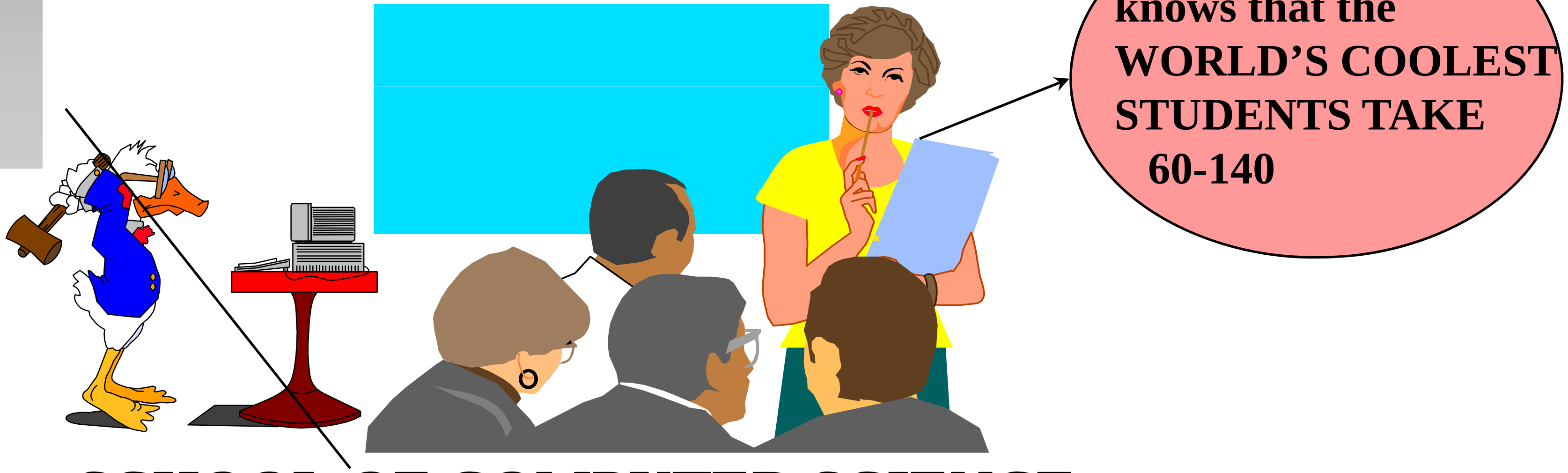
60-140

Introduction to Algorithms and Programming I

- **FALL 2015**
  INSTRUCTOR: DR. C.I. EZEIFE


Everybody knows that the WORLD'S COOLEST STUDENTS TAKE 60-140

SCHOOL OF COMPUTER SCIENCE, UNIVERSITY OF WINDSOR

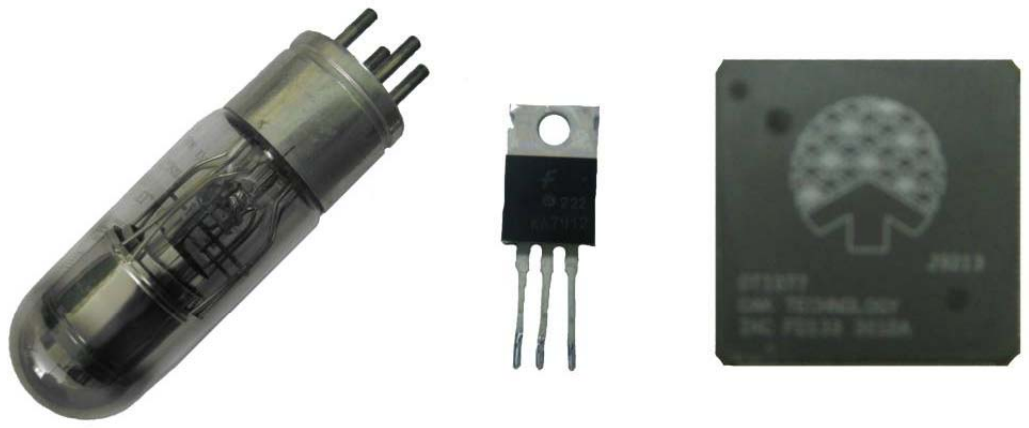
1. Overview of Computer Systems

- Computers are classified based on their generation and type.
- The architecture of different generations of computers differ with advancement in technology.
- Changes in computer equipment have gone through four generations namely:
  - First Generation Computers (1945-1955): Bulky, expensive, consumed a lot of energy because main electronic component was vacuum tube. Programming was in machine language and wiring up plug boards.

Overview of Computer Systems

- **Second Generation Computers (1955-1965): Basic electronic components became transistors. Programming in High level language with punched cards.**
- **Third Generation Computers (1965-1980): Basic technology became integrated circuit (I Cs) allowing many transistors on a silicon chip. Faster, cheaper and smaller in size, e.g., IBM system 360.**
- **Fourth Generation (1980-1990): Personal Computers came to use. Technology in use is large scale integration(LSI). Support for network and GUI.**
- **Higher Generations: Use of VLSI technology.**

Overview of Computer Systems



Vacuum Tube, Transistor, an LSI chip

Types of Computers

- Computers belong to one of these types based on their size, processing power, number of concurrent users supported and their cost.
  - Micro or Personal Computers - support only a single user, very compact in size. Processing power is increasing but still limited when shared by many programs and users, e.g., IBM PC, laptops.
  - Mini Computers (minis) - More processing power can be shared among multiple users, e.g., SUN workstations. Generally, more expensive than micro computers. A network of minis used for

Types of Computers

implementing powerful virtual computing processing powers like grid and cloud computing. Grid computing applies resources of many computers to a single problem (European Data Grid) while cloud computing is used for Internet based computing (e.g., online web mail like GoogleApps) where resources and data are stored online and shared with users.

- Mainframe Computers – Generally, bigger than mini computers, and support hundreds of users at a time, e.g., IBM 370.
- Super Computers - Used for high performance number-crunching applications like processing satellite data from space, e.g., Cray Jaguar with about 150,000 cores (CPUs)..

- Every computer system is made up of hardware and software components.

Hardware Components

- The computer hardware consists of physical electronic components for performing the following functions:

Function	Component
• Data Storage	-Primary memory (RAM and ROM). -Secondary memory (disks & CD-ROMs, usb or flash memory key, tapes)
• Data Processing	Central Processing Unit (CPU)
• Input of Data	Input devices, e.g, Keyboard, mouse, web camera
• Output of Data	Output devices, e.g., printer, monitor, speakers

Data Storage in Main Memory

- Computers represent information (programs and data) as patterns of binary digits (bits)
- A bit is one of the digits 0 and 1.
- Thus, to represent a bit, the hardware needs a device capable of being in one of two states (e.g., a switch of “on” for bit 1 and “off” for bit 0)
- Data and programs are represented as a string of binary digits
- E.g., 9 + 6 in binary are represented as 00001001 and 00000110, then passed to an add circuit to produce binary result.

Data Storage

- Bits of data are stored in memory and bit collections of size 8 make 1 byte.
- A memory cell is made up of 1 to 4 bytes (ie. 8 bits to 32 bits) depending on the word length of the system.
- 1 kilobyte memory has 1024 bytes (approx 10^3 or 2^{10})
- 1 Megabyte memory has approx 10^6 or 2^{20} bytes.
- 1 Gigabyte memory has approx 10^9 or 2^{30} bytes.
- Individual cells in a machine’s main memory are identified with unique names called addresses
- The addresses of 1MB memory are 0 through 1048575 if a memory cell is just 1 byte.

Data Storage in Memory

- Each cell of memory can be read or written (modified) individually.
- RAM is volatile because information stored is lost on power off.
- Thus, secondary memories are used to store data for future use (disks, CD-ROMs and tapes).
- At the user and program level, physical storage addresses are commonly referenced using logical names or addresses like file names for block of data on disk, and variable names for memory cells.

