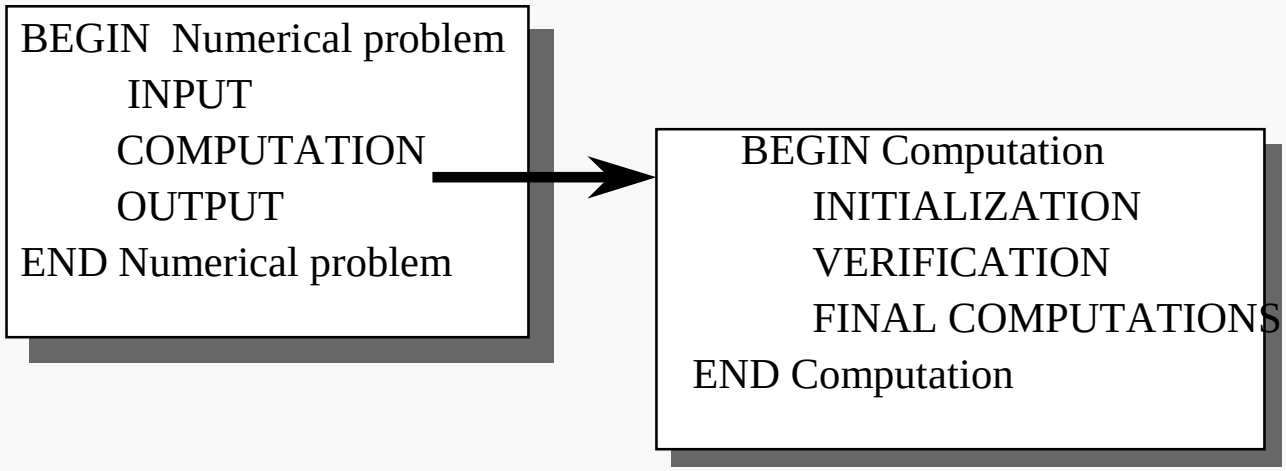
Top Down Design

- A solution method where the problem is broken down into smaller subproblems, which in turn are broken down into smaller problems until each subproblem can be solved in a few steps.
  (Also called: Divide & Conquer strategy)

- Top Down Design <==> Step-Wise Refinement <==> Modular Programming

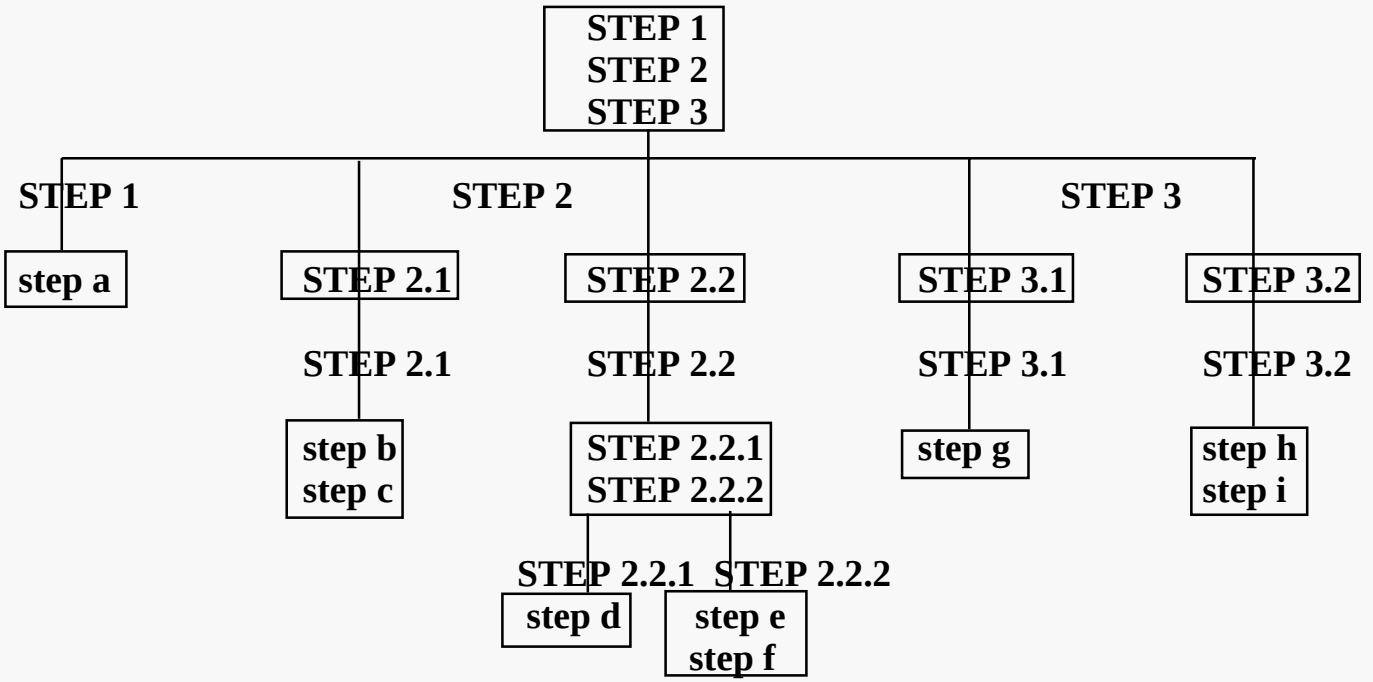
- Problem: An iterative numerical computation



