CSE-105: PROGRAMMING FUNDAMENTALS LECTURE 3: PROBLEM SOLVING

COURSE INSTRUCTOR: MD. SHAMSUJJOHA

Problem Solving

- In this lecture we will learn about:
  - Introduction to Problem Solving
  - Software development method (SDM)
    - Specification of needs
    - Problem analysis
    - Design and algorithmic representation
    - Implementation
    - Testing and verification
    - Documentation

The General Form of a Simple Program

- Simple C programs have the form

```
directives

int main(void)
{
   statements
}
```

The General Form of a Simple Program

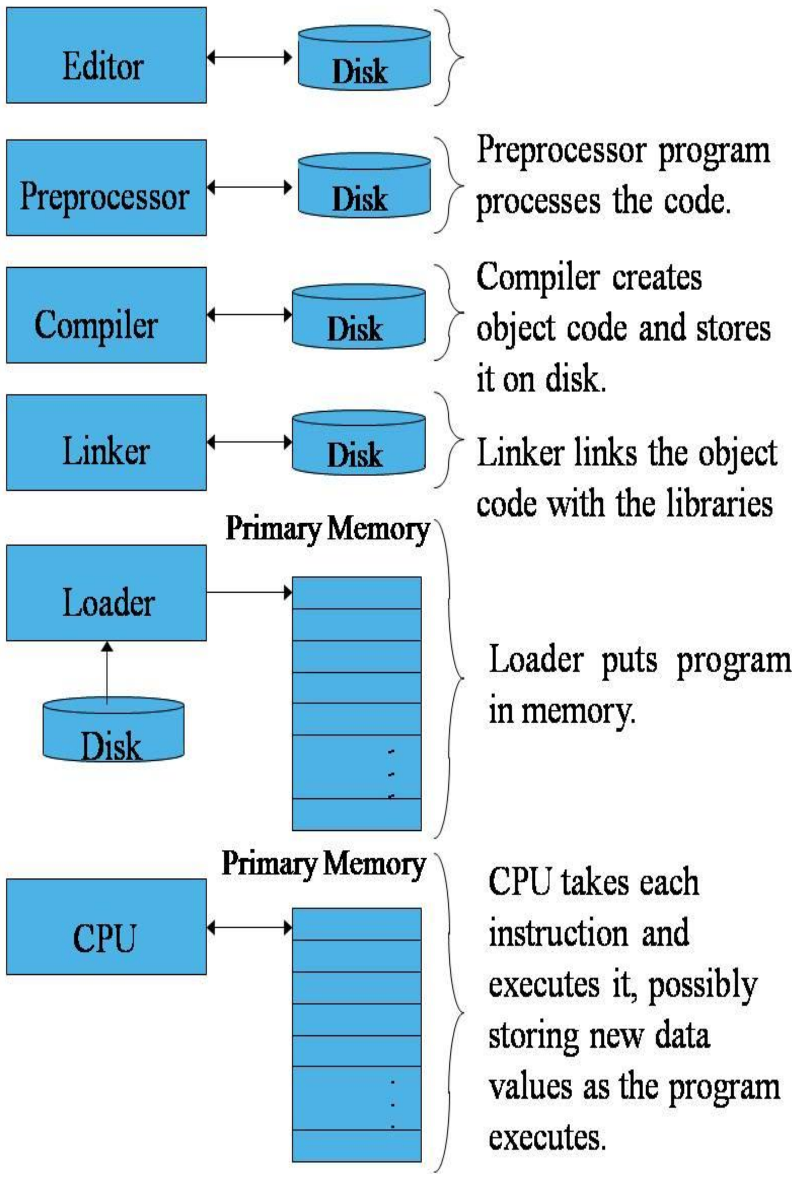
- C uses `{` and `}` in much the same way that some other languages use words like `begin` and `end`, respectively.
- Even the simplest C programs rely on three key language features:
  - Directives
  - Functions
  - Statements

Directives

- Before a C program is compiled, it is first edited by a preprocessor.
- Commands intended for the preprocessor are called directives.
- Example:

```
#include <stdio.h>
```

- `<stdio.h>` is a ***header*** containing information about C's standard I/O library.
- Directives always begin with a **#** character.
- By default, directives are one line long; there's no semicolon or other special marker at the end.



Editor
Disk
Preprocessor
Disk
Preprocessor program processes the code.
Compiler
Disk
Compiler creates object code and stores it on disk.
Linker
Disk
Linker links the object code with the libraries
Primary Memory
Loader
Disk
Loader puts program in memory.
Primary Memory
CPU
CPU takes each instruction and executes it, possibly storing new data values as the program executes.

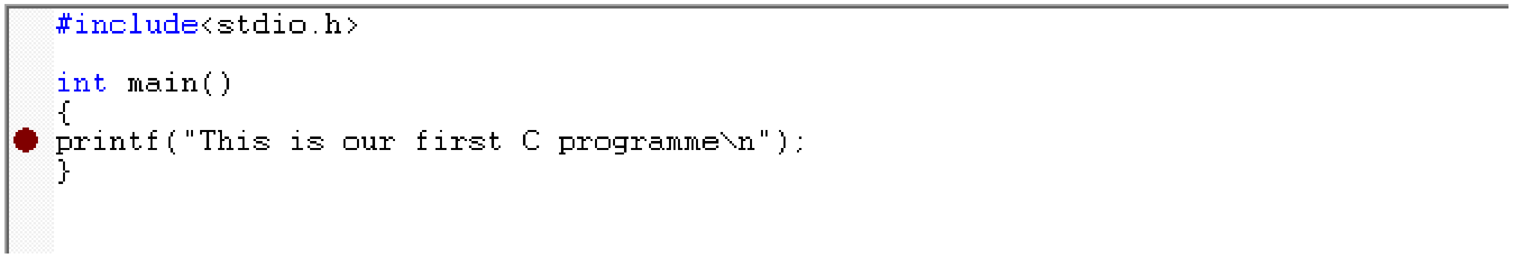
Functions

- A ***function*** is a series of statements that have been grouped together and given a name.
- ***Library functions*** are provided as part of the C implementation.
- A function that computes a value uses a `return` statement to specify what value it “returns”:

```
return x + 1;
```

The `main` Function

- The `main` function is mandatory.
- `main` is special: it gets called automatically when the program is executed.
- `main` returns a status code; the value 0 (Zero) indicates normal program termination.
- If there's no `return` statement at the end of the main function, many compilers will produce a warning message.

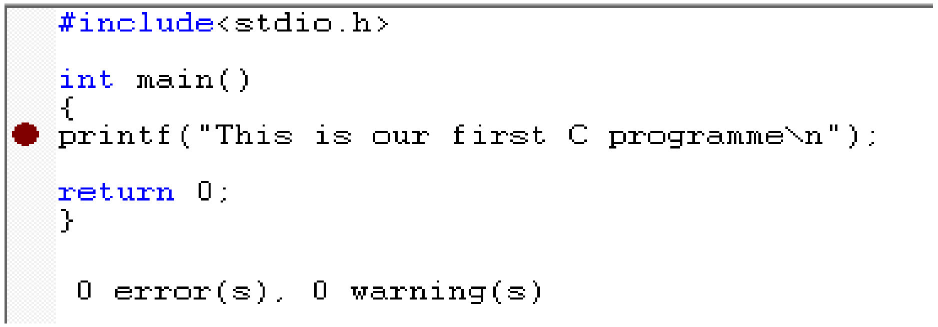


```
#include<stdio.h>

int main()
{
printf("This is our first C programme\n");
}
```

```
warning C4508: 'main' : function should return a value; 'void' return type assumed
```