Data Storage

- While numeric data are represented in binary, characters are represented using standard codes.
- One code is ASCII (American standard code for Information Interchange) which uses seven bits to represent a character.

- Disks are a common storage device for storing information for future use. Storage space is generally more available on disk which are cheaper per unit of storage space than main memory.

The Central Processing Unit (CPU)

- CPU is the part of the computer responsible for fetching instructions and data from memory and executing them.
- Central Processing Unit (CPU): Processes information, arithmetic and logical (+, -, *, /, % and logical/relational operations (e.g. And, Or, Not)).
- It receives instructions and data from input devices which it stores in main memory.
- Later, it fetches these instructions and data from main memory and executes them to produce output (results)

The Input/Output Devices

- Input device accepts input from the user and thus has mechanisms for converting characters into bits, e.g., keyboard or mouse.
- Output device displays output or result of processing to the user, e.g., printer or monitor.

Software Components

- The software system drives the physical hardware components through a sequence of instructions called programs.
- There are many software systems in a computer
  - (1) Operating Systems for managing computer resources, e.g., UNIX, LINUX, MSDOS, Windows 2000/XP/Vista/7, Apple Macintosh OS X.
  - (2) Compilers for translating high level language programs to machine language (bits), e.g., C, PASCAL compilers.

Software Systems

- (3) Network Software for allowing more than one computer to be connected together and to share information (e.g., SSH, SFTP, telnet, ftp).
- (4) Productivity Tools for allowing users to perform daily business and office operations in a more productive fashion called productivity tools (e.g., word processors, database, slide presentation and spread sheet programs)
- (5) Others, e.g., utility applications like virus checkers, games, etc.

Overview of Algorithms & Programming Languages

- Computer Science as a field is involved with issues related to
  - algorithm definition, coding, refinement, analysis and discovery
  - as well as issues related to simulation of human intelligence.
- An algorithm is a sequence of steps for solving a specific problem given its input data and the expected output data.
- Examples of real-life algorithms are
  - operating a laundry machine, playing a video game, baking a cake

Overview of Algorithms & Programming Languages

- Algorithms?
- Algorithms are executed by human beings or computers. An example software for executing algorithms is RAPTOR which is available on our cs servers through NoMachine connection.
- When executed by people, an algorithm needs to be presented at their level of understanding and in a language they understand
- When executed by machine (computer), an algorithm also needs to be presented at its level of understanding and in a language it understands.

Overview of Algorithms & Programming Languages

- Example of an algorithm: Example 1.1
- Find the largest common divisor of 2 positive integers. (The Euclidean algorithm)
  - Begin
  - Input: 2 positive integers, large and small
  - Output: their largest common divisor (LCD)
  - Procedure:
    - Step 1: Read large and small
    - Step 2: Remainder = large % small
    - Step 3: If Remainder == 0

Overview of Algorithms & Programming Languages

- then
  - Step 3.11: LCD = small
  - Step 3.12: Goto Step 4
- else
  - Step 3.21: large = small
  - Step 3.22: small = Remainder
  - Step 3.23: Go to Step 2

- Step 4: Output the LCD of large and small
- Step 5: End

Overview of Algorithms & Programming Languages

- E.g., Find the largest common divisor of 40 and 16

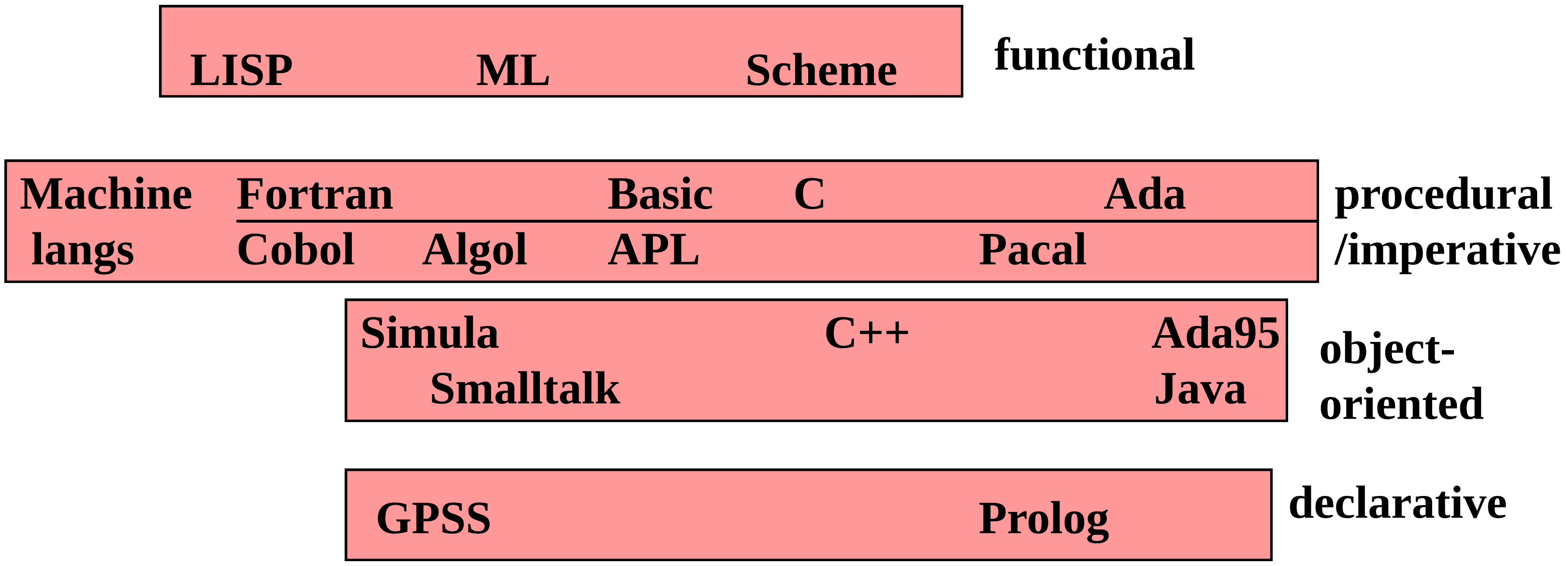
Algorithms & Programming Languages

- Focus of the course (60-140) is on how to discover programs for solving a task (problem solving)
- To do this, we may need to first define the precise sequence of steps for solving this problem represented as an algorithm in pseudocode.
- The computer does not understand pseudocode but a program written in a computer language.
- Thus, for the computer to execute our algorithm, it eventually needs to be translated into a program in a computer language like C.

Algorithms & Programming Languages

- Computer languages are machine language, assembly language and high level languages.
- High level programming languages are easier to use by humans since they are closest to English and Math.
- Current programming languages fall into one of the following four programming paradigms:

Algorithms & Programming Languages



Algorithms & Programming Languages

- Before a program written in a high level language is executed by the CPU, it needs to be translated, linked and loaded into memory in a process called compilation and linking.
- Program preparation process is:
  - Step 1. Type Source program in high level language (eg. C lang) using text editor (eg. pico filename.c)
  - Step 2. Compile to get object program in machine language using cc filename.c
  - Step 3. Link to get load module
  - Step 4. Load into memory to execute with ./a.out

Introduction to C Programming Language

- A C source program file must be given a name with .c extension, e.g., test.c and this file must be prepared with a text editor like Unix/Linux vi editor, nedit, pico or PC's notepad or Visual C++ editor.
- A C compiler is used to compile a C program. To compile on Unix/Linux, use: cc filename.c
- Program instructions that violate the syntax or grammar rules of C will cause syntax errors and must be corrected before a successful compilation is achieved.