Design Specification

- Techniques:
  1. Module Structure Chart (Hierarchical Tree)
  2. Outlining (writing decomposition)
  3. Pseudocode (mixture of English & C++)

- Module Chart Format:



Outline Design Example

Problem: Determine if a positive integer is a perfect number .

Defn: a perfect number is a positive integer that is equal to the sum of its divisors, excluding the number itself; e.g. 6, 28, 496, . . .

1. Get the number to be tested
2. Determine the divisors of the number
3. Check if the divisors add up to the number.

1.1 Prompt the user for the number (p)
1.2 Read the number (check if number is positive int)
2.1 Set the divisor sum = 1
2.2 check each integer, x, from 2 . . . p/2 to see if x is a divisor of p
    2.2.1 Find the remainder of p/x
        2.2.1.1 remainder = p - truncate(p/x) * x
    2.2.2 If the remainder of p/x is 0 then then add x to the divisor sum
3.1 If divisor sum = p then
        output “p is perfect”
    Else
        output “p is NOT perfect”

Pseudocode Design Example

```
Print “Enter the possible perfect number”
Read the number into:  p

Comment initialize sum of the divisors & divisor
SumOfDivisors  <-- 1
Divisor <-- 2

Comment find all possible divisors
While  Divisor <= p / 2  Do
    R <--  p - (Truncate ( p/Divisor ) )* Divisor
    If R = 0 Then
       SumOfDivisors <-- SumOfDivisors + Divisor
    Endif
    Divisor <-- Divisor + 1
Endwhile

If SumOfDivisors = p  Then
    Print p “ is a perfect number ”
Else
    Print p “ is NOT a perfect number ”
Endif
```

Top-Down Design Example

Problem: How to enter & run a C++ program on a PC. (Sys: MS Visual C++ on a hard disk.)

Main Module.
- Turn the system on.
- Start the MS Visual C system.
- Enter program.
- Run program

–Start the MS Visual C system.
(Start Menu, Programs, MS Visual C++, MS Developer Studio.)
(or move cursor to MS C++ icon (Msdev) and double-click left mouse button.)

–Enter program.
Choose File menu, New, Text File.
Type the program in using the editor.
Use the cursor & the backspace keys to make corrections.
Under the File pull down menu, select Save As; enter the name of your program in the dialog box & save.

–Run Program.
Build (compile) the executable (F7) , create a default workspace (Correct any errors).
If an error message appears, note what it says & then press any key. Otherwise enter data as needed and observe any output, until the program stops running.
Under the File pull down menu, select Exit to quit the MS Visual C system.

Top-Down Design Example

Problem: How to enter & run a C++ program on a Unix workstation. (Sys: FreeBSD on a IntelPC)

Main Module.
Turn the system on & Login.
Enter program.
Run program

- Login.
  Type "guest" after the login: prompt & Type "visitor" after the password: prompt.

- Enter program.
  Type emacs filename.cc to start the editor.
  Type the program into the editor.
  Use ctrl-f, ctrl-b, ctrl-n, ctrl-p & the backspace key to make corrections.
  Press ctrl-x ctrl-s to save the file.

- Run Program.
  Type "gcc -o filename filename.cc" to compile the program.
  If an error message appears below, locate & correct the error & recompile. Otherwise type "filename", enter data as needed & observe any output, until the program stops running.

Program Proofs

Program Proof: code analysis to achieve a formal verification of the correctness of a program's output.

Assertion: statement of relationship between a program's variables and data.

- Every program statement can be preceded & succeeded by an assertion, (termed the precondition & postcondition respectively).
- Program proofs involves applying deductive logic to show that the precondition & postcondition assertions are true for all program statements
- Due to practical time considerations assertions are made & proven only for sequences of statements.

Example:

```
// root ≥ 0          <-- precondition
    root2 = sqrt( root ) ;
// root = root2 * root2   <-- postcondition
```

Better postcondition:
// root <= (root2 * root2) + epsilon

Program Proofs (cont)

Loop Invariant: assertions that are true prior to loop execution & following each loop iteration.

Commonly composed of the boolean expression controlling the loop, its negation, ranges of loop variables, data structure states & file status. Derived by analyzing the setup & modifications of the loop variables.

Example: **// integer multiplication by repeated addition**

```
product = 0 ;
factor = multiplier ;
while (factor > 0) {
        product = product + multiplicand ;
        factor = factor - 1 ;
}
```

Loop Invariant:

"product = multiplicand * (multiplier - factor) ;
factor is in the range 0 ... multiplier"

Program Documentation

A natural outgrowth of the software development method is the production of external program documentation prior to program implementation.

