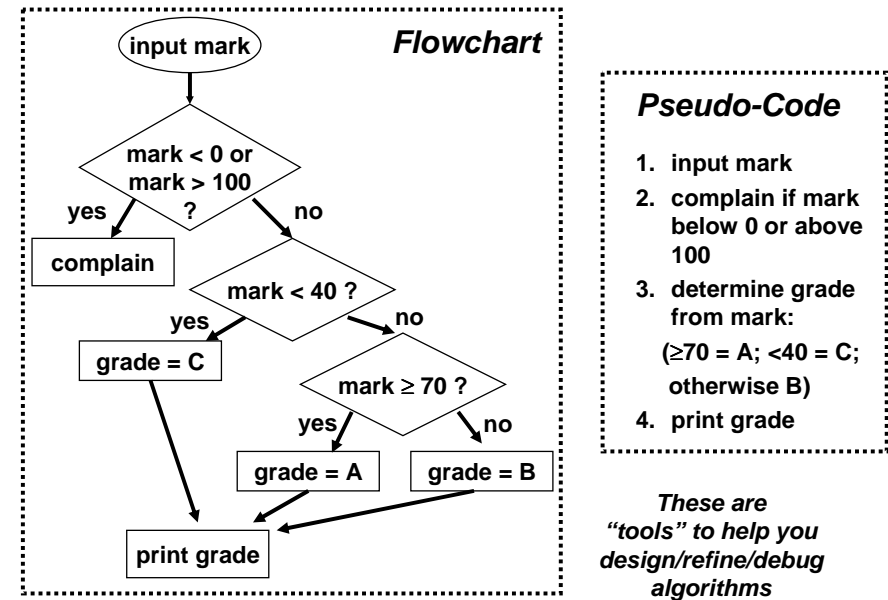## Algorithm Development & Stepwise Refinement

Algorithm Development

- In order for a computer to carry out some task, it has to be supplied with a ***program***, which is an implementation of an ***algorithm***. This is expressed in a computer programming language; however it is possible (and desirable) to talk and reason about algorithms in higher-level terms.

  - Developing a correct algorithm can be a significant intellectual challenge – by contrast, coding it should be straightforward (although coding it ***well*** may not be!)

- The most widely used notations for developing algorithms are ***flowcharts*** and ***pseudo-code***. These are independent of the programming language to be used to implement the algorithm.

  - A flowchart is a diagram containing lines representing all the possible paths through the program.

  - Pseudo-code is a form of "stylised" (or "structured") natural language.

1

Flowchart

Pseudo-Code

1. input mark
2. complain if mark below 0 or above 100
3. determine grade from mark:
   (≥70 = A; <40 = C; otherwise B)
4. print grade

These are "tools" to help you design/refine/debug algorithms

2

Algorithm Development (contd.)

- One of the problems encountered when writing programs is that of **preciseness**. A common fault among algorithms is that the process described is ***almost*** the intended one, but not quite.

  - Analogy: giving directions, following a recipe. These are rarely completely precise, but instead rely on the common sense of the person receiving the instructions. However computers are not equipped with common sense!

- Another common failing is that execution ***usually*** results in the intended process being carried out, but in certain circumstances (unforeseen or overlooked by the designer) it does not.

  - For example, consider the following algorithm to calculate the flight time of an aircraft using information from the timetable:

    Look up departure time
    Look up arrival time
    Subtract departure time from arrival time

  This algorithm will usually give the correct result, but the subtraction should take into account the special case when the plane arrives on the day after departure. Also, what about:
    Different time zones? Daylight savings time?

3

- Another required property of an algorithm is that each step can actually be carried out – in other words, the algorithm is **executable**. The point here is to make sure there are no "impossible" or unknown steps in your algorithm (e.g. algorithm relies on solving a sub-problem which is known to have no solution; algorithm asserts that a sub-problem will be solved without specifying how; etc).

- Most processes are supposed to terminate! There are examples of some which don't need to, but we assume that all programs we are interested in should.

- Thus the designer of an algorithm must ensure:
  - **Preciseness** of the algorithm (no ambiguities)
  - **All possible circumstances** are handled
  - The algorithm is **executable**
  - **Termination** of the algorithm

- Also have to worry about **Efficiency**
  - an algorithm may work correctly but be inefficient – by taking more time and using more resources than required to solve the problem.
  - becomes more important for larger programs.