The **`main`** Function

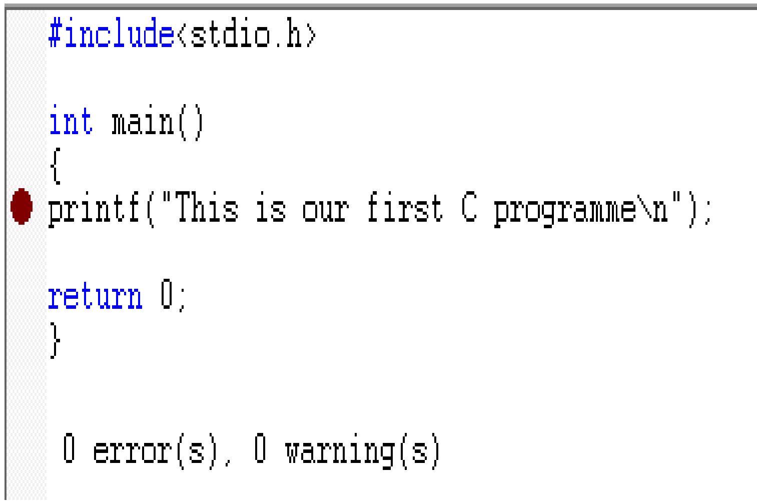
- The `main` function is mandatory.
- `main` is special: it gets called automatically when the program is executed.
- `main` returns a status code; the value 0 (Zero) indicates normal program termination.
- If there's no `return` statement at the end of the `main` function, many compilers will produce a warning message.



```
#include<stdio.h>

int main()
{
printf("This is our first C programme\n");

return 0;
}

 0 error(s), 0 warning(s)
```

Statements

- A ***statement*** is a command to be executed when the program runs.
- `first.c` uses only two kinds of statements. One is the `return` statement; the other is the print statement.
- Asking a function to perform its assigned task is known as ***calling*** the function.
- `first.c` calls `printf` to display a string: "`This is our first C programme`"



```
#include<stdio.h>

int main()
{
printf("This is our first C programme\n");

return 0;
}


 0 error(s), 0 warning(s)
```

Printing Strings

- C requires that each statement end with a semicolon.
- There's one exception: the compound statement.
- Directives are normally one line long, and they don't end with a semicolon.
- When the `printf` function displays a ***string literal***—characters enclosed in double quotation marks—it doesn't show the quotation marks.
- To make `printf` advance one line, include `\n` (the ***new-line character***) in the string to be printed.

Printing Strings

- The statement

```
printf(“Hello World\n");
```

could be replaced by two calls of `printf`:

```
printf(“Hello ");
printf(“World\n");
```

- The new-line character can appear more than once in a string literal:

```
printf(“Hello\nWOrld\n");
```

Variables and Assignment

- Most programs need to a way to store data temporarily during program execution.
- These storage locations are called ***variables*****.**

Types

- Every variable must have a ***type***.
- C has a wide variety of types, including `int` and `float`.
- A variable of type `int` (short for *integer*) can store a whole number such as 0, 1, 392, or –2553.

Types

- A variable of type `float` (short for *floating-point*) can store much larger numbers than an `int` variable.
- Also, a `float` variable can store numbers with digits after the decimal point, like 379.125.
- Drawbacks of `float` variables:
  - Slower arithmetic
  - Approximate nature of `float` values

Declarations

- Variables must be ***declared*** before they are used.
- Variables can be declared one at a time:

```
int height;
float profit;
```

- Alternatively, several can be declared at the same time:

```
int height, length, width, volume;
float profit, loss;
```

Declarations

- When `main` contains declarations, these must precede statements:

```
int main(void)
{
   declarations
   statements
}
```

Assignment

- A variable can be given a value by means of ***assignment:***

  ```
  height = 8;
  ```

  The number `8` is said to be a ***constant.***

- Before a variable is assigned a value—or is used in any other way—it must first be declared.

Assignment

- A constant assigned to a `float` variable usually contains a decimal point:

```
profit = 2150.48;
```

- It's best to append the letter `f` to a floating-point constant if it is assigned to a `float` variable:

```
profit = 2150.48f;
```

- An `int` variable is normally assigned a value of type `int`, and a `float` variable is normally assigned a value of type `float`.
- Mixing types (such as assigning an `int` value to a `float` variable or assigning a `float` value to an `int` variable) is possible but not always safe.

Assignment

- Once a variable has been assigned a value, it can be used to help compute the value of another variable:

```
height = 8;
length = 12;
width = 10;
volume = height * length * width;
  /* volume is now 960 */
```

- The right side of an assignment can be a formula (or ***expression,*** in C terminology) involving constants, variables, and operators.

Printing the Value of a Variable

- `printf` can be used to display the current value of a variable.
- To write the message

  `Height:` *h*

  where *h* is the current value of the `height` variable, we'd use the following call of `printf`:

  ```
  printf("Height: %d\n", height);
  ```
- `%d` is a placeholder indicating where the value of `height` is to be filled in.

Printing the Value of a Variable

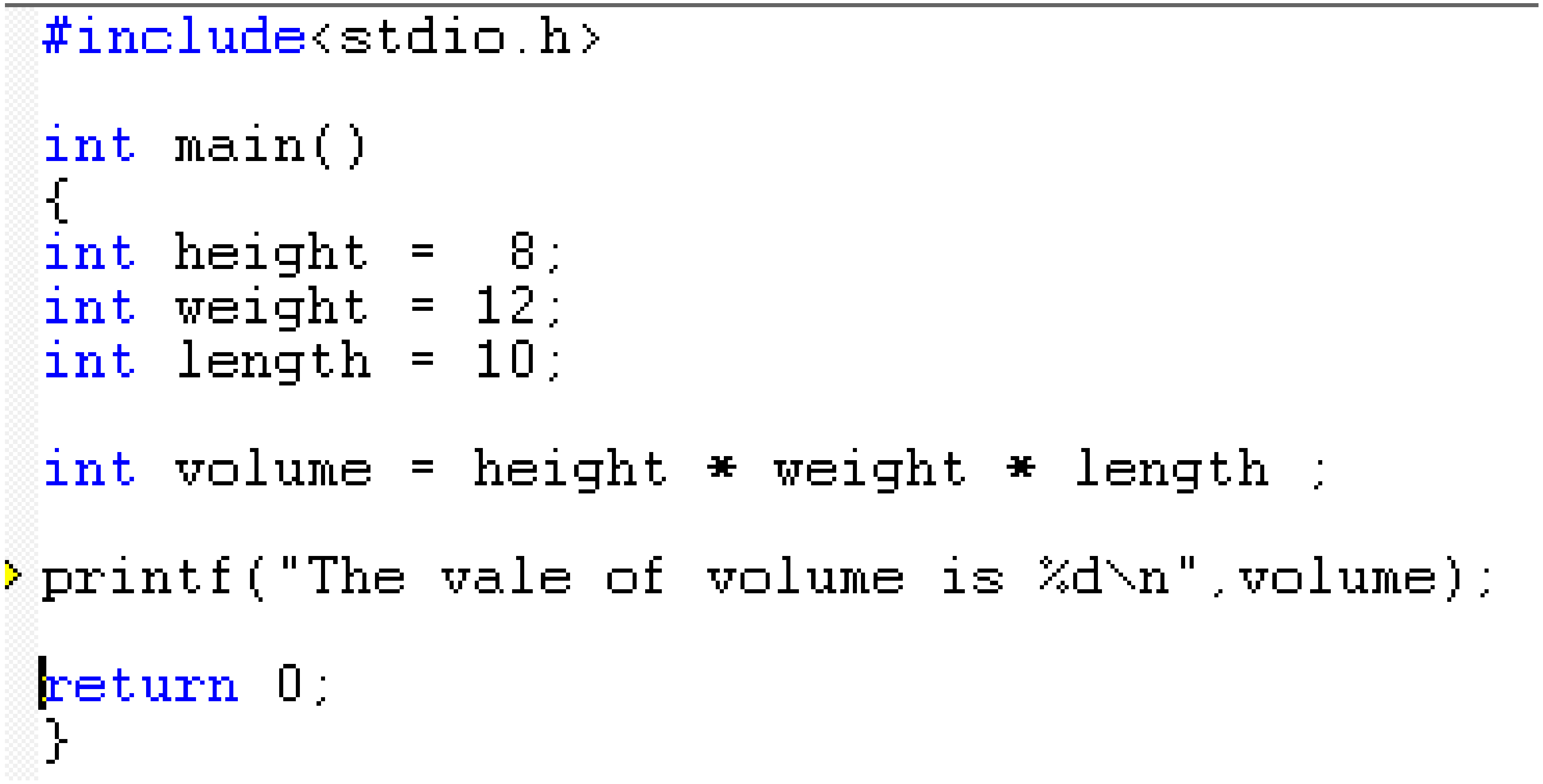
- `%d` works only for `int` variables; to print a `float` variable, use `%f` instead.
- By default, `%f` displays a number with six digits after the decimal point.
- To force `%f` to display *p* digits after the decimal point, put `.`*p* between `%` and `f`.
- To print the line `Profit: $2150.48`