Introduction to C Programming Language

- After compilation, the program is run to obtain the desired result. On Unix/Linux, run with the command: ./a.out
- General structure of a simple C program is:

```
#include <stdio.h>
int main(void)
{
Variables declared here; /*  in correct syntax*/
program instructions;  // in correct syntax
return 0;
}
```

Introduction to C Programming Language

```
■ #include <stdio.h>

/* Simple C program for finding the sum of two integers */
int main(void)
{
        int num1, num2, sum;
        printf("Type the two integers to sum        :");

        scanf("%d %d", &num1, &num2);
        sum = num1 + num2;
        printf("sum = %d \n", sum);

        return 0;
}
```

Introduction to C Programming Language

- To work on Windows development environment, you can download Microsoft Visual C++ compiler and check section 1.8 of book on how to use it to type, compile, link and execute your program.
- Note that sections on how to use pico and nedit text editors on Unix/Linux are 1.5.2 and 1.5.3 of book
- Section on how to use SSH and SFTP is 1.6 of book.
- Section on Macintosh Personal computer is 1.7
- Note that each chapter ends with possible programming errors and has a section on exercises with solutions.

2. Problem Solving Steps

- Objectives
  - Understand what a problem is
  - Discuss six problem solving steps (RCMACT)
- Types of Problems
- 1. Problems with Algorithmic Solutions
  - Have a clearly defined sequence of steps that would give the desired solution
    - E.g. baking a cake, adding two numbers

Problem Solving Steps

- the sequence of steps or recipe for arriving at the solution is called the algorithm

- 2. Problems with Heuristic Solutions
  - Solutions emerge largely from the process of trial and error based on knowledge and experience
  - E.g., winning a tennis game or a chess game, making a speech at a ceremony
- Many problems will require a combination of the two kinds of solution

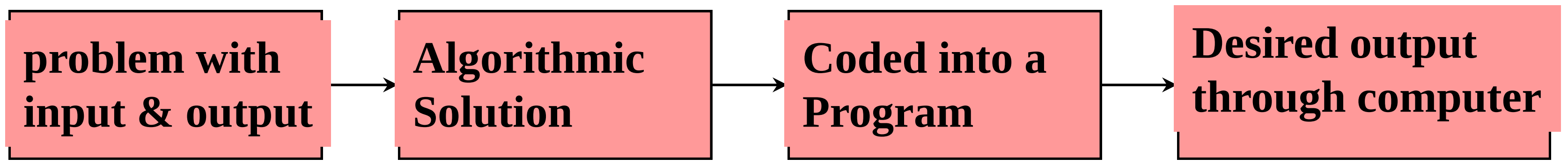
Problem Solving Steps

- In this course, we are mostly concerned with algorithmic problems.
  - computers are good at solving such problems
- Heuristic problem solving (getting computers to speak English or recognize patterns) is the focus of Artificial Intelligence

- What is a Problem?
  - It has three main components of (i) input data set, (ii) desired output data set and
  - (iii) we want to define a sequence of steps (algorithm and/or program) for transforming input data to desired output data.

Problem Solving Steps

- What is a problem’s algorithmic solution?
  - the sequence of steps needed to reach the desired output or the best output data expressed in pseudocode.
- What is a Program?
  - the sequence of steps (algorithms) expressed(coded) in a computer language like C.



Problem Solving Steps

- Example 2.1: Management wants to see the patterns in absenteeism across its two departments, dept1 and dept2 for one week. It is interested in knowing the total absenteeism in each department in the one week it collected data. You are required to identify the input and output data of this problem and attempt to define an algorithm and a program.

Problem Solving Steps

Problem Solving Steps (RCMACT)

- 1. Defining the Problem Requirements (R)
  - clearly defining the problem in words, stating the input and output data as well as the processing logic. It may need knowledge or familiarity with a real life environment to understand the needs of a problem
- 2. Identifying Problem Components (C)
  - From the problem definition, identify the list of problem inputs, outputs, constraints and relationships between input and output data expressed in coherent formulas.
- 3. Possibly break problem solution into small modules (M)
  - This step uses top-down design approach to solve a big problem using structure chart. This step may be skipped for small problems that do not need breaking down.

Steps in Problem Solving

- 4. Design the Algorithm to solve the problem (A)
  - Best among many alternative ways for solving the problem is chosen.
  - Define algorithmic solution for all modules in structure chart.
  - E.g., solution that is most cost efficient, space efficient or fastest.
- 5. Implementation and Coding (C)
  - Translate the algorithmic solution from step 4 to C programming language to obtain a program.

Steps in Problem Solving

- Programs have to obey the grammar rules (syntax) of C and any violation results in a syntax error (called bug).
- A bug needs to be corrected during debugging before the program is accepted by the compiler.
- Other types of error that might need to be corrected during coding for correct results to be obtained are logic and runtime errors.
- The C implementation of Example 2.1 is: (solve)

- 6. Test or Evaluate the solution to ensure it produces desired results(T)
  - A set of complete test data is used to test the correctness of the program by tracing the program/algorithm with hand and running the program.

Difficulties With Problem Solving

- Failing to outline details of the solution (algorithm and program) completely
- Failing to define the problem correctly
- Failing to generate a sufficient list of alternatives
- Failing to use a logical sequence of steps in the solution
- Poor evaluation of the solution (algorithm and program)
- Always remember that computer does not see and needs to be given all details about what to do.

3. Types of Algorithmic and Program Instructions

- Objectives
- 1. Introduce programming language concepts of variables, constants and their data types
- 2. Introduce types of algorithmic and program instructions
- 3. Discuss Read(scanf)/Print(printf) and Assignment instructions.
- Variables and Constants
- Variables and Constants are names for storage locations (memory cells) holding data values processed by the computer

Problem Solving Concepts (Variables and Constants)

- Programmers define data relevant to a problem as constants or variables
- Variables and constants form building blocks for equations and expressions used in problem solving.
- Both variables and constants have specific data types. E.g., alphabetic or numerical value
- Differences Between Variables & Constants
  - The value of a variable may change during processing of a problem in a program, but the value of a constant cannot change

Problem Solving Concepts (Variables and Constants)

- The format for declaring variables in both an algorithm and a C program is:

  ```
  datatype    variablename[,variablenames];
  ```

- The format for declaring constants in both an algorithm and a C program is:

```
const datatype
   variablename=value[,variablenames=values];
```

A constant can also be defined using preprocessor directive: #define constantname value

Problem Solving Concepts (Variables and Constants)

- Example 3.1 : A class of ten students took a quiz. The grades (integers in the range 0 to 100) for quiz are available to you. Determine the class average on the quiz.
  - Identify the constants and variables you need to solve this problem

Variables and Constants

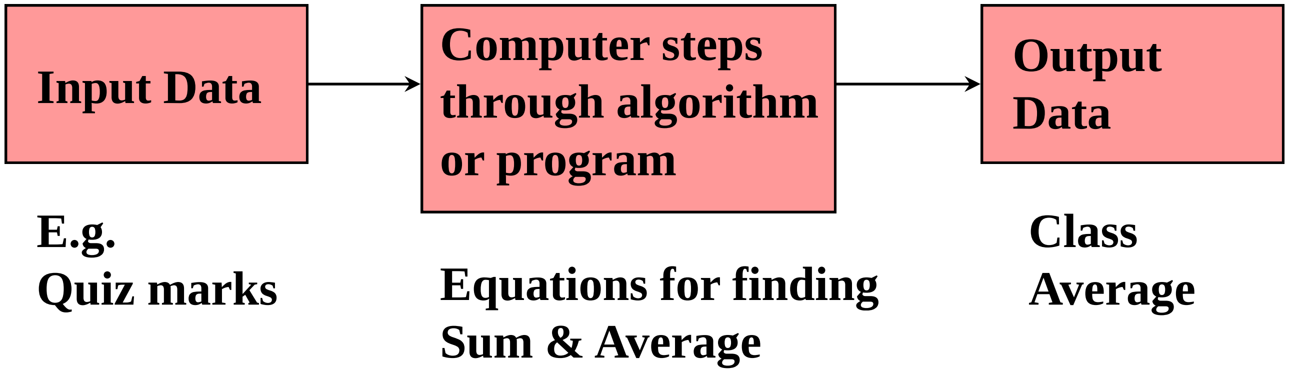
- **Show the variables and constants needed to solve these problems.**
- **Example 3.3: You are required to count the number of 60-140 students who have completed assignment #1. The class has 250 students**
- **Example 3.4: Find the sum and product of 2 numbers.**

Issues concerning Variables/Constants

- Rules for naming variables/constants differ from language to language. C allows unlimited number of alphanumeric characters.
- It is good problem solving practice to use variable names close to the meaning of its data values
- Multiple word variable names should be separated with underscore to make them more readable, e.g., asn1_140
- Use variable names with less than 15 characters to avoid ambiguities.
- Check Table 3.1 of text for keyword names not to be used as variable names in programs.