4

Stepwise Refinement

- Break a **complex problem** down into a number of **simpler steps**, each of which can be solved by an algorithm which is smaller and simpler than the one required to solve the overall problem.
  — Smaller and simpler, therefore easier to construct and sketch in detail
  — Sub-algorithms can themselves be broken into smaller portions
- **Refinement** of the algorithm continues in this manner until each step is sufficiently detailed.
  — Refinement means replacing existing steps/instructions with a new version that fills in more details.
- *Example: Making tea.* Suppose we have a robot which carries out household tasks. We wish to program the robot to make a cup of tea. An initial attempt at an algorithm might be:
  1. Put tea leaves in pot
  2. Boil water
  3. Add water to pot
  4. Wait 5 minutes
  5. Pour tea into cup

- These steps are probably not detailed enough for the robot. We therefore refine each step into a sequence of smaller steps:

  1. Put tea leaves in pot

  might be refined to
  1.1 Open box of tea
  1.2 Extract one spoonful of tea leaves
  1.3 Tip spoonful into pot
  1.4 Close box of tea

  Similarly:
  2. Boil water   **5. Pour tea into cup**
  might be refined to   might be refined to
  2.1. Fill kettle with water   **5.1. Pour tea from pot into cup**
  2.2 Switch on kettle   **until cup is full**
  2.3 Wait until water is boiled
  2.4 Switch off kettle

- Suppose the original 5 steps have been refined into sub-algorithms where necessary (e.g. it was not necessary to refine step 4 because it is simple enough for the robot to directly execute).

- Some of the sub-algorithms need further refinement. For example, the step
  2.1. Fill kettle with water
  could be refined to
  2.1.1. Put kettle under tap
  2.1.2. Turn on tap
  2.1.3. Wait until kettle is full
  2.1.4. Turn off tap

- Others steps may also require further refinement. After a number of refinements the robot is able to execute every step.

- The program is then constructed by translating the final refinement of each step into C program statements.

Original Algorithm	First Refinement	Second Refinement
1. Put tea leaves in pot	**1.1 Open box of tea**	**1.1.1 Take tea box from shelf**
		1.1.2 Remove lid from box
	1.2 Extract one spoonful	
	1.3 Tip spoonful into pot	
	1.4 Close box of tea	**1.4.1 Put lid on box**
		1.4.2 Replace tea box on shelf
2. Boil Water	**2.1 Fill kettle with water**	**2.1.1 Put kettle under tap**
		2.1.2 Turn on tap
		2.1.3 Wait until kettle is full
		2.1.4 Turn off tap
	2.2 Switch on kettle	
	2.3 Wait until water boiled	**2.3.1 Wait until kettle whistles**
	2.4 Switch off kettle	
3. Add water to pot	**3.1 Pour water from kettle until pot is full**	
4. Wait 5 Minutes		
5. Pour tea into cup	**5.1 Pour tea from pot into cup until cup is full**	

- When using stepwise refinement the designer must know **when to stop refining**. They must know when a particular step of the algorithm is sufficiently described to need no further refinement.

  — In this example the designer must know that the instruction **Switch on kettle** is directly executable by the robot, but that **Fill kettle with water** is not.

- In our case we are issuing instructions to a computer using a high-level programming language and hence **experience** will tell us when a step is directly implementable in that language or not.

- The above algorithm consists of a **sequence of steps**, each of which will be executed exactly once and in order – termination of the last step implies termination of the algorithm. However, algorithms with only sequences of steps can't do much…

  — Example: What happens if the tea-box is empty?

- If the tea-box is empty we wish to specify an extra step:
  Get new box of tea from cupboard

  — This step would not be carried out unless the tea-box is empty, and hence an algorithm incorporating this would not be entirely sequential anymore.

- We can express this by rewriting step 1.1 as
  1.1.1. Take tea box from shelf
  1.1.2. If box is empty
      then get new box from cupboard
  1.1.3. Remove lid from box

  — Step 1.1.2 expresses both the step to be selected and the condition under which this selection should be made.

- More complicated conditions can use **AND**, **OR**, **NOT**

- Another common requirement is the need for *iteration*.

  — Example: suppose all we have access to is a timer which waits 1 second. Therefore we cannot simply say "wait 5 minutes"; this step must be refined further, e.g.