  use the following call of `printf`:

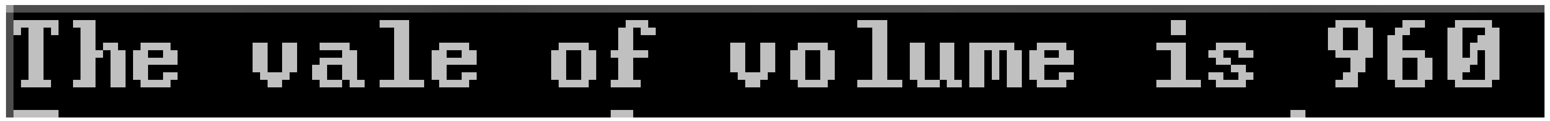
  ```
  printf("Profit: $%.2f\n", profit);
  ```
- There's no limit to the number of variables that can be printed by a single call of `printf`:

  ```
  printf("Height: %d  Length: %d\n", height, length);
  ```

The Code



```
#include<stdio.h>

int main()
{
int height =  8;
int weight = 12;
int length = 10;

int volume = height * weight * length ;

printf("The vale of volume is %d\n",volume);

return 0;
}
```



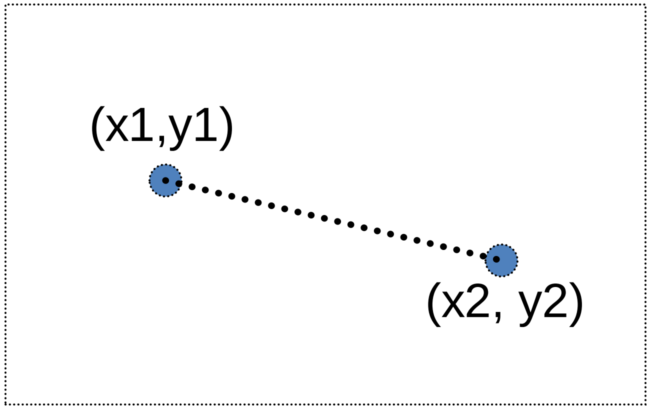
```
The vale of volume is 960
```

Problem Solving Methodology

1. State the problem clearly
2. Describe the input/output information
3. Work the problem by hand, give example
4. Develop a solution (Algorithm Development)

   and Convert it to a program (C program)
5. Test the solution with a variety of data

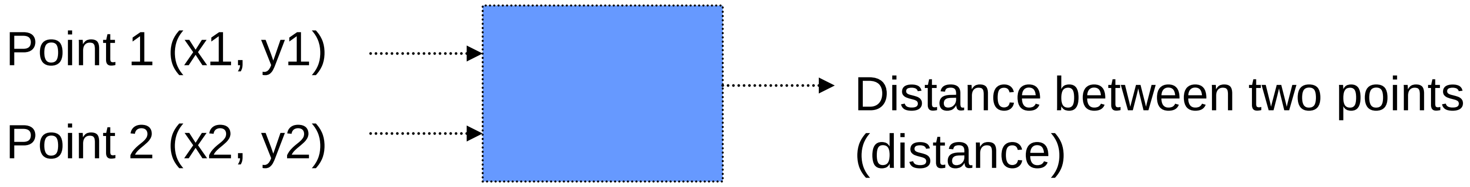
Example 1



1. Problem statement

Compute the straight line distance between two points in a plane

2. Input/output description



Example 1 (Cont …)

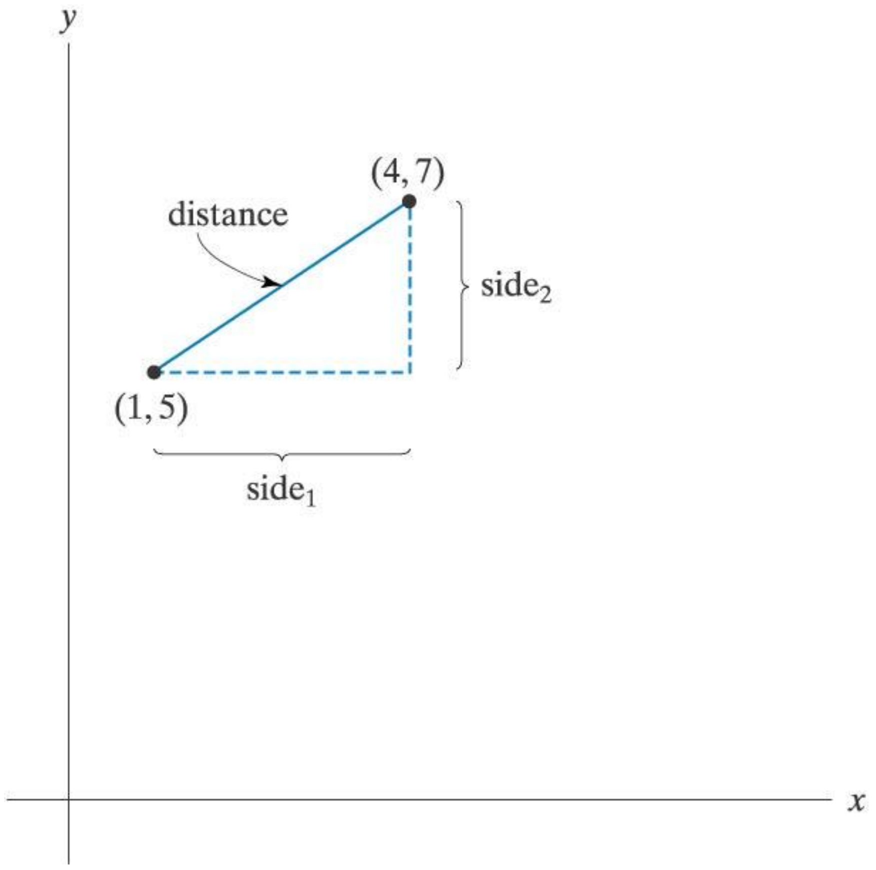
3. Hand example

side1 = 4 - 1 = 3
side2 = 7 - 5 = 2

$$\text{distance} = \sqrt{\text{side1}^2 + \text{side2}^2}$$

$$\text{distance} = \sqrt{3^2 + 2^2}$$

$$\text{distance} = \sqrt{13} = 3.6055$$



Example 1 (Cont …)

