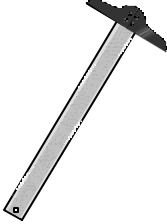


Topic: Program design

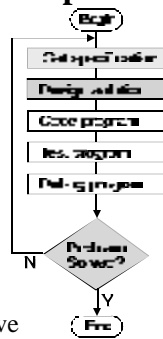
- The software development process
- Pseudocode and flowcharts
- Control structures:
 - Sequence
 - Decision
 - Loop
- Modular decomposition and stepwise refinement
- Object-oriented design and UML



David Keil 1/03 1

The software development process






- Understanding the problem: *what*
 - input
 - output
 - their relationship
- Designing a solution: *how*
- Algorithms are language independent
- Coding, testing come later
- Development process is iterative



David Keil 1/03 2

Two notations for low-level design

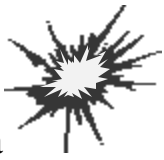
- Both notations show order of execution
- Pseudocode
 - informal
 - precise
 - text outline format
- Flowcharts
 - graphical
 - shapes denote steps
 - arrows show flow of control

	Begin/end
	Processing
	I/O
	Decision test
	Flow

David Keil 1/03 3

Algorithm:

A precise plan to solve a problem in a finite number of steps



- Program designs use algorithms
- Most computation is algorithmic
- Flowcharts and pseudocode can express algorithms

David Keil 1/03 4

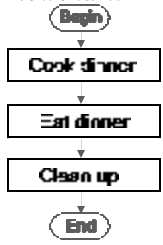
The *sequence* control structure

Problem:
Accomplish dinner routine

Pseudocode:

1. Cook dinner
2. Eat dinner
3. Clean up

Flowchart:



David Keil 1/03 5

An algorithm to add numbers

1. Prompt for integers *input1*, *input2*
2. $sum \leftarrow input1 + input2$
3. Display *sum*

- In pseudocode and flowcharts, the symbol \leftarrow stands for assignment of a value to a variable
- An algorithm is almost always *general purpose*: it works with *variables*, not only specific constant values

David Keil 1/03 6

The *decision* control structure

Problem:
Prepare to do homework

Pseudocode:
If exercises are confusing
review chapter

Flowchart:

David Keil 1/03 7

Finding absolute value

Input n
If $n \geq 0$
display n
otherwise
display $(-n)$

- With the branch control structure, one and only one of the alternatives executes
- In pseudocode, the subordinate (conditional) steps are normally indented

David Keil 1/03 8

Multiple alternatives

- Require multiple diamonds in flowchart

If $n = 1$ display 'a'
 otherwise
 if $n = 2$ display 'b'
 otherwise
 if $n = 3$ display 'c'
 otherwise
 if $n = 4$ display 'd'

David Keil 1/03 9

The *repetition* (loop) control structure

Flowchart:

Problem:
Telephone someone

Pseudocode:
Repeat
Dial number
until someone answers

David Keil 1/03 10

Counting to 10

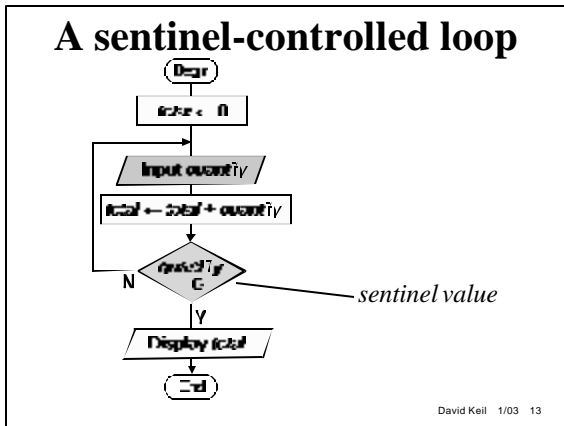
- A *counted* loop; counter is i

David Keil 1/03 11

Loop invariants

- A loop invariant is a logical (true/false) assertion about values and outputs at the beginning of a loop iteration
- *Example*
 Loop invariants for the previous slide are
 - The value of i is between 0 and 11
 - All values in $\{1 \dots i - 1\}$ have been displayed
- Loop invariants help assure correctness