Data Types

- Input Data are facts (values) used by the computer to process algorithmic solutions and programs to a problem while output data are the results (values) produced by the computer after running the program.


Input Data
Computer steps through algorithm or program
Output Data
E.g. Quiz marks
Equations for finding Sum & Average
Class Average

Data Types

- Data are of many different types:
- 1. Integer Data Type(called int or long int in C): Integers (whole numbers, e.g., 1577, -18). Arithmetic operations can be performed on this data type. Example declarations are: int age, score;
  long int bignumber;
- 2. Real Data Type(in C are float or double): numeric value with whole number and decimal parts, e.g., 23.75, 230000 or 2.3E5 in Scientific notation. Arithmetic operations are performed on this data type. E.g.,
  float salary;
  double bigrealnumber;

Data Types

- 3. Character Data Type(called char in C): all letters, numbers and special symbols (surrounded by single quotation signs), e.g., ‘A’, ‘a’, ‘1’, ‘+’. E.g.,

```
char grade, location=‘A’;
```

- 4. String Data Type(implemented as char variablename[ ] in C)
  - Combination of more than one character (surrounded by double quotation signs), e.g., “Randy”, “85611”, “519-111-2345”. Eg declarations:

```
char lastname[15], address[30];
```

Data Types

- 5. Boolean or LOGICAL Data Type (implemented with int in C)
  - TRUE or FALSE are the only data values.
  - In C, an int variable with a value of 0 can be treated as logical type FALSE, while a value of not equal to 0 (like 1 or a number > 1) is TRUE. E.g.,

```
          int flag=0;  /* declares flag as FALSE */
```

- Each data type has a data set, the set of values from which any datum of that data type is specified.

Data Types

Data Type	Data Set	Example Data
Integer(int)	All whole numbers, e.g., -2^{31} to $(2^{31}-1)$	1999, -67
Real (float/ double)	All real numbers (whole + decimal parts)	1999.0, 258923.61 0.00581
Character (char)	All letters, numbers, and special symbols.	‘A’,`b’,`K’, `1’,`8’,`+’
String(char variable[ ])	Combination of >1 chars.	“Atlas”,“956”
Logical (int with 0 or !0)	TRUE FALSE	TRUE FALSE

Operations on Character/String Data

- 1. Character/String data can be compared and arranged in alphabetical order (using their ASCII codes)
  - A comparison between characters ‘A’ and ‘B’ gives ‘A’ < ‘B’ since 65 < 66 (see Appendix A of text)
  - The character string “Money” is greater than “Make” because `o’=111 > `a’=97 but > is not used for string comparison in C. String functions and processing are discussed in ch. 8.
- 2. Other character and string operations including a lot of built-in functions are available in C and more details are in sections 4.3 and 8.4 of text.

Uses for Different Data Types

- 1. Numerical Data (integer(int/long int) and real(float/double))
  - used in business, government & academic applications for values such as salary, price, scores
  - used for numbers that will require some computations on them
  - E.g., number of employees, assignment marks, salary.
- 2. Character Data (char)
  - used for initials, grades or things needing only one character. No mathematical calculation allowed.

Uses for Different Data Types

- 3. String (char variablename[ ])
  - used for names, labels and things needing more than one character and not needing any mathematical calculation
  - e.g., student number, phone number & account number.
- 4. Boolean (int with value 0 or !0)
  - used in making yes-or-no decisions
  - e.g., is a student's grade 'A' ?

Rules for Data Types

- 1. Data types are not usually mixed. E.g., character data cannot be placed in a variable memory location designated as numerical. C allows use of cast operator for type conversion when necessary.
- 2. Data defining the value of a variable or a constant will be one of four data types: numerical, character, string and Boolean.
- 3. The use of any data outside the data set of the data type results in an error.
- 4. Only valid operations on a data type are allowed. E.g., numbers designated as string type cannot be used in calculations.

Algorithmic Structure

```
▪ [Global Input/Output Variables]
[Function Prototype list : type and parameters]
  Mainalgorithm
  {
  Input: Variables/ Constants lists and their types
  Output: Variables lists and their types
  Others: Variables/Constants lists and their types
  /* Now the body of Main Driver or Control Module is defined*/
     Instruction 1;
     Instruction 2;
            :
     Instruction n;
  }
  [function definition 1] …
  [function definition n]
```

C Program Structure

- **#include <stdio.h>**
  [Optional Global Variable declarations]
  [Function Prototype list : type and parameters]
  int main(void)
  {
  variable declarations;
  /* Now the body of Main Driver or Control Module is defined*/

```
   Instruction 1;
   Instruction 2;
     :
   Instruction n;
    return   0;
}
```

[function definition 1] …
[function definition n]

Types of Algorithmic Instructions

- An algorithmic or program instruction can be one of the following types:
  - 1. Read (scanf in C) or Print (printf in C) instruction - Read instruction is used to read data from the key board while a Print instruction prints output to the monitor.E.g.,
    ```
    Algorithm: Read (Num1);
               Print (Num1);
    C Program: scanf("%d", &Num1);
               printf("%d", Num1);
    ```
  - 2. Assignment instruction - used to copy a computed value on the right hand side of an equation to the memory cell labeled the left hand side of the equation. E.g., both algorithm and program: `sum = Num1 + Num2;`

Types of Algorithmic Instructions

- **3. A function call - used to delegate some portion of the task to a small independent program segment. E.g., In both algorithm and program: Compute_Product(Num1, Num2,&product);**
- **4. A Decision instruction - used to decide between which one of a number of alternative instructions to execute. E.g., if ((large % small) ==0) lcd = small;**
- **5. A Repetition instruction - used to cause a sequence of instructions to be executed repetitively a number of times or until some event occurs. E.g, while, do-while and for instructions.**

Read(scanf) and Print(printf) Instructions

- **Read (scanf) instructions get input data typed by the user from the key board, while print (printf) instructions display the value of a variable or an expression on the screen.**
- **The general forms of these two instructions are:**
  In an Algorithm:
  Read(variable1, variable2,…, variable*n*);
  Print(variable1, variable2, …, variable*n*);
- The format of C program scanf and printf instructions are:
- scanf("format specifiers", &variable1, &variable2, .. ,&variablen);
- printf("format specifiers", variable1, variable2, .. ,variablen);

Read(scanf) and Print(printf) Instructions

- Both scanf and printf accept a number of parameters (arguments).
- A parameter could be a variable name, an expression or a string literal, but only variable name parameters are accepted by scanf.
- Both scanf and printf have the first parameter as a string literal for format specifiers (specifying the data type of the variables or data in the parameters).
- Format specifiers for int is %d and %ld for long int, %f for float and %lf for double, %c for char and %s for string.

Read(scanf) and Print(printf) Instructions

- Example 3.5: Find the sum of two numbers
  Algorithm: Read (num1, num2);
  Print (“The sum of”, num1, “and”, num2, “is”, sum);
- C Program:

```
scanf(“%d  %d”, &num1, &num2);
printf(“The sum of %d and %d is %d”, num1, num2,
sum);
```

- Note that if the variable type for scanf is string, then, the address operator, &, does not precede the variable.