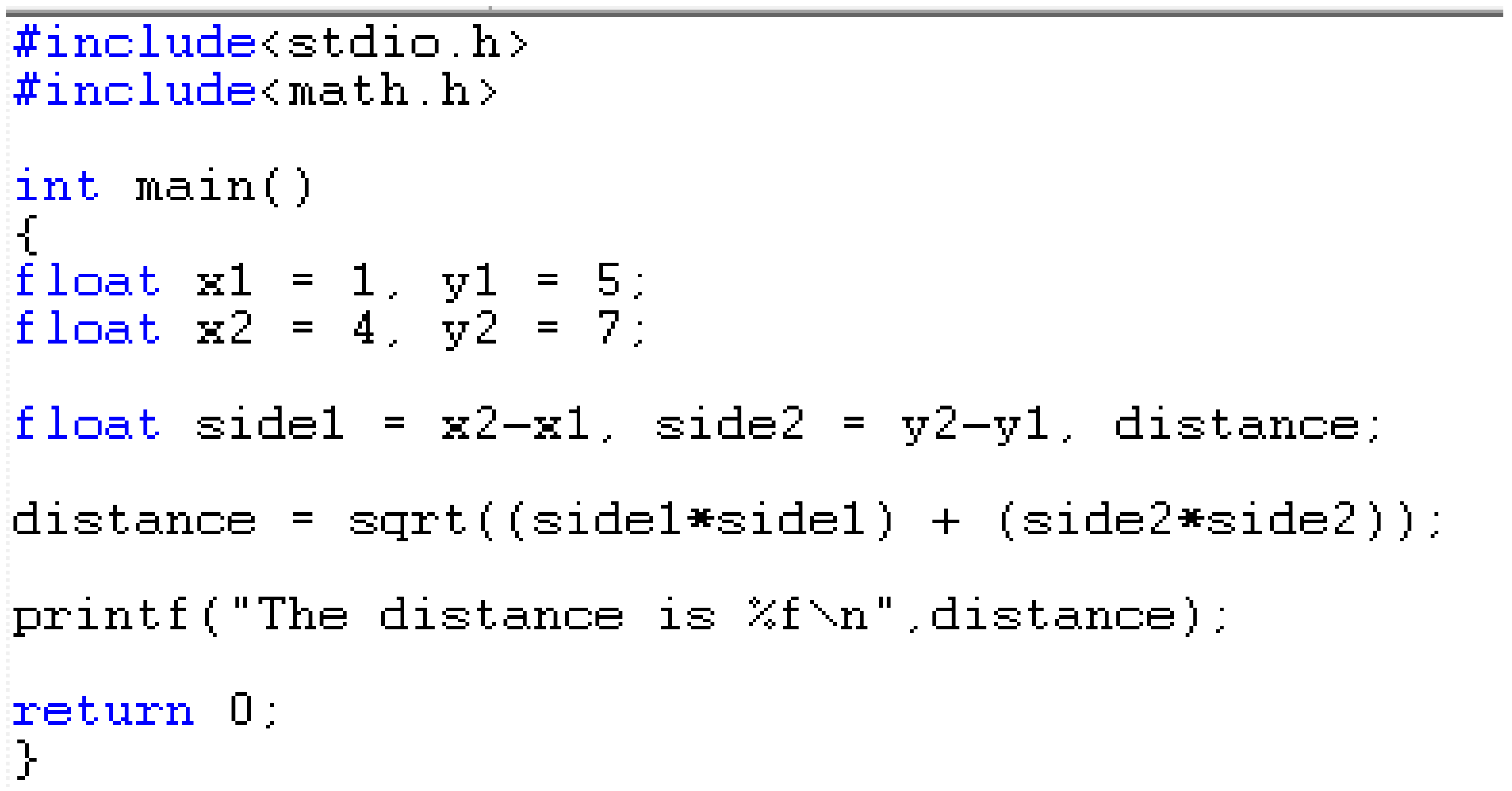
4. Algorithm development and coding

a. Generalize the hand solution and list/outline the necessary operations step-by-step

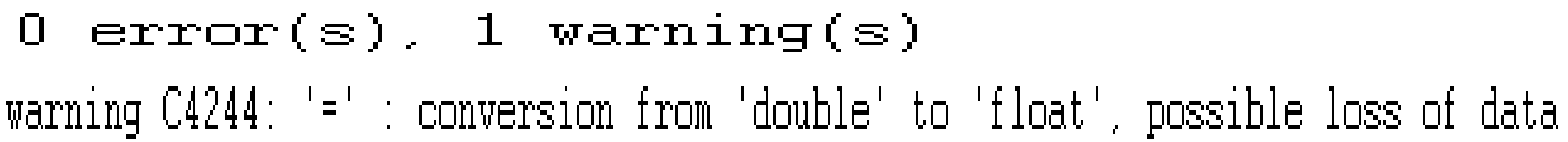
1) Give specific values for point1 (x1, y1) and point2 (x2, y2)
2) Compute side1=x2-x1 and side2=y2-y1
3) Compute $\text{distance} = \sqrt{\text{side1}^2 + \text{side2}^2}$
4) Print distance

b. Convert the above outlined solution to a program using any language you want (see next slide for C imp.)

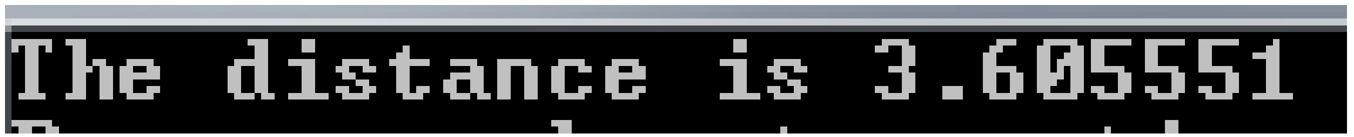
Example 1 (Cont …)



```c
#include<stdio.h>
#include<math.h>

int main()
{
float x1 = 1, y1 = 5;
float x2 = 4, y2 = 7;

float side1 = x2-x1, side2 = y2-y1, distance;

distance = sqrt((side1*side1) + (side2*side2));

printf("The distance is %f\n",distance);

return 0;
}
```



```
0 error(s), 1 warning(s)
warning C4244: '=' : conversion from 'double' to 'float', possible loss of data
```



```
The distance is 3.605551
```

Example 1 (Cont …)

