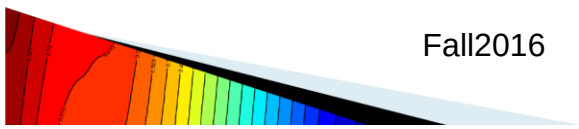
GNG 1106

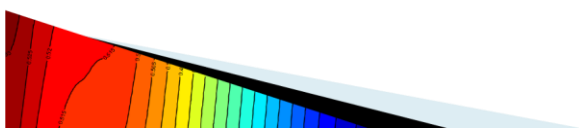
Fundamentals of Engineering Computation

Engineering Problem Solving Method

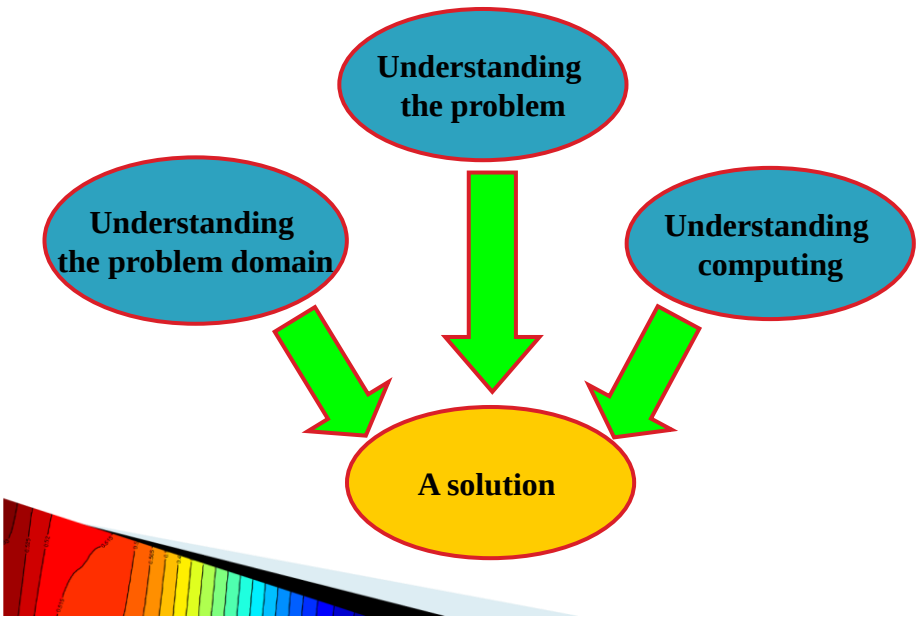


Fall2016

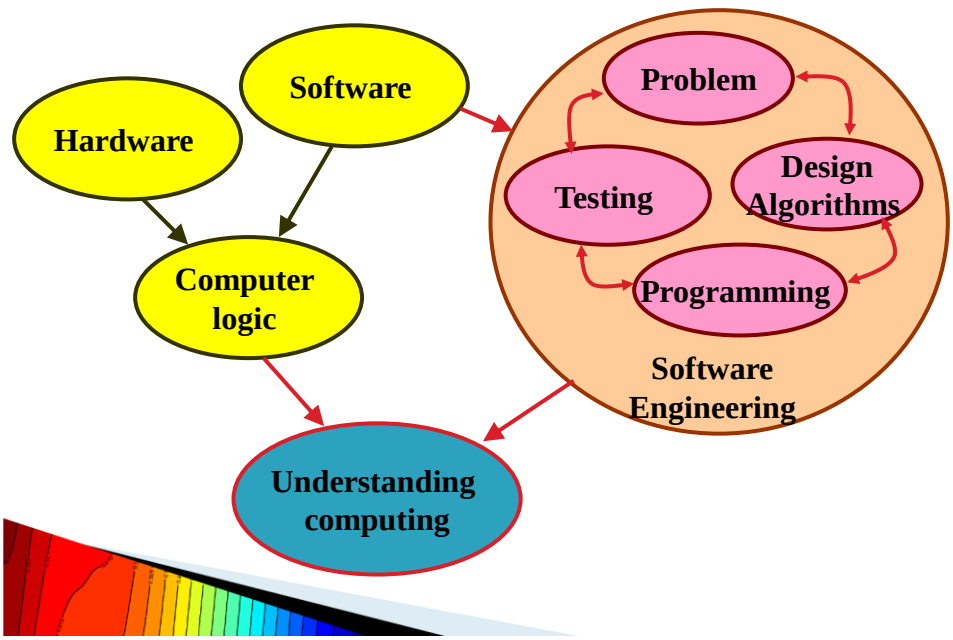
Topic 1: Developing software



Solving problems with computing

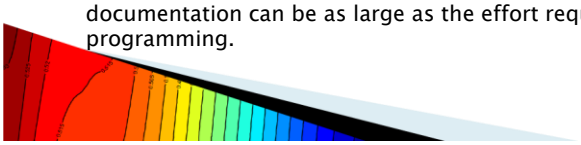


Some Computing Concepts



Principles of Software Engineering

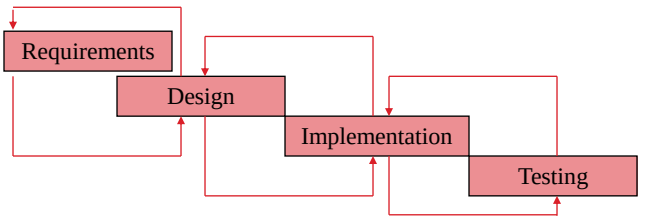
- Engineers from all disciplines are often heavily involved in the development of large and complex software systems.
  - Flight simulators, chemical plant process control, electrical circuit simulators,…
- There are huge problems associated with the development of large software systems.
  - The problems encountered are similar to those found in any large engineering project:
    - Product design, human resources, management, cost control, selection of tools, quality control…
- By software we include:
  - programs,
  - internal documentation (comments in the code),
  - external documentation (software report, users manual and help files).
  - In a large software system, the documentation is absolutely essential to the maintenance of the software and directly affects its longevity.
  - In a large software system, the effort required for the preparation of the documentation can be as large as the effort required for the software design and programming.



The Software Life Cycle

- It is generally agreed upon that the following 5 stages are present in the "life cycle" of a software system development.
  1. Requirements analysis and definition.
  - The services to be provided by the system and the constraints are established in consultation with the users/ client(s).
  2. System and software design.
  - Basically, software design presents the tasks to be performed by the software system that can be easily coded into a set of programs and/or sub-programs (i.e. C functions).
  - Test plans are also developed at this stage.
  3. Software implementation and unit testing.
  - At this stage, the software design step 2 is coded. Each program and sub-program is tested individually.
  4. System integration and testing.
  - At this stage, the programs and sub-programs are integrated to create the complete software system and the system is tested.
  5. Operation and software maintenance.
  - **This is often the longest stage in the software life cycle.**
  - The system is installed and put into practical use.
  - Software maintenance involves correcting and improving the software

- In reality, the software development cycle does not consist of a linear progression through steps 1 to 4. There is significant information exchange throughout the development cycle and the information can flow forward and backward as represented schematically below:



- During the operation phase of the software life cycle, maintenance activities may include:
  - the definition of new requirements that have been perceived;
  - modifications to the software design (due to advances in hardware for example);
  - a re-coding of certain or all program and sub-programs due to advances in programming languages (say migrate from COBOL to C++) or algorithms;
  - additional testing and verification.
  - The maintainability of software is obviously very important since it will directly affect the software's longevity.
  - Improving the maintainability of a software product is a goal of software engineering.