Output Formatting with printf

- **Use the format specifier %E or %e to display a floating point number in exponential form, %o to display in octal, %x or %X to display in hexadecimal. E.g.,**
  printf (“%e\n”, pi*10); **will print 3.14159e01.**
- **Specify the number of columns, “c”, used to print an integer value with specifier %cd, e.g., %3d, %4d. E.g.**
  printf (“%3d\n”, 25);
  printf (“%4d\n”, 25);
- **The number of columns, “c”, and number of digits, “d”, to the right of decimal point for a floating point value is specified with %c.df, e.g., %8.1f.**
  printf (“%8.1f\n”, 3.14159);

Output Formatting with printf

- An escape sequence is used for printing characters not printable through simple inclusion in printf control string. E.g., (" printed as \``, newline printed as \n) for printf is represented by a backslash followed by a particular escape character. See Table 3.6 for details.
- Other characters like % must be typed twice to be printed with printf as in printf(" 50%%\n");
- Check tables 3.3 to 3.6 for format control string parameters.

Assignment Instructions

- An assignment instruction is used to read a value from a memory cell (any variable on its right hand side) and to assign a value to a memory cell (the only variable on its left hand side).
- The general form of an assignment instruction (in both algorithm and C program) is:
  variable = expression;
- Example 3.6: Copy the contents of variable assn1 to assn2.
- Solution: assn2 = assn1;
- Example 3.7: Copy the sum of assn1 and assn2 into assn3
- Solution: assn3 = assn1 + assn2;

Expressions

- **What is an expression?**
- **An expression is a variable, a constant or a literal or a combination of these connected by appropriate operators. There are 3 basic types namely:**
  - **Arithmetic expressions : variables are numerical and connected by arithmetic operators (+,-,/,%,*)**
  - **Relational expressions : variables are any type (but same type on both sides of operator) connected by relational operators (<,>,<=,>=,==,!=). Result is boolean**
  - **Logical expression: apply to logical values using logical operators NOT (!), AND (&&), OR (||).**

Operators

- Operators tell the computer how to process data
- They are used to connect data (operands) in expressions and equations to produce a result.
- Types of Operators
  - 1. Arithmetic: addition (+), subtraction (-), multiplication (*), division (/), integer division (/), modulos division (or remainder after integer division %).
  - E.g. if Jane has worked 17 days during the month in a 5-day work week, how many whole weeks has she worked and how many days not belonging to a week has she worked?

```
Numweek = TotalDays / 5;
Numdays = TotalDays % 5;
```

Operators

- 2. Relational Operators: <, >, <=, >=, == and !=
- Used to program decisions and need data operands of any type except the two operands must be same type.
- E.g., 31 > 15   is TRUE
- “Alpha” < “Beta” is wrong operation in C because string comparison is done with function (strcmp).
- ‘C’ < ‘A’ is FALSE
- Results of these operators are either TRUE (!0) or FALSE (0)

Operators

- 3. Logical Operators: Used to connect relational expressions (decision-making expressions)
- Logical operators are NOT (!), AND (&&), OR (||)
- E.g., Refuse registration into 60-141 if mark in 60-140 is less than 50%.
- if (m60140 < 50)

```
  printf (“Error, Need at least a 50%% in 60-140”)
else
  printf(“Registration in 60-141 successful\n”);
```

- Operands and results are logical type (TRUE or FALSE)

Definition of Logical Operators

- NOT (!)

A	! A
T	F
F	T

AND (&&)

A	B	A && B
T	T	T
T	F	F
F	T	F
F	F	F

OR (||)

A	B	A || B
T	T	T
T	F	T
F	T	T
F	F	F

Precedence Hierarchy of Basic Operators in C

Higest	Operator	Associativity Rule
↑	( )	left to right
	Functions	left to right
	!, (-)	right to left
	*, /, %	left to right
	+, -	left to right
	<, >, <=, >=,	left to right
	==, !=	left to right
	&&	left to right
Lowest ↓	||	left to right

- Parenthesis can be used to overide precedence order

Setting up Numerical Expression

- 1. Set up the following mathematical expression for use in a C program.
- a. $X(3Y + 4) - \frac{4Y}{X + 6}$
- Ans:
- b. Set up the following math equation as a C equation.
- $Y + 3 = X (Z + 5)$
- Ans:

Setting Up Relational Expression

- 1. Given the expression X is less than Y + 5, set it up for use in a C program.
- Ans:
- 2. In order to cash a check, a customer must have a driver's license (A) or a check-cashing card (B) on file at the store, set up this transaction for a C program.
- Ans:

Evaluating a Mathematical Expression

- Evaluating a math expression means assigning values to all variables and testing the result to determine if it is correct.
- Evaluate
- 5 * (X + Y) - 4 * Y / (Z + 6)
  with X = 2, Y=3, and Z=6

Evaluating Relational/Logical Expressions

- Evaluate the Relational Expression
  A - 2 > B
  where A=6, B= 8
-
- Evaluate the logical Expression
  A && B || C && A
  with A=TRUE, B=FALSE, C= TRUE

Evaluating a Relational/Logical Expressions/assignment instructions

- Evaluate the following assignment instruction
  Q = !(A < B) && (C || D)
  where A=4, B=2, C=TRUE and D=FALSE

Other C Operators

- 1. C’s Increment Operator (++) for adding 1 to a variable.
  E.g., Num=Num + 1; is same as :
  Num++; (postfix form that adds 1 after using Num)
  and
  ++Num; (prefix form that adds 1 before using Num)
- 2. C’s Decrement Operator (--) for subtracting 1 from a variable.
  E.g., Num = Num – 1; is same as:
  Num--;
  and   --Num;

Other C Operators

- 3. Bit Operations in C
  - A) Bitwise OR (|) for ORing two bit values.E.g., 1 | 0 is 1, 1 | 1 is 1 and 0 | 0 is 0
  - B) Bitwise AND (&) for ANDing two bit values. E.g., 001111 & 010111 is 000111
  - C) Bitwise Exclusive OR (^) that returns 1 only when one of the two input bits is 1. E.g., 1 ^ 1 = 0, 0 ^ 0 = 0 and 1 ^ 0 = 1, 0 ^ 1 = 1.
  - D) Bitwise Inverse operation (~), which flips the value of an input bit. E.g., ~ 1 is 0 and ~0 is 1.

Other C Operators

- E) Bitwise left Shift operation (<<) for shifting the input value a number of bits to the left. E.g.,
  00010101 << 2 is 01010100
- F) Bitwise Right shift operation (>>) for shifting the input value a number of bits to the right. E.g.,
  00010101 >> 2 is 00000101

- 4. C's sizeof operator, which returns the number of bytes a variable of type requires. E.g., sizeof(int) is 4.
- 5. C's cast operator, which accepts an expression as its operand and converts the type of the operand to the type specified by the cast operator. Used as:

  (Type) Expression

Other C Operators

- 6. C's Operator Assign Operations: Used for writing short forms of various forms of assignment instructions. These instructions have an arithmetic (+, -, *, /, %) or bitwise (<<, >>, ^, ~, |, &) operator preceding an assignment operator. General form is:
- Variable operator= value;
  E.g., total += 40 means total = total + 40;
  total -= 10 means total = total – 10;
  total /= 4 means total = total / 4;
- Figure 3.4 of text shows the comprehensive operator precedence and association order in C.