David Keil 1/03 12



Tracing an algorithm

- Allows designer to check result of algorithm, including internal (undisplayed) values
- Use one column per value traced; one row per loop iteration.
- *Example* (See previous slide):

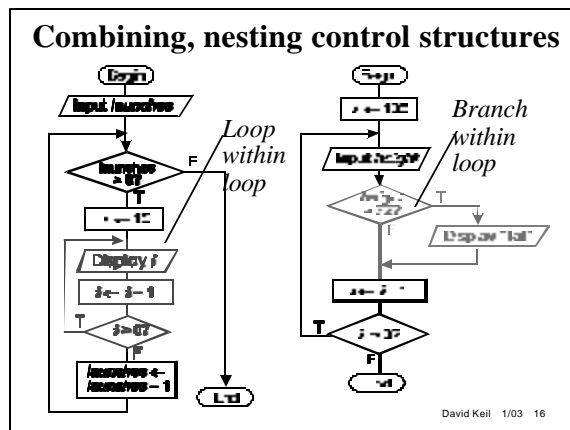
quantity	total	output
	0	
3	3	
2	5	
1	6	
0	6	6

David Keil 1/03 14

Three control structures are sufficient

- Three control structures: *sequence*, *branch*, and *loop*
- They are used in structured programming; may be combined
- *Any computable problem can be solve* using only these three
- *Note*: Some problems are not computable

David Keil 1/03 15



Structured flowcharts have one entrance, one exit

```

    graph TD
      subgraph Unstructured
        Start3([Begin]) --> Dec5{ }
        Dec5 -- N --> End3([End])
        Dec5 -- Y --> Loop3[ ]
        Loop3 --> Dec5
      end

      subgraph Structured
        Start4([Begin]) --> Dec6{ }
        Dec6 -- N --> End4([End])
        Dec6 -- Y --> Loop4[ ]
        Loop4 --> Dec6
      end
  
```

- Structured pseudocode also avoids "Go to step..."

David Keil 1/03 17

The picture tells a story

- A flowchart should show what happens, at a glance

Not too clear More readable

David Keil 1/03 18

Modular decomposition

- One strategy: divide and conquer
- All languages use modularity
- An organization is modular

- Modular design may be *top-down*
- Subprograms implement modular designs

David Keil 1/03 19

Module hierarchy charts tell which subprogram uses which

- Module *do_problem* uses modules *calculate*, *get_input* and *display_result*
- *calculate* uses *add* and *divide*
- Stepwise refinement may lead to further decomposition
- Module hierarchy contains names, not steps, in contrast to flowchart

David Keil 1/03 20

A divide-and-conquer strategy to solve problem of walking *n* steps

Walk (steps)
 If steps > 0
 Take a step
 Walk (steps - 1)

- This algorithm is *recursive* because it uses itself

David Keil 1/03 21

Review of functions

- *Function*: A set that is a mapping from one set to itself or to another set
- *Examples*:
 $Index('B') = 2$ $Odd(3) = true$ $Twice(1) = 2$

David Keil 1/03 22

Operators and functions

- Two possible ways to express the same transformation of data
- *Examples*:
 $a + b$ $sum(a, b)$
 $(2 + 5) \times 4$ $product(sum(2,5), 4)$
 $\sum_{k=1}^4 k$ $sum(1,2,3,4)$

David Keil 1/03 23

Computable function:

- A mapping obtained by an effective step-by-step procedure
- *Examples*: successor, sum
- Computer programming is concerned with coding computable functions
- An uncomputable function:
 “Does this program work correctly?”