- Shown below are the relative costs for each phase of the software life cycle (1975):

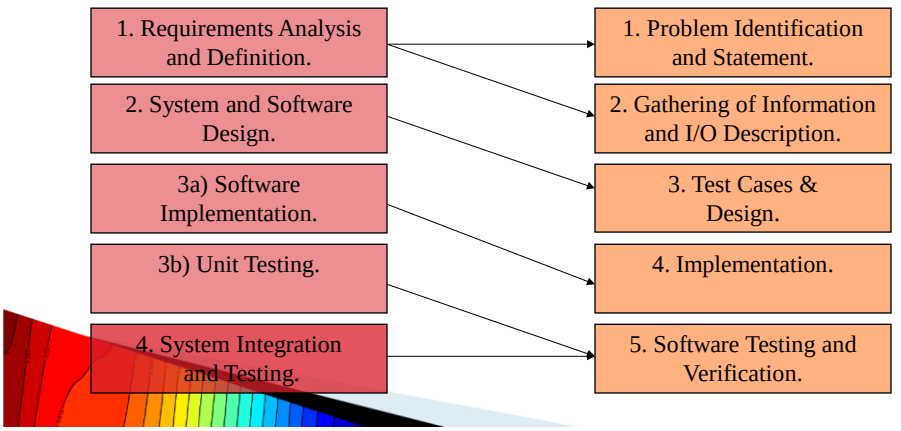
System Type	Phase Costs (%)		
	1 and 2	3	4
Command/Control Systems	46	20	34
Space borne Systems	34	20	46
Operating Systems	33	17	50
Scientific Systems	44	26	30
Business Systems	44	28	28

  - Regardless of the software system constructed, implementation costs (step 3 – the coding) are always the lowest.
  - Much variation exists in the cost of software maintenance but it is common for maintenance costs to exceed total development costs by factors of 2 to 4 over the software lifetime.

From The Software Life Cycle to Engineering Problem-Solving Method.

- Engineering problem-solving using computers usually follows a prescribed methodology that incorporates sound engineering principles, such as the methodology that we have adopted in this course. The software life cycle and our problem solving methodology interact as shown below.
  - Software Life Cycle

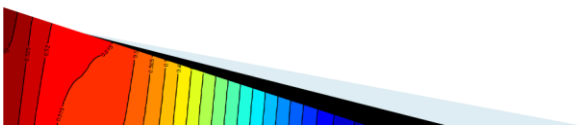
Problem Solving Methodology



Software Design: Top-Down Approach

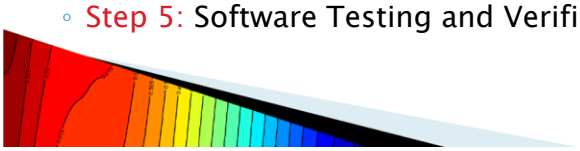
- The second phase in the software life cycle and the fourth step in our problem solving strategy consist of formulating algorithms and designing software. This is commonly achieved by applying a top-down approach with successive stepwise refinement.
- The top-down approach with stepwise refinement is an iterative method that yields in the end a detailed description of the programs and sub-programs to be implemented.
- The approach is termed top-down because:
  - We begin at the highest level by describing what the general tasks (e.g. what the main function does).
  - The general tasks are supported by other tasks (e.g. the functions called by main), without defining details of the smallest tasks.
  - Subsequent iterations refine previous iterations by specifying in more and more detail (e.g. functions called by other functions).
  - At each iteration, it is possible to revise tasks already defined.
  - We proceed downwards into the smallest details (simplest functions).

Topic 2: Problem Solving Method



Our Engineering Problem-Solving Method

- We shall apply in this course a formal problem-solving in the course project.
  - The method is designed to get us thinking about various aspects of a problem before attempting to formulate a software solution.
  - The format of the software report for the project that must be prepared follows closely the method.
- Our method is a five-step approach to problem solving.
  - Step 1: Problem Identification and Statement.
  - Step 2: Gathering of Information and Input/Output Description.
  - Step 3: Test Cases (Hand-Solved Examples) and Design.
  - Step 4: Implementation.
  - Step 5: Software Testing and Verification.