4. Problem Solving Tools (Top-Down Design)

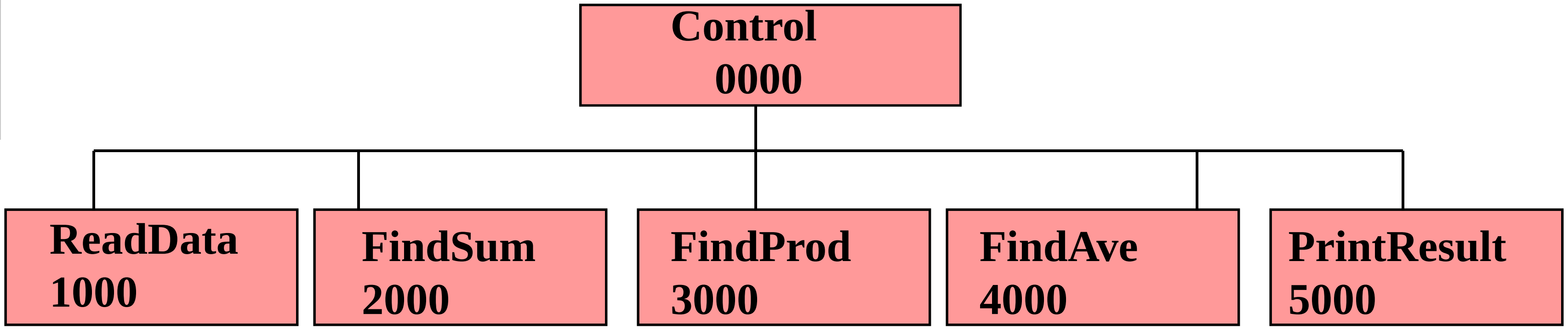
- Objective: 1. Discuss structure chart
  - 2. Discuss functions and algorithms with parameters, local and global variables
  - 3. Discuss Built-in Functions and flowcharts.
- Top-down design approach to problem solving is based on the principle of "divide and conquer".
- It breaks down the problem to be solved into smaller sub-problems using the problem solving tool of structure chart

The Structure Chart

- Example 4.1: Write a solution (algorithm/program) that inputs three different integers from the keyboard and then prints the sum, average and product of these numbers. Use top-down design approach showing all 6 problem solving steps.
- Top-down design approach uses a structure chart with the main problem as the control module and the subtasks located below it.
- A module processes only those tasks directly below and connected to it.

The Structure Chart

- Modules are given unique number labels based on their level with the top labeled 0000



Types of Modules in a Problem

- 1. Control Module or the Main Driver: shows the overall flow of the problem and calls other modules
- 2. Init Module - for initializing data; e.g., Sum=0, knt=1
- 3. Read and Data Validation Module
- 4. Calculation Modules: for arithmetic calculation, string manipulations like sorting
- 5. File Maintenance Modules for adding or deleting records from a file
- 6. Print Modules: Prints outputs
- 7. Wrap-Up Module: E.g. closing files or printing to mark normal end of program.

Cohesion and Coupling

- In separating a problem into parts, it is desirable to
  - 1. Create modules that perform independent functions with one entrance and one exit -- cohesion
    - Cohesion allows modules to perform independent tasks (e.g., functions FindSum, FindDiff)
  - 2. But modules need to work together towards solving the bigger problem --- Coupling
    - Coupling allows connecting modules through an interface where data can be transferred from one module to another (e.g., through parameters and global variables)

Coupling Techniques

- That is, how can data be communicated between modules? (3 approaches)
  - 1. Use of Parameters in functions, e.g.,
    - a) Call-by-Value Parameters with a function return value for the function output. An example call is Sum = FindSum(X,Y)
    - b) Call-by-Reference Parameters for function outputs. Example call is FINDSUM(X,Y,&Z)
  - 2. Use of Data that all modules can access (global variables)

Top-Down Design: Cohesion and Coupling

- Top-Down design is achieved through cohesion (separating a large problem into independent modules) and coupling (modules working together towards one goal).

Advantages of Top-Down Design

- 1. Many programmers can work on a large problem producing faster results
- 2. It is much easier to write and test many small modules than a single one
- 3. It is much easier to modify small modules.
- 4. Reusability: a defined module can be used several times by any module.

Functions

- A function is a set of instructions that performs specific tasks and can return only one value although it can modify others through parameters (call-by-reference).
- Functions contribute towards the solution of a problem
- They mostly make the solution of a big problem more efficient because they can be reused, more elegant because it is structured and easier to read
- Many languages provide a variety of built-in functions.
- However, modules of the structure chart are defined as functions in the problem solution by the problem solver.

Functions

- A function needs to be provided for each module in the structure chart.
- In the solution, we first provide each function prototype used to tell the compiler functions to expect, the type of result they return and their parameter types
- Secondly, we provide the full algorithmic/program definition of each function
- Format for specifying a function prototype is functiontype functionname (type for par1, type for par2, …, type for parn);
- Example function prototype: int FindSum(int, int, int *);
- Example funtion header: int FindSum(int X, int Y, int *Sum);

Functions Definition Structure

- Functiontype Functionname ([type par1], ..[type par*n*])

```
{
        [local variable declaration];
        instruction1;
        instruction2;
        :
        instructionn;
        [return (output variable or expression or 0)];
}
```

Functions Definition Example

```
int FindSum(int first, int second)
{
        int sum;    /* This is a local variable of FindSum  */

        sum = first + second;
        return(sum);

}
```

Function Parameters

- Parameters are data needed by the functions to return results
- E.g., in X = SQRT(N), N is the parameter, and in Getsum(X,Y,&Z), X,Y,&Z are the parameters.
- Parameters are surrounded by parenthesis
- A function can have 0 or more parameters
- E.g., rand( ) generates a random number
- A parameter can be a constant, variable or an expression
- E.g. valid call is: Getsum(29, 10, &Z)

Parameters

- Parameters are data passed from one module (function) to another.
- They enable us to avoid using global variables so that we can improve on data protection.
- Parameters are enclosed in brackets in the definition of the module (these are called formal parameters).
- Any other module can request the services of this module by specifying the actual value for each parameter in the exact order they appear in the definition and with same data type.

Parameters

- Parameters used in the calling statements are called actual parameters.
- The formal list and the actual list do not need to have the same name so long as they are in the correct order with the right data type.
- Parameters can be passed in two ways
  - 1. Call-by-Value: Here, only the value of the variable specified in the actual parameter list is passed to the module called (not its address).
  - means value the actual parameter had in the calling module is not overwritten.

Parameters

- 2. Call-by-reference Parameter: The address of the memory location for the actual parameter is passed to the called module. Example in the function call, FindSum(X, Y, &Sum), the actual parameter &Sum is the address of the Sum in the calling module.
- So, the value it has in the calling module before the call can be changed or replaced.

Address Operator (&), Pointer Variable and Indirection Operator (*)

- A pointer variable stores only memory addresses
- A pointer variable has to be declared before use in a program with the format:

  datatype_pointed_to *variablename;

E.g., if in main, Num1 is an integer variable with value 35 and Sum is another int variable with value 200.

- We might want to call a function to find the sum of Num1 and Sum with the call FindSum(Num1, &Sum).
- The actual parameter &Sum is the address of the variable Sum. This means that in the definition of

Address Operator (&), Pointer Variable and Indirection Operator (*)

- the function FindSum, the second formal parameter has to be declared as a pointer variable that points to an integer value. Thus, the Function header is:
- void FindSum(int Num1, int *Sumf)
- Here, the formal parameter Sumf, is a pointer variable.
- Note that in the function call, FindSum(Num1, &Sum), the address operator (&) is used to obtain the address of the variable Sum in main.
- The indirection operator (*) is used to obtain the value pointed to by the pointer variable using the pointer variable name. E.g., to add and print the sum in main using the Sumf in the function FindSum, we use:
- *Sumf += Num1;
  printf (“%d”, *Sumf);

Parameter Example (Example 4.2)

- **Given the following program solution, show the values of the variables a, b, c, x, y, z in the control module (module1) after each function call to module2.**
-

```
/*   function prototype declaration for Module2   */
void module2(int, int, int *);
void main(void) {
int a=3, b=4, c=5, x=7, y=8, z=10;
/*   body of main   */
module2 (a, b, &c); /* a first call to Module2   */
module2 (x, y, &z);  /* a second call to Module2 */
}
void module2 (int a, int b, int *c)
{    a += 4;
     b += 4;
     *c += 4;     }
```

Parameter Example

The Algorithms/Programs

- Algorithms are mostly written in Pseudocode (a cross between English language and high level programming language)
- Each instruction in an algorithm should directly convert to a programming language statement during coding.
- Each module in the structure chart has a separate set of instructions in the algorithm defined as a function.

