Programming?

- To Develop Problem Solving Skills
  - It is very important to develop problem solving skills. Programming is all about solving problems.
  - Requires creativity and careful thought.
  - Analyze the problem and break it down into manageable parts (modules, procedures, functions)
- It's also rewarding!

Program Development

- Planning is a critical issue
  - Don't type in code "off the top of your head"
- Programming Takes Time
  - Plan on writing several revisions
  - Debugging your program
- Programming requires precision
  - One misplaced semi-colon will stop the program

Exercise in Frustration

- Plan well (using paper and pencil)
- Start early
- Be patient
- Handle Frustration
- Work Hard
- Don't let someone else do part of the program for you.
- Understand the Concepts Yourself!
- Solve the problem yourself!

Step 1
Good Programming Habits

- 1. Analysis
  - Is the computer the appropriate tool for solving this problem?
  - Would the problem be better solved with human interaction or paper and pencil?
  - Sometimes human judgment is preferable.

Step 2
Good Programming Habits

- 2. Specification of the Problem
  - Formulate a clear and precise statement of what is to be done (clear and unambiguous).
  - Know what data are available
  - Know what may be assumed
  - Know what output is desired & the form it should take
  - Divide the problem into sub problems
  - Doesn't discuss "how to" solve the problem yet.

Step 3
Good Programming Habits

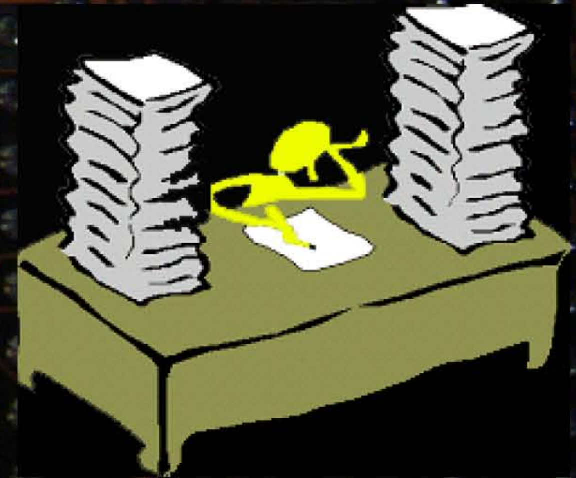
- 3. Develop an Algorithm
  - Algorithm:
    - a finite sequence of <u>effective</u> statements that when applied to the problem, will solve it.
  - Effective Statement:
    - a clear unambiguous instruction that can be carried out.
  - Algorithms should have:
    - specific beginning and ending that is reached in a reasonable amount of time (a finite amount of time).
  - This is done before sitting down at the computer.

Step 3.5
Good Programming Habits

- 3.5 Document the Program
  - Programming Style
    - Upper / Lower Case, Indenting, format
  - Comments
  - Descriptive Identifier Names
    - Variables, Constants, Procedures, Functions
  - Pre & Post Conditions
    - For each Procedure and Function
  - Output

Step 4
Good Programming Habits

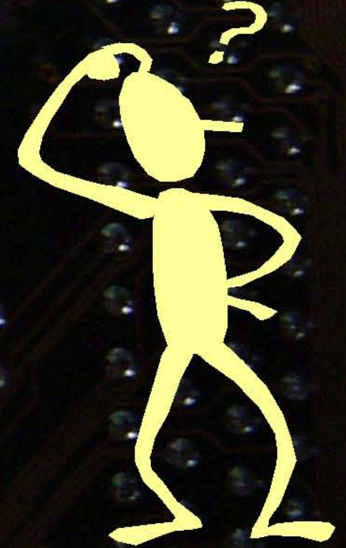
- 4. Code the Program
  - After algorithms are correct
  - Desk check your program
    - Without the computer, just paper and pencil



- 4.1 Type and Run the Program
  - Look for errors
    - Syntax Errors (semi colon missing, etc.)
    - Logic Errors (divide by zero, etc.)