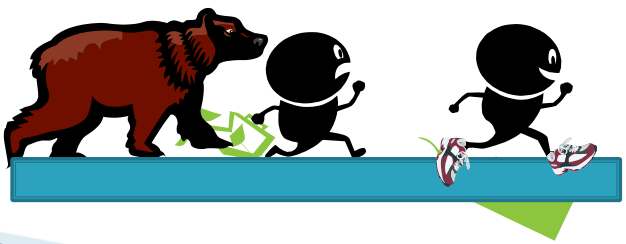


Step 1: Problem Identification and Statement

- Gets us thinking about what is the real problem that must be solved and forces us to make an unambiguous problem statement.
- The identification of the real problem to be solved is often a non-trivial task.
- Beware of the Wrong Problem
  - The bear
  - Long lineups

The Bear

- Mike and Jack are hiking in the woods when suddenly a bear attacks them, forcing them to climb a tree. As the bear starts to climb the tree too, it becomes obvious that Mike and Jack soon need to jump down and make a run for it. Jack starts putting on his running shoes:
- Mike: You are an idiot, Jack! There is no way those running shoes are going to help you outrun the bear!
- Jack: Actually, Mike, I don't need to outrun the bear, I just need to outrun you ;)



Long Lineups

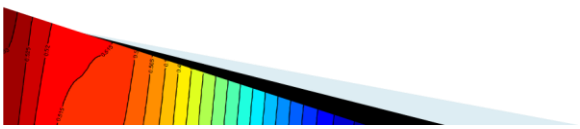
- An amusement park receives lots of complaints that their lineups are too long.
- They increase their ride infrastructure and personnel to speed up the queues, but complaints are still coming in.
- One of their engineers recommends the following solution, which works and significantly reduces the number of complaints:
  - At specific distances in the line, put a sign saying how many minutes are left to the front of the line. But instead of putting the right value, put a value that is 1.5 times the expected; i.e., manipulate people's expectation by under-promising and over-delivering.
- Similar problem and solution for long traffic lights.

Step 2: Gathering of Information and Input/Output Description

- What are the constraints?
- What are the physical laws?
- What are the mathematical equations?
- At this point we gather all relevant information required to solve the problem: constraints, theorems, laws, rules, equations, data…
- Forces us to clearly identify the input information to the problem and the expected results or output.

Step 3: Test Cases (Hand-Solved Examples) and Design

- There may be many possible solutions – pick one.
- At this point we attempt to hand-solve the problem to ensure our comprehension of the applicable information gathered in step 2.
- We also prepare the test cases if possible (it usually is in engineering).
- Once we have ensured our comprehension of the relevant information, we can proceed with the software design.
- A top-down approach is used where tasks are separated into functions.
- The algorithm is a step-by-step outline of each function in English.
  - a list of steps to follow to accomplish the function's task in which order is important.
  - The algorithm must summarize every logical step that is required in the function.
- Steps 1 through 3 including the algorithm must be clear enough in our report for another person to be able to do the coding.



Rise and Shine Algorithm

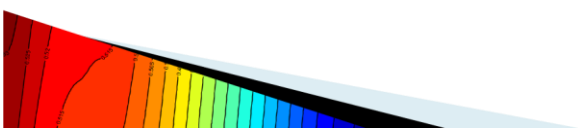
Well Ordered

- Get out of bed
- Take a shower
- Get dressed
- Eat breakfast
- Drive to work

Badly Ordered

- Get out of bed
- Get dressed
- Take a shower
- Eat breakfast
- Drive to work

Arrive at work wet!



Design and Algorithm of a Function

- Consider the design of a function that finds the minimum value in an array of real values

`double findMin(int n,  double array[])`

Parameters

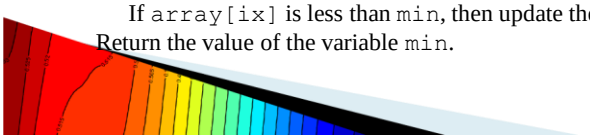
n – the number of elements in the referenced array
array – the referenced array of real values

Return value

The minimum value found in the array

Logic/Algorithm

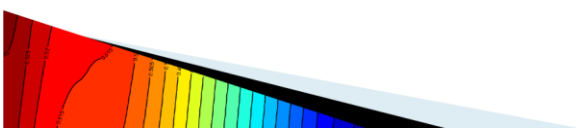
This function scans the array using a determinant loop to find the minimum value in the array.
Use an index, `ix`, as the counter for the loop and to index the elements of the array.
Initialise a variable `min`, to the first element of the array `array[0]`.
Initialise the index `ix` to 1 and loop over all values up to `n`-1. In each iteration of the loop:
Compare the value of the array element `array[ix]` to the value of `min`.
If `array[ix]` is less than `min`, then update the value of `array[ix]` with `min`.
Return the value of the variable `min`.