Algorithm Example (Example 4.3)

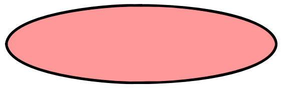
- Provide the algorithm and program to Example 4.1 using the structure chart already defined and parameters. Also provide the flowchart solution.

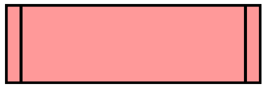
Flowchart

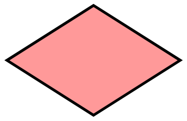
- The solution to a problem can be organized in a number of ways and each algorithmic solution corresponds to a flowchart.
- A flowchart is a graphical representation of an algorithm. It shows the sequence of execution of the instructions.
- Flowchart and algorithm represent the same execution flow in different forms.
- A flowchart always starts at the top of the page with straight and neat connecting flow lines.

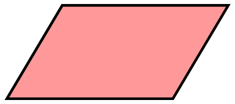
Flowchart

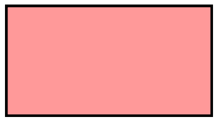
- Flowchart symbols are given below:

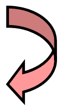

Start, End, Exit
Return, enter

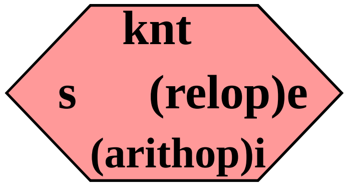

Module


Decision, while,
switch

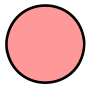

Read, Print

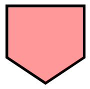

Assignment instr.

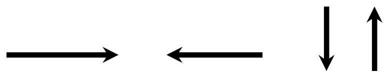

Indicating function parameters



Automatic counter
for (knt = s; knt *relop* e; knt *arithop*= i)


On-Page Connectors


Off-Page Connectors


Flow lines

Local and Global Variables

- Cohesion and Coupling are realized through the concept of local and global variables
- In a module, the difference between local and global variables is in their scope (where, in which modules their values are allowed to be used)
- Local variables can be used only inside the module they are declared.
- Global variables are declared outside functions and can be used by only functions below their declarations in the algorithm or program.
- Global variable is one coupling method, a better coupling method is use of parameters

Local and Global Variables

- **Example 4.4: Solve the problem of Example 4.1 using global variables and not parameter passing. Show all the local and global variables in your solution.**

Local and Global Variables

Shortfalls of Global Variables

- 1. Side Effects: A global variable may be accidentally or wrongly altered by an incorrect or malicious module (that is, no protection of data)
- 2. No Duplication of Variable names: When an inner module declares a local variable with same name as the global variable, all changes it makes to this variable is local and it no longer has access to the global variable.

Built-in Functions

- Some common built-in functions provided by C language are:
- 1. Mathematical functions: E.g., sqrt(x), exp(x), log(x), ceil(x), floor (x), pow(x,y), fabs(x) for absolute value
- 2. String Functions:E.g., copy part of the string into another variable, find number of characters in a string. E.g., strcmp(s1,s2), strstr(s1,s2), strcpy(s1,s2), strlen(s), strcat(s1,s2)
- 3. Character Functions: For manipulating character data. E.g., isdigit(C), islower(C) for seeing if C is lower case letter or not.
- 4. Conversion Functions: convert data from one data type to another. E.g., used to convert a string value to a numerical value, in “C lang.” atoi(“2593”)=2593 (integer value)

Built-in Functions

- **5. Utility Functions: used to access information outside the program and the language in the computer system, e.g., date and time functions.**

5. Program Logic Structures

- **Objectives**
- **1. Program Logic structures (General)**
- **2. Discuss Sequential logic structure**
- **3. Discuss solution testing and documentation**
- **The logic structure of a program enforces the sequence of execution of instructions in the program and the main logic structures are:**
    - **sequential logic structure and function calls**
    - **Decision logic structure and**
    - **Repetition logic structure**

Program Logic Structures

- Thus, to provide a good solution to any problem, we should proceed as follows:
- 1. Use top-down design approach when necessary.
- 2. For defining both the control module and the functions in the solution, use the relevant structure(s) among the three program logic structures:
  - a. Sequential structure (executing instructions. one after the other)
  - b. Decision Structure (executes one of many alternative instructions.)
  - c. `Repetition Structure (executes a set of instructions. many times

Program Logic Structures

- 3. Eliminate duplication of steps in parts of same program by using a module that can be re-used
- 4. Improve readability using proper naming of variables, internal documentation and proper indentation.

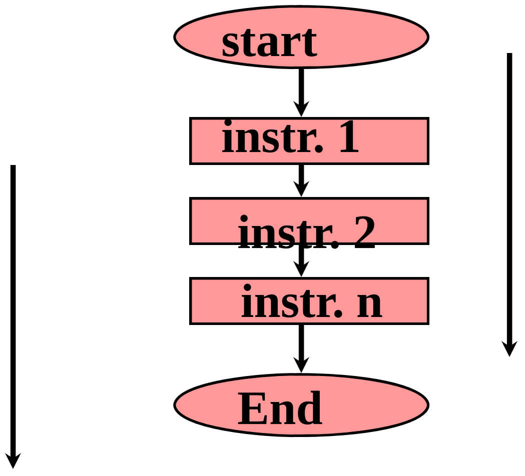
Problem Solving with Sequential Logic Structure

- **Sequential logic structure is the most common and simplest structure**
- **Sequential structure asks the computer to process a set of instructions in sequence from top to the bottom of an algorithm.**

Problem Solving with Sequential Logic Structure

- This is the default structure and all problems use this structure in possible combination with other structures
- #include <stdio.h>

```
int main (void)
 {
     Input  variable list ;
     Output variable list;
     Instruction 1;
     Instruction 2;
          :
     Instruction n;
     return  0;
 }
```



- Execution flow is Instruction1, followed by 2, 3, 4, etc.

Testing the Solution

- Testing the algorithm or program entails selecting test data to check the correctness of the algorithm/program.
- With the test data, stepping through the program should give the expected results
- Test data should be selected to test all possible situations that may arise (e.g. -ve, 0, +ve)
- Program testing entails pre-computation of correct result first, followed by hand simulation or tracing of the program to obtain the result produced by the program, which should be the same as the correct result.

Internal and External Documentation

- Internal documentation are remarks written with the instructions to explain what is being done in the program
- External documentation are manuals written for the user to know how to use the program
- Objective of internal documentation is to make program easily readable, maintainable and expandable by either the original programmer or another programmer.
- It includes
  - the input, output and processing information

Internal and External Documentation

  - Variable usage, writer of the program and
  - other acknowledgements
- Objective of External documentation is to make program easy to use.
- In solving problems, experienced problem-solvers use the sequence of steps:
  - 1. The Structure Chart
  - 2. The Algorithm or the flowchart and/or
  - 3. The program.
- Thus, a problem solver can go straight to step 3 or get to step 3 through step 1, or through both steps 1 and 2. Ultimate solution is 3 (the program).

Problem Solving with Sequential Logic Structure

- Check Examples 5.1 and 5.2 in the book showing programs with only sequential logic instructions, use of built-in functions, and selecting test data to evaluate all possible paths of a program.
- Other important parts of program solution shown by example 5.2 are use of internal documentation (comments) for making programs readable and maintainable as well as external documentation (e.g., user manual to specify how the program should be executed) and the type of input and output data it takes and prints.