5. Testing

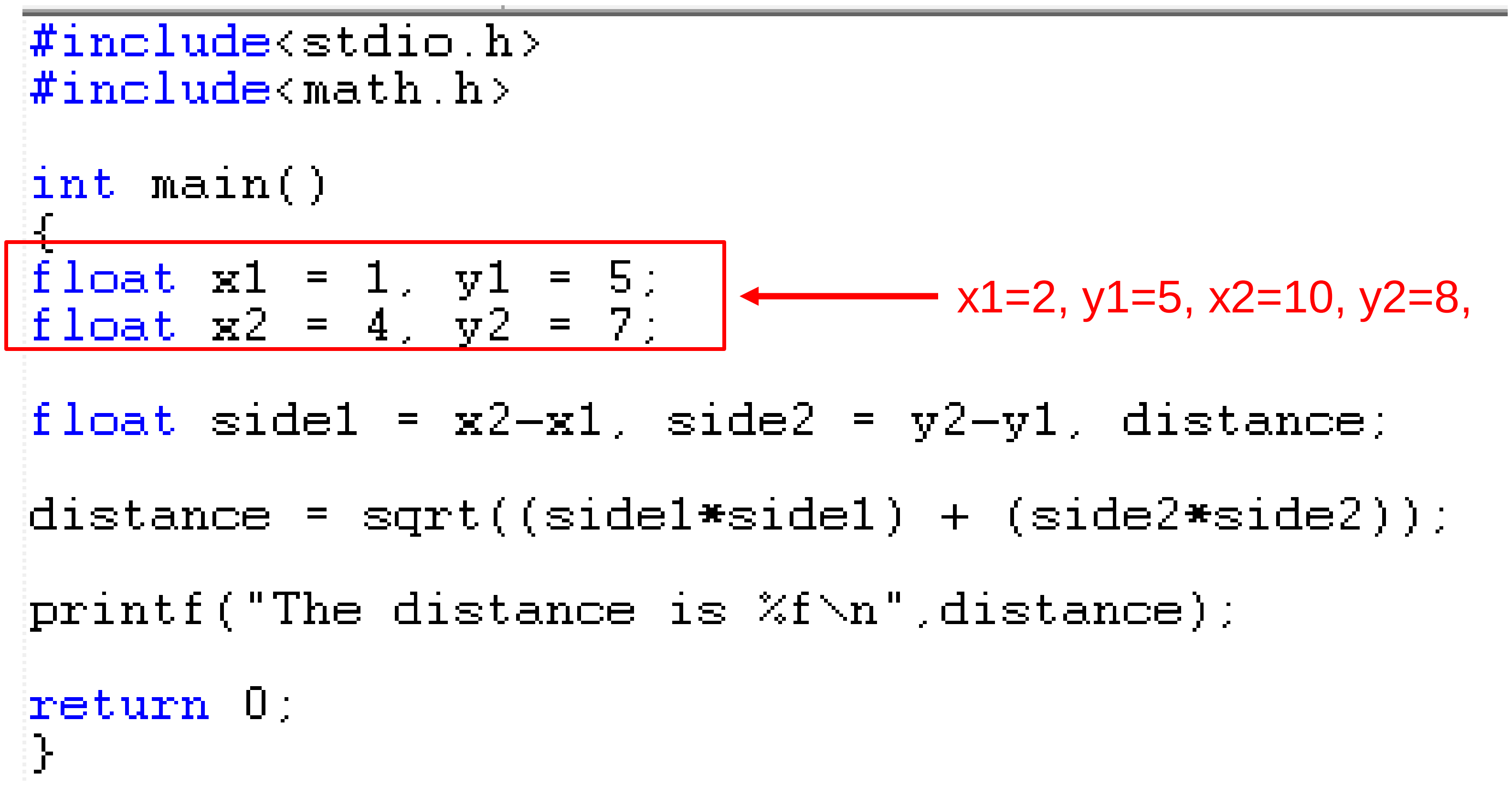
- After compiling your program, run it and see if it gives the correct result.
- Your program should print out

```
      The distance is 3.6055
```

- If not, what will you do?

Example 1 (Cont …)

How will you find the distance between two other points (2,5) and (10,8)?



```
#include<stdio.h>
#include<math.h>

int main()
{
float x1 = 1, y1 = 5;
float x2 = 4, y2 = 7;

float side1 = x2-x1, side2 = y2-y1, distance;

distance = sqrt((side1*side1) + (side2*side2));

printf("The distance is %f\n",distance);

return 0;
}
```

x1=2, y1=5, x2=10, y2=8,

More Problem Solving

- Problem solving is the process of transforming the description of a problem into a solution by using our knowledge of the problem domain and by relying on our ability to select and use appropriate problem-solving strategies, techniques and tools.
- Computers can be used to help us **solving problems**

Software Development Method (SDM)

1. Specification of needs
2. Problem analysis
3. Design and algorithmic representation
4. Implementation
5. Testing and verification
6. Documentation

Specification of Needs

- To understand exactly:
  - what the problem is
  - what is needed to be solve
  - what the solution should provide
  - if there are constraints and special conditions.

Problem Analysis

- In the analysis phase, we should identify the following:
  - Inputs to the problem, their form and the input media to be used
  - Outputs expected from the problem, their form and the output media to be used
  - Special constraints or conditions (if any)
  - Formulas or equations to be used

Design and Algorithmic Representation

- An algorithm is a sequence of a finite number of steps arranged in a specific logical order which, when executed, produces the solution for a problem.
- An algorithm must satisfy these requirements:
  - It may have **input(s)**
  - It must have an **output**
  - It should not be **ambiguous** (there should not be different interpretations to it)
    - Every step in algorithm must be clear as what it is supposed to do

Design and Algorithmic Representation cont..

- It must be **general** (it can be used for different inputs)
- It must be **correct** and it must solve the problem for which it is designed
- It must execute and terminate in a **finite** amount of time
- It must be **efficient** enough so that it can solve the intended problem using the resource currently available on the computer

- An algorithm can be represented using <u>pseudocodes</u> or <u>flowcharts</u>.

Control Structure

- In order to tackle a problem, we need
  - a correct algorithm
  - to apply the algorithm at the 'good' moment
  - to decide which algorithm to apply (sometimes there are more than one, depending on conditions)
  - to know if a certain operation must be repeated

  →In short: we need a suitable ***Control Structure***

- In 1966, two researchers, C. Bohn and G. Jacopini, demonstrated that **any algorithm** can be described using only **3 control structures**: ***sequence***, ***selection*** and ***repetition***.

Pseudocodes

- A pseudocode is a semiformal, English-like language with limited vocabulary that can be used to design and describe algorithms.
- Criteria of a good pseudocode:
  - Easy to understand, precise and clear
  - Gives the correct solution in all cases
  - Eventually ends

Pseudocodes: The **Sequence** control structure

- A series of steps or statements that are executed in the order they are written in an algorithm.
- The beginning and end of a block of statements can be optionally marked with the keywords *begin* and *end*.
- Example 1:

```
Begin
    Read the birth date from the user.
    Calculate the difference between the
    birth date and today's date.
    Print the user age.
End
```

Pseudocodes: The **Selection** control structure

- Defines two courses of action depending on the outcome of a condition. A condition is an expression that is, when computed, evaluated to either true or false.
- The keyword used are *if* and *else*.
- Format:

```
if condition
    then-part
else
    else-part
end_if
```

Example 2:

```
if age is greater than 55
        print "Retire"
else
        print "Work Work Work"
end_if
```

Pseudocodes: The **Selection** control structure

- Sometimes in certain situation, we may omit the else-part.

```
if number is odd number
  print “This is an odd number”
end_if
```

Example 3

Pseudocodes: The **Selection** control structure

- Nested selection structure: basic selection structure that contains other if/else structure in its then-part or else-part.

```
if number is equal to 1
    print "One"
else if number is equal to 2
    print "Two"
else if number is equal to 3
    print "Three"
else
    print "Other"
end_if
```

Example 4

Pseudocodes: The **Repetition** control structure

- Specifies a block of one or more statements that are repeatedly executed until a condition is satisfied.
- The keyword used is *while*.
- Format:

```
while condition
   loop-body
end_while
```

Pseudocodes: The **Repetition** control structure

- Example 5: Summing up 1 to 10

```
set cumulative sum to 0
set current number to 1
while current number is less or equal to 10
    add the cumulative sum to current number
    add 1 to current number
end_while
print the value of cumulative sum
```

Pseudocodes: The **Repetition** control structure

- Subsequently, we can write the previous pseudocodes (example 5) with something like this.
- Example 6: Summing up 10 numbers

```
cumulative sum = 0
current number = 1
while current number is less or equal to 10
     cumulative sum = cumulative sum + current number
     current number = current number + 1
end_while
print the value of cumulative sum
```

- Note that in this algorithm, we are using both the *sequence* and *repetition* control structure

Pseudocodes: The **Repetition** control structure

- Example 7:

```
Begin
   number of users giving his birth date = 0
   while number of users giving his birth date < 10
     begin
        Read the birth date from the user.
        Calculate the difference between the birth      date and
        today's date.
        Print the user age.
        if the age is greater than 55
           print "retired"
        else
           print "keep working"
        end_if
           number of user giving his birth date + 1
        end
   end_while
End
```

Pseudocodes: The **Repetition** control structure

- Example 8:

```
while user still wants to play
   begin
         Select either to play on network or play against computer
          if play on network
                 create connection to remote machine
                 play game with connected computer
          else
              select mission
              play game locally
          end_if
         Ask user whether he/she still wants to play
    end
end_while
```

Pseudocodes: The **Repetition** control structure

- Example 9:

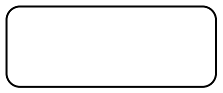
```
while user still wants to play
begin
Select either to play on network or play against computer
if play on network
create connection to remote machine
play game with connected computer
Else
select mission
play game locally
end_if
Ask user whether he/she still wants to play
end
end_while
```

- For readability, always use proper *indentation!!!*

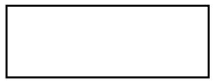
Flowcharts

- Flowcharts is a graph used to depict or show a step by step solution using **symbols** which represent a task.
- The symbols used consist of geometrical shapes that are connected by **flow lines**.
- It is an alternative to pseudocoding; whereas a pseudocode description is verbal, a flowchart is graphical in nature.