David Keil 1/03 24

Functions vs. computation

- A function in mathematics is a passive *set of pairs*
- A computation entails *action*
- A computer program or subprogram computes a mathematical function
- A C or C++ “function” is actually a subprogram, not a mathematical function

Recursive functions and computable functions

- For natural numbers a, b in $\{0, 1, 2, 3, \dots\}$:

$$sum(a,b) = \begin{cases} b & \text{if } a = 0 \\ sum(a-1, b+1) & \text{otherwise} \end{cases}$$

- The definition of *sum* is *recursive* because it uses itself
- The form of this definition is a *recurrence*
- It guarantees computability

Some recurrences

$$sum(a,b) = \begin{cases} b & \text{if } a = 0 \\ sum(a-1, b+1) & \text{otherwise} \end{cases}$$

$$product(a,b) = \begin{cases} 0 & \text{if } b = 0 \\ a & \text{if } b = 1 \\ a + product(a, b-1) & \text{otherwise} \end{cases}$$

$$even(a) = \begin{cases} true & \text{if } a = 0 \\ false & \text{if } a = 1 \\ even(a-2) & \text{otherwise} \end{cases}$$

Russian peasants' algorithm

Product (a, b)

```

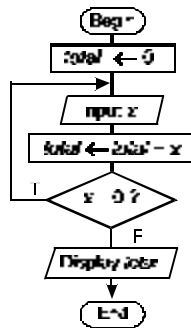
result ← 0
while a > 0
  if a is odd
    add b to result
  a ← ⌊a ÷ 2⌋
  b ← b × 2
Return result
    
```

Example: $13 \times 5 = 65$

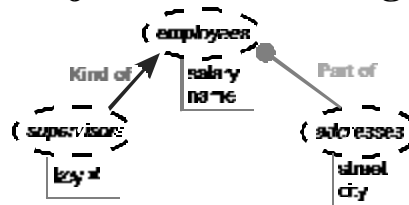
	a	b	product
			0
1.	13	5	5
2.	6	10	5
3.	3	20	25
4.	1	40	65
5.	0	80	

Non-algorithmic computations

- The process diagrammed at right will never terminate if user continues to input a non-zero value



Object-oriented design



- Object-oriented design focuses on classes of objects and their attributes and behaviors

Unified Modeling Language

- A new standard graphical notation for system specification and design
- Motivated by need to depict interactions between systems and their environments, initiated by external *actors*
- Diagrams include *use-case, activity, class, state, interaction*
- Supports an *object-oriented methodology*

David Keil 1/03 31

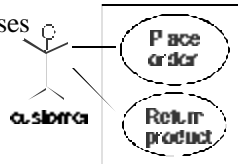
Software projects start with problem descriptions

- What service must the software provide?
Example: Process customer orders from catalog
- What assumptions are made?
Example: Some customers may find Web access convenient
- What risks are involved?
Example: Some users are inexperienced with Web access

David Keil 1/03 32

Use cases and system specification

- Use case: A typical interaction between system and an *actor* in its environment
- Actor: A role, such as customer, manager, supplier, salesperson
- Actors initiate use cases
- Example of two uses cases (UML use-case diagram):

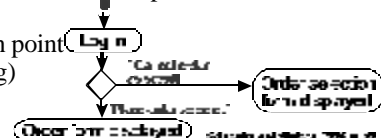


David Keil 1/03 33

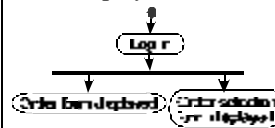
UML activity diagrams

- Depict order in which steps occur

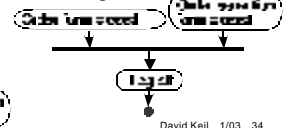
- *Examples:*
A decision point (branching)



- A fork (2 forms display at once)



- A join (execution converges)



David Keil 1/03 34

Four CS I skills objectives

- Vague specification → precise specification (analysis)
- Problem specification → flowchart (design)
- Flowchart → trace (design verification)
- Flowchart → program (coding)
- Program → trace (testing)

David Keil 1/03 35

Discussion problems

- Tell what control structures are required.
- Solve, using pseudocode or flowchart.

1. Calculate the cost of transporting a crate of goods, given user input for the dimensions of the crate, the cost per mile per cubic foot, and the number of miles.
2. Input and echo retail-inventory part numbers and quantities until the user inputs part # 0.
3. Input the dimensions of a rectangular plot of land and tell whether or not its area is over an acre (~42,000 square feet).

David Keil 1/03 36

Loop discussion problems

1. Design and trace an algorithm that will loop to accept input of exactly three pairs of integers (a, b) and compute and display the value of $a - b$ for each input pair.
2. Design and trace a loop to compute and display the product of two input non-negative integers. Display nothing if input includes a negative number.