Step 4.2 Good Programming Habits

- 4.2 Test the Results
  - Does it produce the correct solution?
  - Check results with paper and pencil.
  - Does it work for all cases?
    - Border, Edge, Extreme Cases
  - Revise the program if not correct.
  - The coding process is not completed until the program has been tested thoroughly and works properly (recheck the specifications).



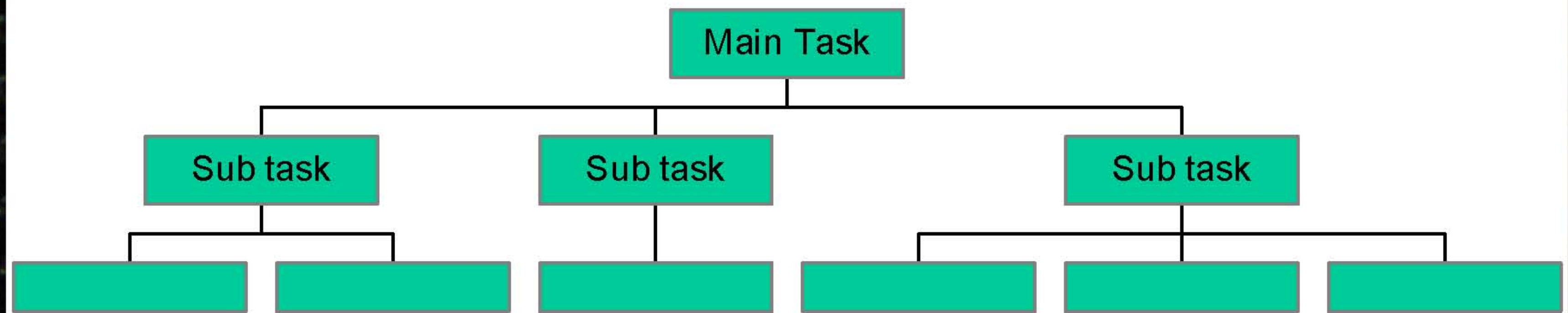
Step 5
Good Programming Habits

- 5. Interpretation
  - The program may execute without any obvious errors.
  - It may not produce the results which solve the problem.
    - G.I.G.O Get what you ask for, not what you want.
    - Recheck your program with the original specifications

Top Down Design

- Subdivide the problem into major tasks
  - Subdivide each major task into smaller tasks
    - Keep subdividing until each task is easily solved.
- Each subdivision is called stepwise refinement.
- Each task is called a module
- We can use a structure chart to show relationships between modules.

Top Down Design 2



Top Down Design 3

- Pseudocode
  - is written in English with C++ like sentence structure and indentations.
  - Major Tasks are numbered with whole numbers
  - Subtasks use decimal points for outline.

Top Down Design 4

1. Get Information
   1.1. Get starting balance
   1.2. Get transaction type
   1.3. Get transaction amount
2. Perform computations
   2.1. If deposit then
        add to balance
        Else
        subtract from balance
3. Display the results
   3.1. Display starting balance
   3.2. Display transaction
        3.2.1. Display transaction type
        3.2.2. Display transaction amount
   3.3. Display ending balance

Writing Programs

- Vocabulary
  - reserved words
    - have a predefined meaning that can't be changed
  - library identifiers
    - words defined in standard libraries
  - programmer supplied identifiers
    - defined by the programmer following a well defined set of rules

Writing Programs 2

- Words are CaSe SeNsItIvE
  - For constants use ALL CAPS (UPPERCASE)
  - For reserved words and identifiers use lowercase
- Syntax
  - rules for construction of valid statements, including
    - order of words
    - punctuation

Writing Code

- Executable Statement
  - basic unit of grammar
    - library identifiers, programmer defined identifiers, reserved words, numbers and/or characters
  - A semicolon terminates a statement in many programming languages
- Programs should be readable

noformat.cpp format.cpp

The Use of Comments

- Comments should be included to help make the program more clear to someone reading the code other than the author.
- Use comments after the header to explain the function of the program, & throughout the program

Test Programs

- Test programs are short programs written to provide an answer to a specific question.
- You can try something out
- Practice the programming language
- Ask "what if" questions
- Experiment: try and see