19

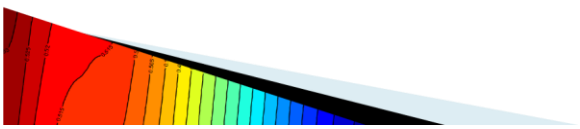
Solution Generation Techniques

- Brainstorming
  - Experience helps
- Inversion
  - Working from the bottom to the top.
- Searching the web!
  - Many solutions already exist and can be found if you search for it.
    - Careful: Are they reliable?
    - Not everything on the web is correct.
    - Understand the solution you are applying.



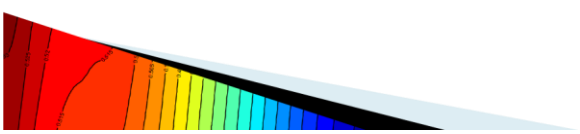
Step 4: Implementation

- Build the machine (hardware).
- Write the program (software).
  - Once the design has been completed on paper, we are ready to proceed with the software implementation in C.
  - Note that it is only in step 4 that we finally touch a computer!



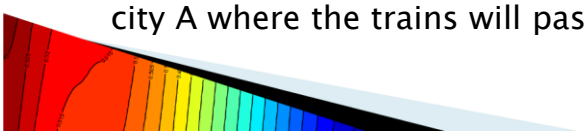
Step 5: Software Testing and Verification

- We cannot claim that our software is functional until it has successfully passed a battery of test cases
  - Check that the software (solution) works in many different situations.
- The selection of appropriate test cases is extremely important in software verification
  - Try different inputs.
  - Test with the wrong inputs.
  - Don't forget the limiting cases
- The idea is to try a number of cases to test all aspects of the developed software



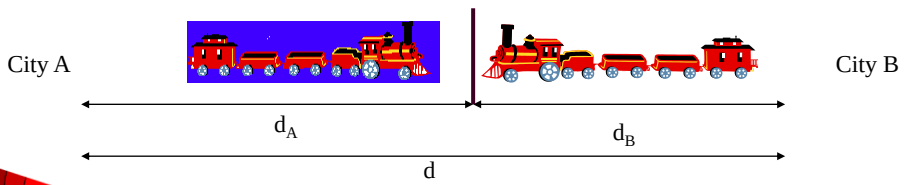
Problem Solving Method – An Example

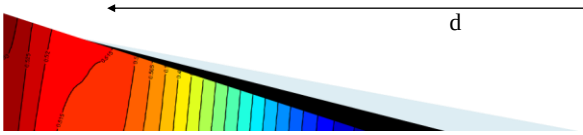
- A train leaves city A travelling towards city B at some given speed (km/h). At the same time a train leaves city B going to city A at some given speed (km/h). Write a program that will compute the distance from city A where the trains will pass each other.
- Step 1: Problem Identification and Statement.
  - In this case the problem is rather simple and unambiguously defined so step 1 is simply a re-statement of the problem described:
    - A train leaves city A travelling towards city B at some given speed (km/h). At the same time a train leaves city B going to city A at some given speed (km/h). Write a program that will compute the distance from city A where the trains will pass each other.



- Step 2: Gathering of Information and Input/Output Description.
  - Fetch any missing information and summarize:
    - From basic physics: distance = velocity * time
    - Distance from city A to city B: d km
    - Velocity of train from city A to city B: v_A km/h
    - Velocity of train from city B to city A: v_B km/h
    - Both trains leave their respective stations at the same time
    - For the train that leaves city A: $d_A = v_A * t$
    - For the train that leaves city B: $d_B = v_B * t$
    - The total distance from city A to city B is: $d = d_A + d_B$

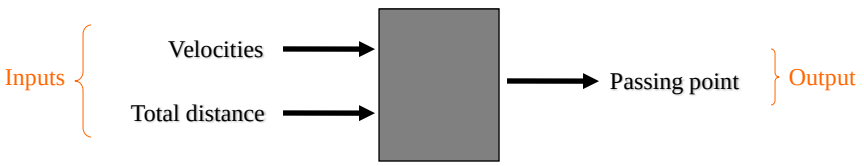
This information was missing!