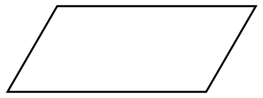
Flowchart Symbols



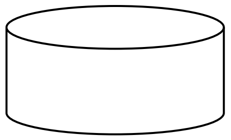
Terminal symbol - indicates the beginning and end points of an algorithm.



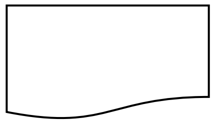
Process symbol - shows an instruction other than input, output or selection.



Input-output symbol - shows an input or an output operation.

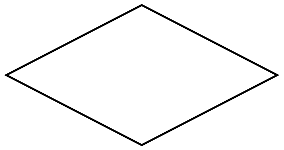


Disk storage I/O symbol - indicates input from or output to disk storage.

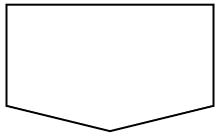


Printer output symbol - shows hardcopy printer output.

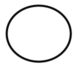
Flowchart Symbols cont…



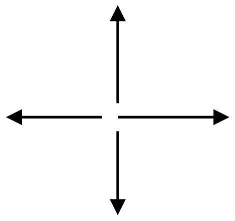
Selection symbol - shows a selection process for two-way selection.



Off-page connector - provides continuation of a logical path on another page.

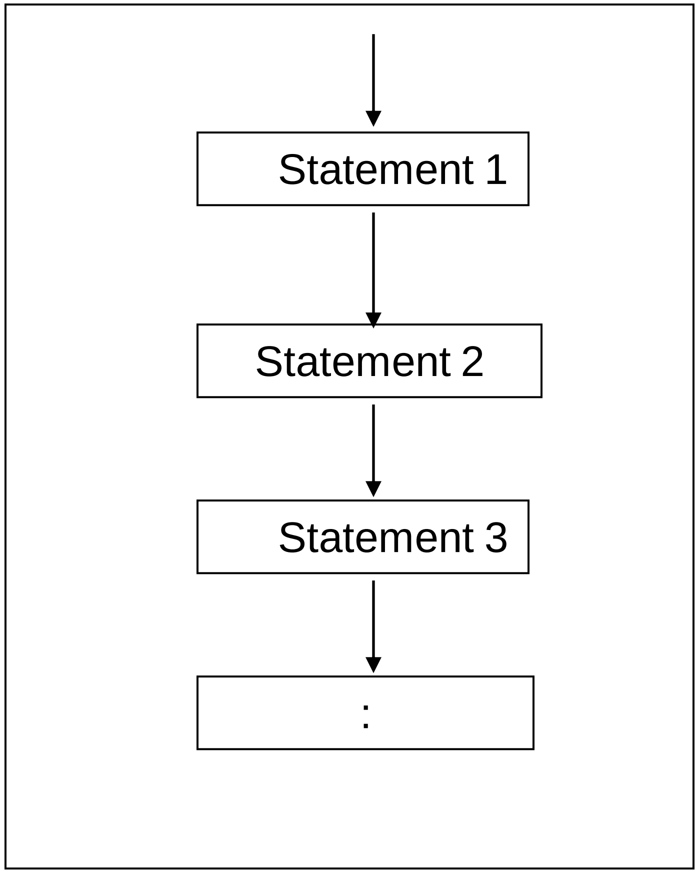


On-page connector - provides continuation of logical path at another point in the same page.

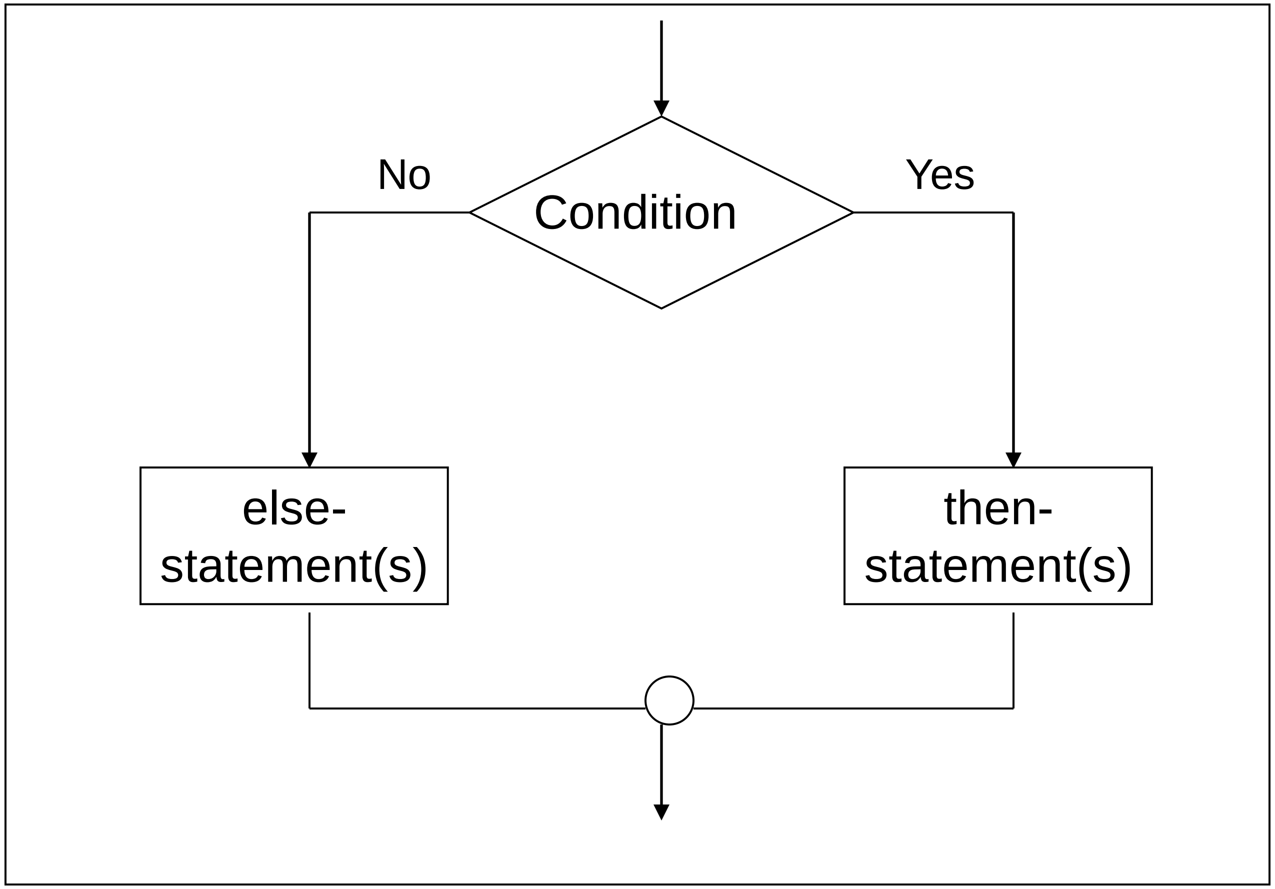


Flow lines - indicate the logical sequence of execution steps in the algorithm.