    4.1. Set counter to 1
    4.2. WHILE counter<=300 DO
        wait
        increase counter by 1

- Although the above iteration is expressed using "while", this does not mean that the program must use a `while` loop. Any of the loop structures supported in C can be used. Similarly, the loop in the program does not have to use a loop counter which counts from 1 up to 300, e.g. you may prefer to count down to 0…

- **In most programming languages, you can/should explicitly structure your program into "modules" that each do some specific task. Why? Makes programs easier to build, understand, debug, maintain; and encourages re-use of modules in different programs.**
- **In C, this can be done with a *function*: a piece of code which is given a name so that it can be invoked by other parts of the program**

```
#include "stdio.h"
char convert_mark(int mark) {
  if (mark < 40) return 'C';
  else if (mark >= 70) return 'A';
  else return 'B';
}
void main(void) {
  int mark;
  char grade;
  printf("enter the mark: ");
  scanf("%d", &mark);
  grade = convert_mark(mark);
  printf("You got a %c\n", grade);
}
```

If refine problem well, each function is fairly short, & its job is clear. (Soon, we'll discuss details of how to write functions; for now the point is just WHY)

- **Usually, the structure of the refinement process is reflected in the structure of the function invocations…**

Sources of errors

- Many errors made in analysing the problem, developing an algorithm, and/or coding the algorithm, only become apparent when you try to compile, run, and test the resulting program.
- The earlier in the development process an error is made, or the later it is discovered, the more serious the consequences.

Sources of errors:
- *Understanding the problem to solve.* An error here may be obvious e.g. your program does nothing useful at all. Or it may be more subtle, and only become apparent when some exceptional condition occurs e.g. a leap year, incompetent user, …
- *Algorithm design.* Mistakes here result in **logic errors**. The program will run, but will not perform the intended task e.g. a program to add numbers which returns 6 when given 3+2 has a logic error.
- *Coding of the algorithm.* Often the compiler will complain, but messages from the compiler can be cryptic. These errors are usually simple to correct e.g. spelling errors, misplaced punctuation, …
- *Runtime.* Errors may appear at run time e.g. divide some number by zero. These errors may be coding errors or logic errors.

13

Sources of errors (contd.)

- Programs rarely run correctly the first time (!)
- Errors are of three types:
  - syntax errors
  - run-time errors
  - logic errors

- **Syntax errors:** detected by the C compiler
  - source code does not conform to one or more of C's grammar rules
  - examples of syntax errors:
    - undeclared variable
    - missing semicolon at end of statement
    - comment not closed
- Often one mistake leads to multiple error messages – can be confusing!

14

Sources of errors (contd.)

- **Run-time errors:**
  - detected and displayed by computer during execution
  - occur when program directs computer to perform illegal operation
    Example: int x=y/0;
  - will stop program execution and display message

- **Logic errors:**
  - caused by faulty algorithm
  - do not usually cause run-time errors
  - Includes errors that do not prevent execution of program
    Example: float f; scanf("%d",&f);
  - difficult to detect for large and complex algorithms
  - sign of error: incorrect program output
  - cure: thorough testing and comparison with expected results

15

Sources of errors & how to reduce them

- Errors made in understanding the problem may well require you to restart from scratch.
  - You may be tempted to start coding, making up the solution and algorithm as you go along. This may work for trivial problems, but is a certain way to waste time and effort for realistic problems.
- A program can be tested by giving it inputs and comparing the observed outputs to the expected ones.
  - Testing is very important but (in general) cannot prove that a program works correctly.
  - For small programs, formal (mathematical) methods can be used to prove that a program will produce the desired outputs for all possible inputs. These methods are rarely applicable to large programs.
- A logic error is referred to as a **bug**, so finding logic errors is called **debugging**.
  - Debugging is a continuous process, leading to an *edit-compile-debug* cycle. General idea: insert extra `printf()` statements showing the values of variables affected by each major step. When you're satisfied program is correct, comment out these `printf()`'s.

16

Debugging tips – common errors in C

- In a `for` loop, initialisation and condition end with semicolons.
  Wrong – `for(initialisation, condition; update)`

- Must use braces in `for` and `while` loops to repeat more than one statement.

- nested structure: first closing brace is associated with innermost structure.
  Example of "unexpected" behaviour:
```
if(c=='y') {
    scanf("%d", &d);
    while(d!=0) {
        sum+=d;
        scanf("%d", &d);
} else printf("end"); /* done even when c=='y' */
```

- inequality test on floating point numbers. Example of wrong way:
```
while(value!=0.0) { /* could have value<1e-9 */
    ...
    value/=2.8; /* now value will be regarded as 0 */
}
```

17

Debugging tips – common errors in C (contd.)

- Should ensure that loop repetition condition will eventually become **false**.
  Example where this doesn't happen:
```
do {
    ...
    printf("one more time?");
    scanf("%d", &again);
} while(again=1); /* assignment, not equality test */
```

- loop count off by one, either too many or too few iterations. Or infinite loop, or loop body never executed. Can be hard to discover!

18