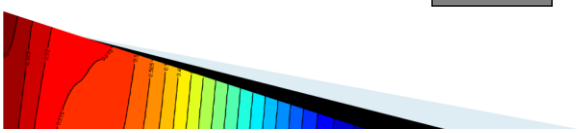




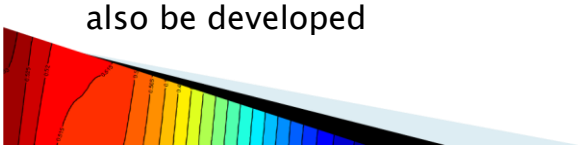
- The trains pass each other after travelling for the same time t (because they have the same start times) so let's find the time when they will meet:
  - $d = d_A + d_B$
  - $d = v_A * t + v_B * t$
  - $t = d/(v_A + v_B)$
- So the distance from city A where they will meet is:
  - $d_A = v_A * t$
  - $d_A = (v_A * d)/(v_A + v_B)$

◦ Input/Output Description:





- Step 3: Test cases and Design.
  - Test Case 1
    - Consider the case where city A is Montreal and city B is Toronto.
    - Input consists of the following:
      - $d = 550$ km
      - $v_A = 100$ km/h
      - $v_B = 150$ km/h
    - Carrying out the computation yields the output.
      - $d_A = (100 * 550)/(100 + 150) = 220$km
  - Other test cases can also be developed (often at Step 5)
  - Test cases for limit values and invalid values should also be developed



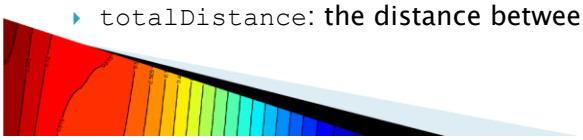
Design

The software shall use a structure to hold the names of the cities and the input values provided by the user. In addition to the main function, three other functions are used, one to obtain the values from the user, one to calculate the passing point from City A with the given values, and one to display the results.

Data Structures

The data structure INPUT is defined to contain the following members:

- `cityA`, `cityB`: these are character arrays used for storing the names of the cities
- `speedA`, `speedB`: the speed of train leaving City A and the train leaving City B respectively
- `totalDistance`: the distance between the 2 cities



27

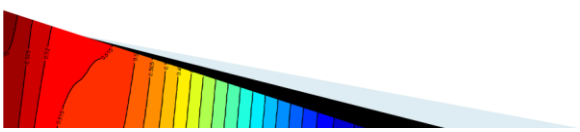
`void main(void)`

Logic/Algorithm

The main function contains the logic for the overall control of the software. It contains a loop to allow the user to repeat calculations. Use a post-test loops that asks the user to terminate the software after each computation. The steps of the loop are:

- Call `getInput` to fill in the input values in the structure variable `input` (type `INPUT`).
- Calls `calculatePass` to compute the passing distance from City A and stores in variable `dCityA`.
- Calls `displayResults` to show user the results.
- Ask user to quit the program and store answer in char variable `answer`.
- Repeat loop if the answer is not 'y' (yes).

- At the end of the program, print the message "Program terminated"



28

`void getInput(INPUT *iPtr)`

Parameters:

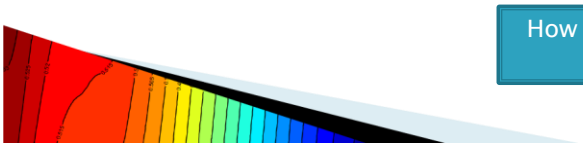
`iPtr`: pointer to a structure variable of type INPUT used for storing the input values obtained from the user.

Logic/Algorithm:

Obtains the values for each member of the referenced structure variable:

- Prompt the user for the name of city A and store answer in member `cityA`.
- Prompt the user for the name of city B and store answer in member `cityB`.
- Prompt the user for the speed of train A leaving from city A and store answer in member `speedA`.
- Prompt the user for the speed of train B leaving from city B and store answer in member `speedB`.
- Prompt the user for the distance between the cities and store the answer in member `totalDistance`

How can this design be improved?