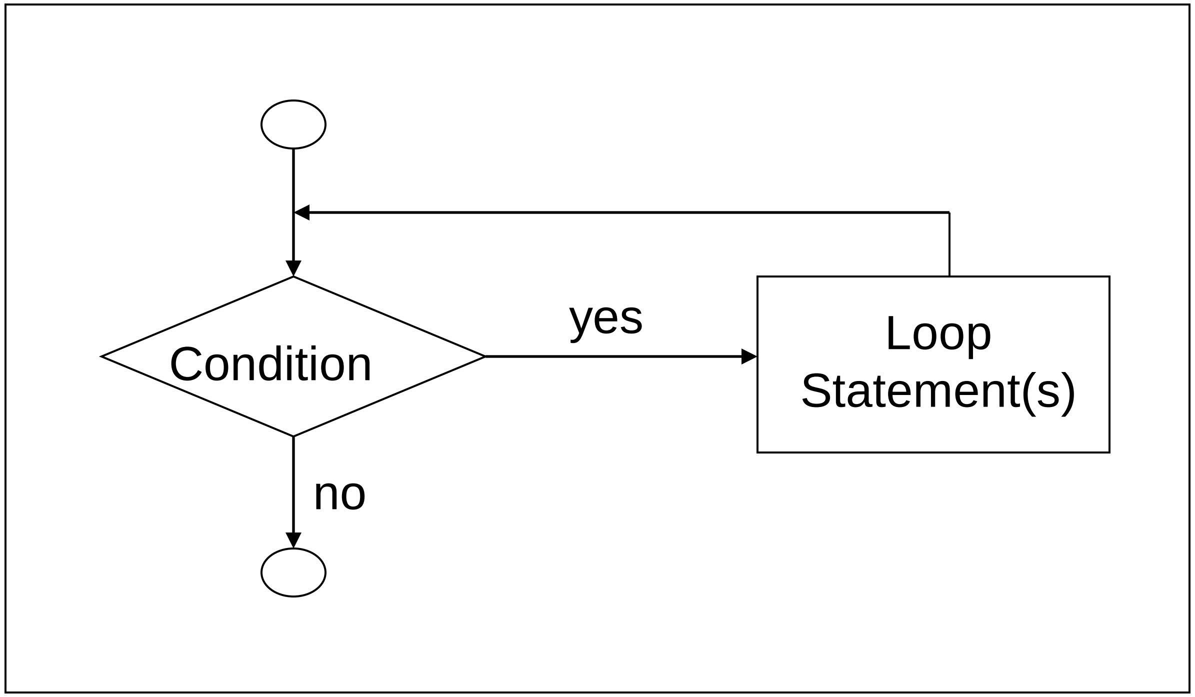
Flowchart – sequence control structure



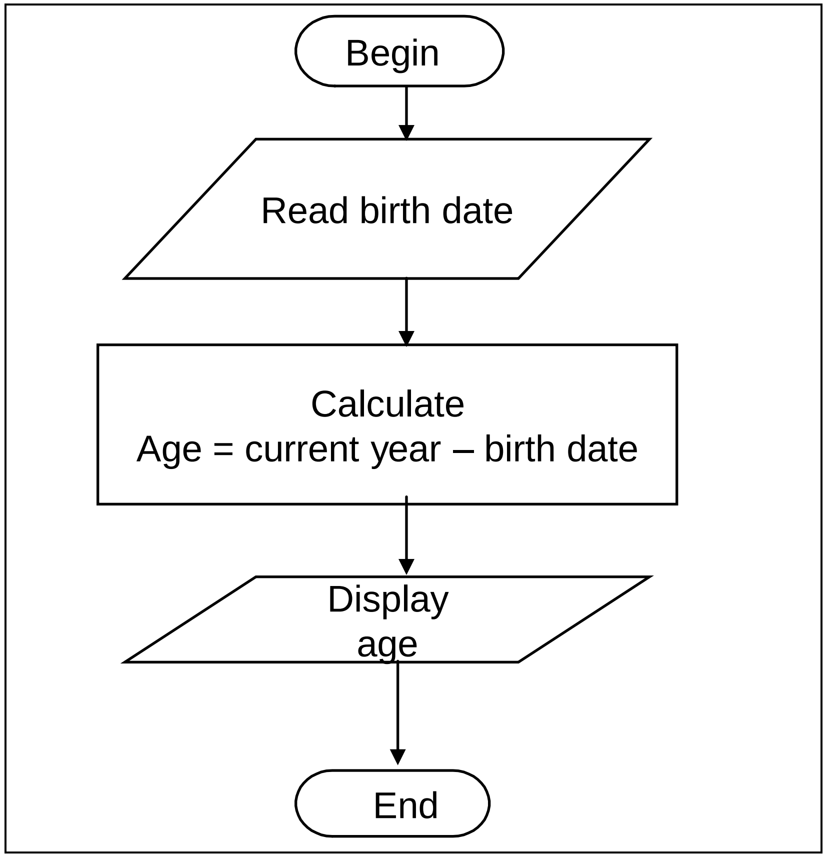
Flowchart – selection control structure



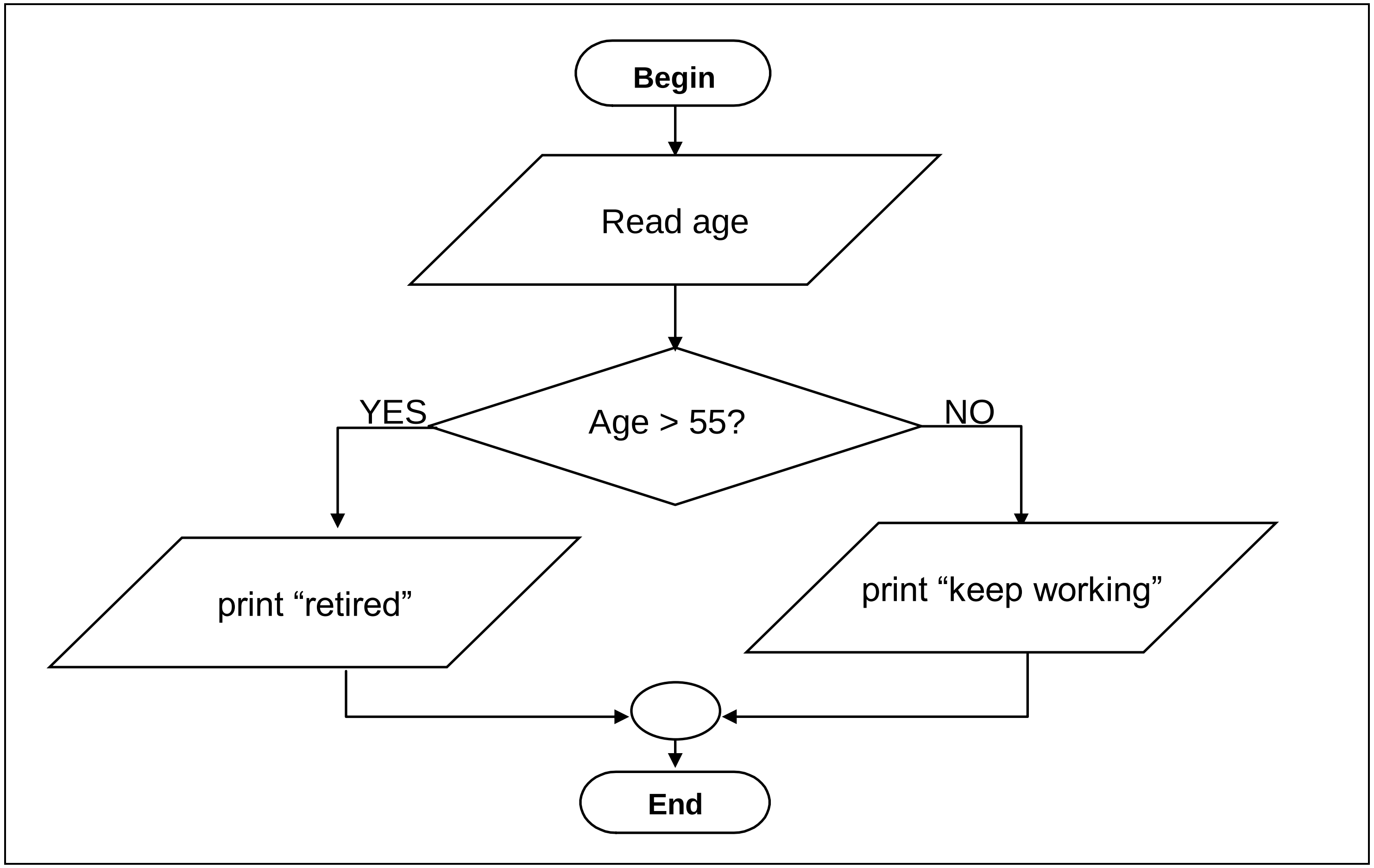
Flowchart – repetition control structure