The documentation describes the purpose of a program and its data.

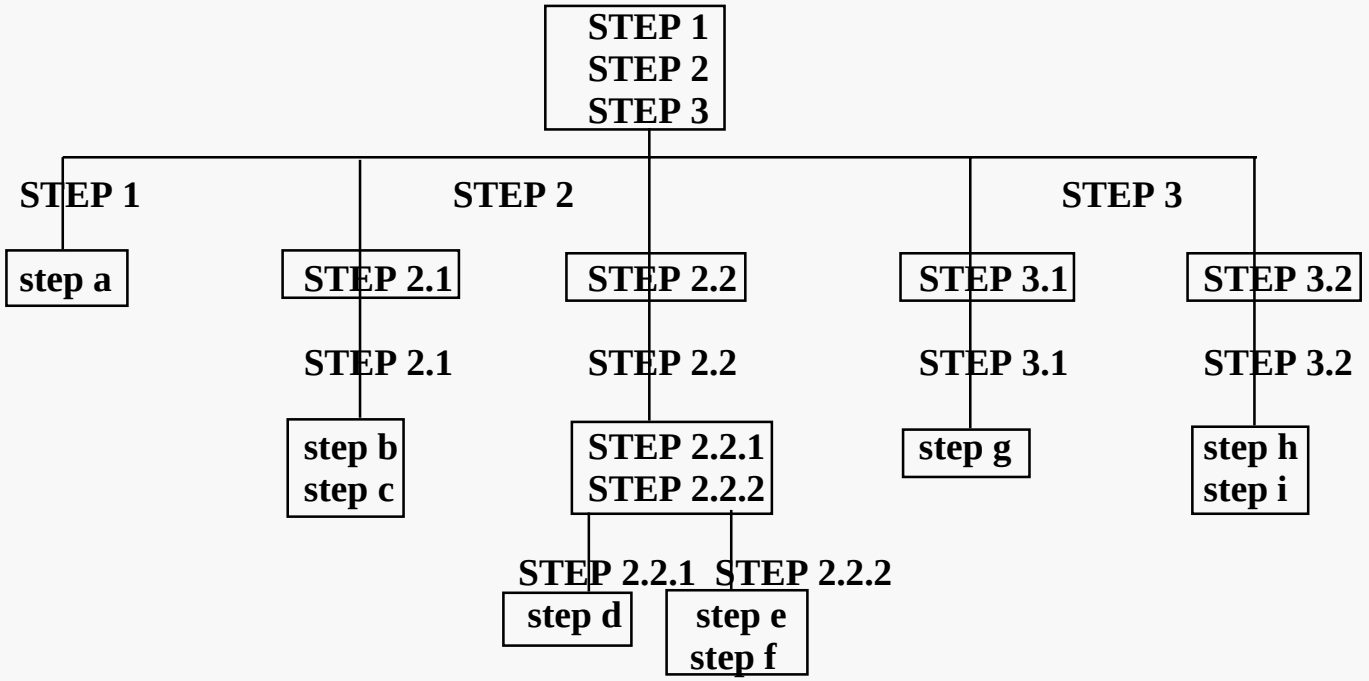
Internal documentation, in the form of in-line comments, as well as module documentation, is produced in conjunction with code development.

See the "Elements of Programming Style" in the appendices.

Modular Decomposition

Divide and Conquer

One of the most fundamental ideas in problem solving involves subdividing a problem into subproblems and solving each subproblem individually. Quite often, the attempt to solve a subproblem introduces new subproblems at a lower level. The subdivision of subproblems should continue until a subproblem can easily be solved. This is similar to the refining of an algorithm.



Structure Chart

A ***structure chart*** is a documentation tool that enables the designer to keep track of the relationships among subproblems throughout this refinement process.

The structure chart gives a detailed description of the original problem in terms of subproblems. This top-down description will make it easier to write a program to solve the original problem.

We can solve each box in the structure chart by using a C++ ***function***. A function is a grouping of statements into a single unit. This single unit can be invoked to solve its subproblem simply by activating it by means of a ***function call***.

Since the structure chart gives a pictorial mechanism for visualizing the solution to a problem and each of the pieces of the structure chart corresponds to a function, the solution to a problem is obtained by correctly invoking a series of functions. In order to solve problems, we need only solve all the subproblems by means of functions. Thus a C++ program is a collection of functions that solve a series of subproblems. This is the essence of top-down design.

Procedural Abstraction

The top-down approach to program design involves ***procedural abstraction***. This enables us to associate meaningful names with more complicated algorithm steps, reference these steps by name, and defer their implementation details until later.

Our programs are written using logically independent sections, similar to the manner in which we developed the solution algorithm (as logically independent steps).

Using logically independent program sections enables the programmer to hide the details of subproblem implementations from one another. This concept, ***information hiding***, is critical in all stages of the software development process.