29

`double calculatePass(INPUT *iPtr)`

Parameters:

`iPtr`: pointer to a structure variable of type INPUT that contains the input from the user.

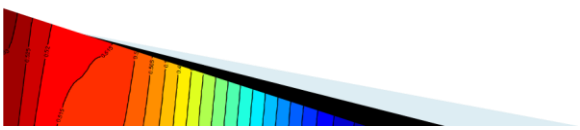
Return Value

The passing distance from City A.

Logic/Algorithm:

Computation of the passing distance is don by applying the formula $d_A = (v_A * d)/(v_A + v_B)$ applied to the input values in the structure variable referenced by iPtr to compute the passing distance and store in the variable passingDistance.

Return the value of passingDistance.



30

void displayResults(INPUT *iPtr, double pDist)

Parameters:

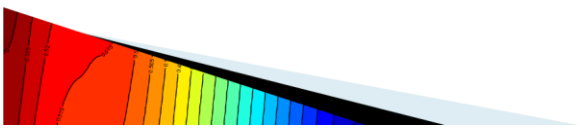
`iPtr`: pointer to a structure variable of type INPUT that contains the input from the user.

`pDist`: Passing distance

Logic/Algorithm:

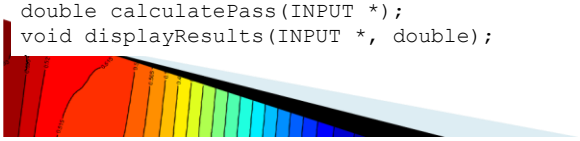
- Displays the results with the messages (shown for given test case):

```
Train A leaves from Montreal at speed 100.00 km/hr
Train B leaves from Toronto at speed 150.00 km/hr
Distance between Montreal and Toronto is 550.00 km
Passing distance from Montreal is 220.00 km
```

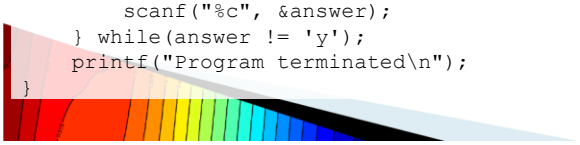


Step 4: Implementation.

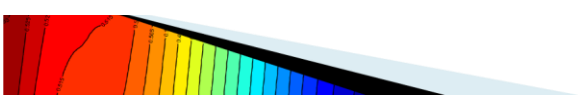
```
/*-------------------------------------------------------------------
File: train.c
Author:Gilbert Arbez, Fall 2016
Description: Computes passing distance of trains leaving from two
different
             cities.
-------------------------------------------------------------------*/
#include <stdio.h>
#define STR_LEN 100
typedef struct
{
    char cityA[STR_LEN];  // Name of City A
    char cityB[STR_LEN];  // Name of City B
    double speedA;        // Speed of train leaving from city A
    double speedB;        // Speed of train leaving from city B
    double totalDistance; // distance between the cities
} INPUT;
// function prototypes
void getInput(INPUT *);
double calculatePass(INPUT *);
void displayResults(INPUT *, double);
```



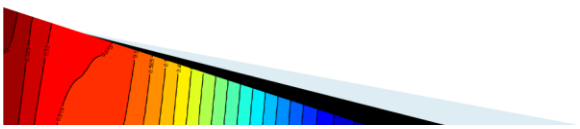
```
/*-------------------------------------------------------------------------
Function: main
Description: Overall control of the problem to compute the passing
             distance of trains travelling in opposite directions.
             Allows the user to repeat the computation many times.
-------------------------------------------------------------------------
*/
void main(void)
{
    // Variable declarations
    INPUT input;    // struct variable for storing user input
    double dCityA;  // Passing distance from cityA
    char answer;    // to get answer from user to quit program
    // Loop to repat computations
    do
    {
        getInput(&input);  // Get input
        dCityA = calculatePass(&input); // COmpute passing distance
        displayResults(&input, dCityA);  // Display results
        // Ask to quit the program
        fflush(stdin);
        printf("Do you wish to quit (y/n)? ");
        scanf("%c", &answer);
    } while(answer != 'y');
    printf("Program terminated\n");
}
```