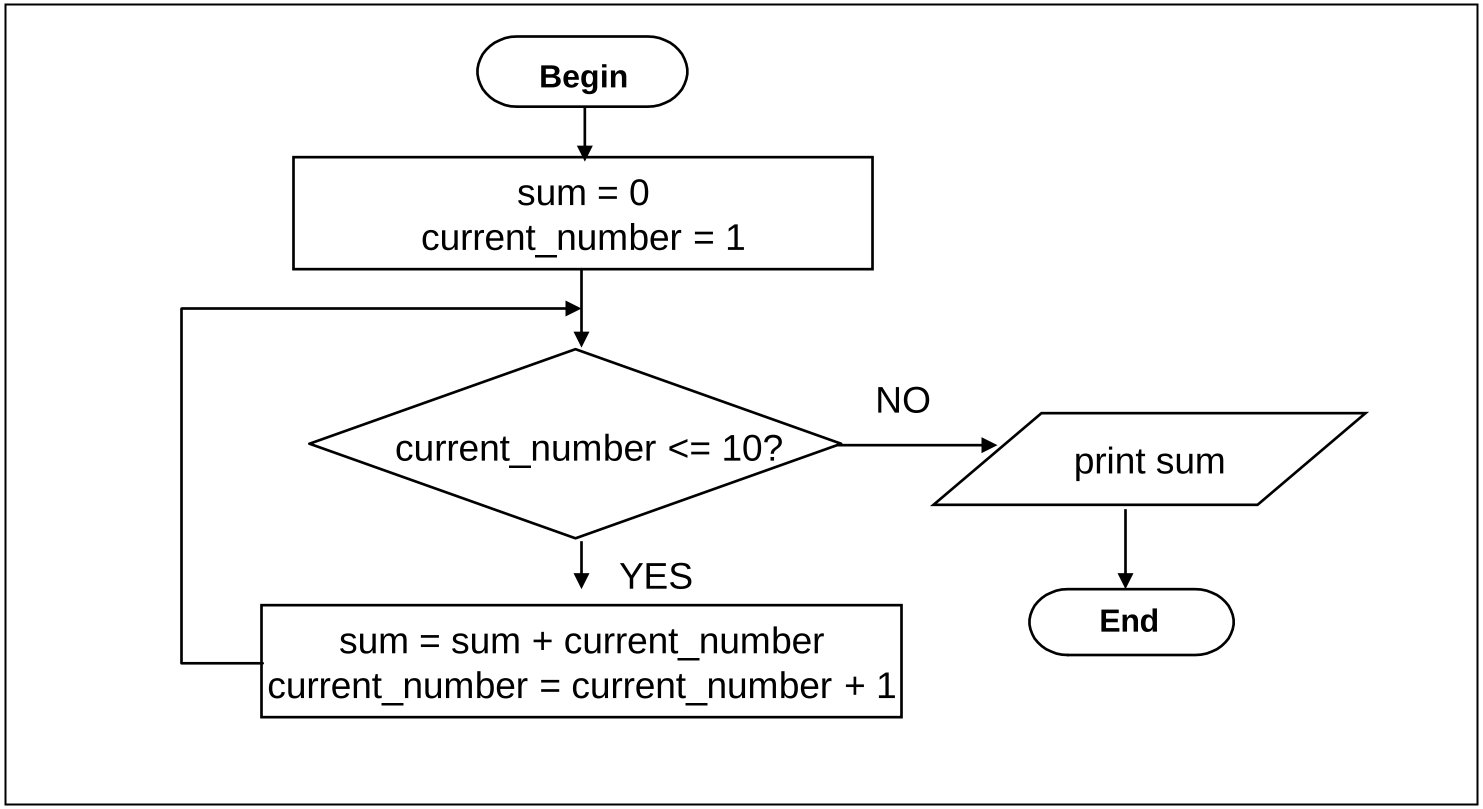
Flowchart – example 1



Flowchart – example 2



Flowchart – example 5



Flowchart – Home Work

- Write the equivalent flowchart for each of the examples given in pseudocoding, i.e

  Example 3

  Example 4

  Example 7

Implementation

- The process of ***implementing*** an algorithm by writing a computer program using a programming language (for example, using C language)
- The output of the program **must** be the solution of the intended problem
- The program must **not** do anything that it is **not** supposed to do
  - *(Think of those many **viruses, buffer overflows, trojan horses,** etc. that we experience almost daily. All these result from programs **doing more** than they were intended to do)*

Testing and Verification

- Program **testing** is the process of executing a program to demonstrate its correctness
- Program **verification** is the process of ensuring that a program meets user-requirement
- After the program is ***compiled***, we must ***run*** the program and test/verify it with **different inputs** before the program can be released to the public or other users (or to the instructor of this class)

Documentation

- Contains details produced at all stages of the ***program development cycle***.
- Can be done in 2 ways:
  - Writing comments between your line of codes
  - Creating a separate text file to explain the program
- Important not only for other people to use or modify your program, but also for you to understand your own program after a long time *(believe me, you will forget the details of your own program after some time ...)*

Documentation cont…

- Documentation is so important because:
  - You may return to this program in future to use the whole of or a part of it again
  - Other programmer or end user will need some information about your program for reference or maintenance
  - You may someday have to modify the program, or may discover some errors or weaknesses in your program
- Although documentation is listed as the last stage of software development method, it is actually an ongoing process which should be done from the **very beginning** of the software development process.

Volume calculation

- Write a pseudocode and a flowchart for a C program that read the value of the height, width and length of a box from the user and print its volume.

Already completed ?

Calculating Electricity Bills

The unit for electricity usage is kWh. For domestic usage, the monthly rate is 21.8 taka/unit for the first 200 unit, 25.8 taka/unit for the next 800 units and 27.8 taka/unit for each additional units.

Given the amount of electricity units (in kWh) used by a customer, calculate the amount of money needs to be paid by the customer to PDB. A bill statement needs to be printed out.

Write a pseudocode and a flow chart to solve the above problem.

Sum of 1 to n

- Write a pseudocode and a flowchart for a program that reads a **positive** integer ***n*** and then computes and prints the sum of all integers between 1 and ***n***.

Summary

- This chapter introduced the concept of problem solving-a process of transforming the description of a problem into a solution.
- A commonly used method – SDM which consists of 6 steps
- 3 basic control structures : sequence, selection and repetition structures
- Pseudocode vs. Flow chart

Questions or Suggestions

THANK YOU!

Inquiry
dishacse@yahoo.com