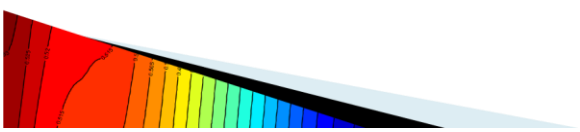
```
/*-------------------------------------------------------------------------
Function: getInput
Parameters:
     iPtr: pointer to a structure variable of type INPUT used for
           storing the input values obtained from the user.
Description: Gets city names, train speeds and distance between the
             cities from the user.
-------------------------------------------------------------------------*/
void getInput(INPUT *iPtr)
{
    printf("Please give the name of city A: ");
    scanf("%s", iPtr->cityA);
    printf("Please give the name of city B: ");
    scanf("%s", iPtr->cityB);
    printf("Please give the speed of the train leaving %s: ",
            iPtr->cityA);
    scanf("%lf", &iPtr->speedA);
    printf("Please give the speed of the train leaving %s: ",
            iPtr->cityB);
    scanf("%lf", &iPtr->speedB);
    printf("Please give the distance between %s and %s: ",
            iPtr->cityA, iPtr->cityB);
    scanf("%lf", &iPtr->totalDistance);
}
```



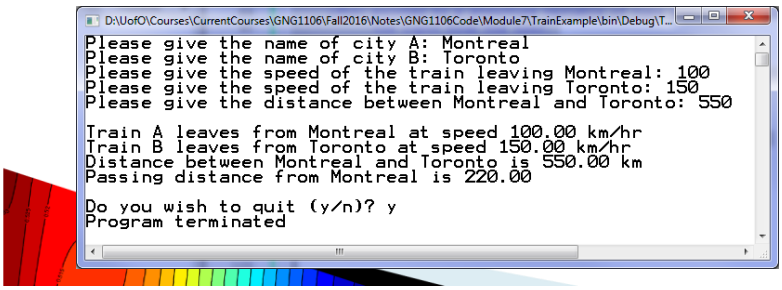
```
/*-------------------------------------------------------------------------
Function: calculatePass
Parameters:
     iPtr: pointer to a structure variable of type INPUT that contains
           the input from the user.
Return Value:
    The passing distance from City A.
Description: Compute the passing distance of the train from City A.
-------------------------------------------------------------------------*/
double calculatePass(INPUT *iPtr)
{
    // Variable declarations
    double passingDistance;
    // Calculation
    passingDistance = (iPtr->speedA * iPtr->totalDistance);
    passingDistance = passingDistance/(iPtr->speedA + iPtr->speedB);

    return(passingDistance);
}
```



```
/*-------------------------------------------------------------------------
Function: displayResults
Parameters:
     iPtr: pointer to a structure variable of type INPUT that contains
           the input from the user.
     pDist: Contains the passing distance from City A
Description: Displays results to the user.
-------------------------------------------------------------------------*/
void displayResults(INPUT *iptr, double pDist)
{
           // Display results
        printf("\nTrain A leaves from %s at speed %.2f km/hr\n",
               iptr->cityA, iptr->speedA);
        printf("Train B leaves from %s at speed %.2f km/hr\n",
               iptr->cityB, iptr->speedB);
        printf("Distance between %s and %s is %.2f km\n",
               iptr->cityA, iptr->cityB, iptr->totalDistance);
        printf("Passing distance from %s is %.2f\n\n",
                iptr->cityA, pDist);
}
```



- Step 5: Software Testing and Verification.
  - Test the solution generated by the software by comparing with the test cases:
    - $d_A = (100 * 550)/(100 + 150) = 220km$
  - For debugging the software, it may be useful to check extra variables if used (i.e.: test more than just the output values) :
    - $d_B = d - d_A = 550 - 220 = 330km$
    - $t = d_A/v_A = 220/100 = 2.2h$
    - $d_B = v_A * t = 150 * 2.2 = 330km$



Next Module