6. Problem Solving with Decisions

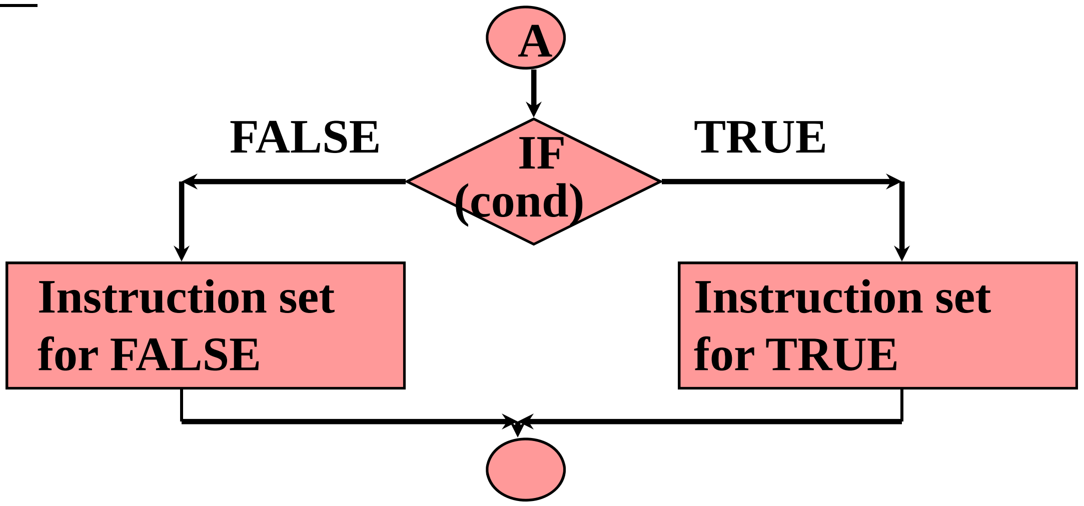
- Objectives
- 1. Discussing Problem solution using both Sequential and Decision logic structures.
  - if/else and switch_case instructions

- The Decision Logic Structure has two main instructions - the if instruction and the switch_case instruction.
- <u>The if/else instruction</u>
- Meaning is IF the condition is true, we execute the TRUE part (a set of instructions), else (that is, condition is false), we execute the ELSE part (another set of instructions)

Problem Solving with Decisions

```
– if   (condition(s))
            {
               TRUE instructions;     }
            else
               { FALSE instructions; }
```

–



Decision Structure

Problem Solving with Decisions

- **Example 6.1: Write decision instruction to indicate if a given integer number is an even number.**

```
if ((Num%2) == 0)
        printf(" The number is an even number\n");
        else
        printf(" The number is an odd number\n");
```

Problem Solving with Decisions

- Example 6.2: A retail store allows part-time workers a rate of $8.00 an hour for a maximum of 20 hours of work in a week. However, a part-time worker earns $10.00 an hour for each additional hour over 20 hours. Write a decision instruction to compute a given part-time worker's wage for a week.

Decision Logic (If Instructions)

- The if form
  - Process all decisions sequentially one after the other (no ELSE part of the instructions)
  - if (condition1) (Draw the flowchart)
    Instructions_T1;
  - if (condition2)
    Instructions_T2;

Decision Example

- Example 6.3: Solve the problem of Example 6.2 using straight through if instructions (with no else part).
Num_hours	Week_wage
hours <= 20	8 * hours
hours > 20	(8 * 20) + (10 * (hours-20))

Decision Example

- Comparing solution versions 6.2 (if with else part) and 6.3 (if with no else part) of the same problem, which is more efficient and why?

if INSTRUCTION

- Now, when is it efficient to use the if form (with no else part) of the decision structure?
- With some problems where the sequence of tests are to be conducted on different variables, the only solution is the straight through if structure (with no else part).
- Example 6.4:
  - Assume you want to assign a number of students (S), to different classrooms for an exam such that each room takes only 150 students and once you have got 150 students for one room you initialize S back to 0. Similarly, the GA's (G) for supervising

if/then INSTRUCTION

- the exams are assigned 10 to each room. Once you have got 10 GA's assigned, you initialize G back to 0. Write decision instructions for initializing both S and G to 0.

Nested if/else Form

- Nested if/else form is the if instruction where either the “TRUE” sequence of instructions (first part) or the “FALSE” sequence of instructions (“else” part), or both sequences contain another “if” instruction.

Nested if/else Form

```
if      (decision expression 1 is true )
        if          (decision expression for expression1-true is true)
                    instructions for when expression of expression1-true is true;
                    else
                    instructions for when expression of expression1-true is false;
        else
                    if          (decision expression for expression1-false is true)
                       instructions for when expression of expression1-false is true;
                    else
                      instructions for when expression of expression1-false is false;
```

- **This structure has if instruction in both the True and else parts.**

Nested if/else Form

- Example 6.5:
  - In a city, the monthly bus fare for seniors 65 years or older is half the normal rate of $45.00 for adults while fare rate for kids under the age of 18 is one_third the normal rate. Write an IF instruction to determine what fare to charge a person given his/her age.
- Solution
  - Conditions(Age) Actions(fare)

Conditions(Age)	Actions(fare)
Age >= 65	1/2 * 45
65 > Age >= 18	45
18 >Age	1/3 * 45

Nested if/else Form

- For problems involving nested if instructions in only the Else or True part, they can be expressed in two ways, namely:
  - 1) Using positive logic, and
  - 2) Using negative logic.
- Positive logic writes the instruction such that some action (like assignment instruction) is executed if decision expression evaluates to TRUE but another IF instruction is executed when decision evaluates to FALSE

Using Positive Decision Logic

(Draw the flowchart)

```
if (Age >= 65)
        Fare = (0.5) * 45.0;
else
       if (Age >= 18)
                Fare = 45.0;
           else
                Fare = 0.33 * 45.0;
```

Using Negative Logic

- Process a set of instructions when “if expression” evaluates to FALSE but process another decision instruction when “if expression” evaluates to TRUE.
- Can use negative logic to decrease the number of tests
- The Age example with negative logic
- if (Age < 65) (Draw Flowchart)

```
if  (Age < 65)                              (Draw Flowchart)
        if  (Age < 18)
                  Fare = (1/3) * 45.0;
           else            Fare = 45.0;
        else
                  Fare = (1/2) * 45.0;
```

Logic Conversion

- **May help improve on efficiency or readability of a solution**
- **E.g., a decision should always have instructions for the TRUE section but not necessarily for the FALSE section.**
- **A solution with no instructions for the TRUE section is better converted to negative logic.**
- **How? To convert from positive to negative logic do the following:**

Logic Conversion

- 1. Write the opposite of each relational operator in every decision as:
  - operator / opposite

operator	opposite
<	>=
<=	>
>	<=
>=	<
==	!=
!=	==

- 2. Interchange all the TRUE set of instrs. with the corresponding ELSE set of instrs.

Logic Conversion

- Example 6.6
  - Calculate the number of bonus air miles earned given that the bonus air miles earned by customers is 100 if traveled miles exceed 5000 in a period of time, but 60 bonus air miles are earned if traveled miles only exceed 3000 while 10 bonus air miles are earned otherwise by the customer. Write two positive logic if program/algorithmic solutions for the above problem. Then, write 2 negative logic solutions.

Logic Conversion Example

Which if Logic to Choose?

- Choose the if logic that is most efficient, and most readable
- Most efficient logic is characterized by
  - 1. Fewer tests both when you know about data and when you don't
  - 2. Easiest to maintain (modify)
- In the above example, solutions 1 and 2 (positive logic) are most readable with same number of tests. So, any of the two may be chosen.

switch_case Instruction

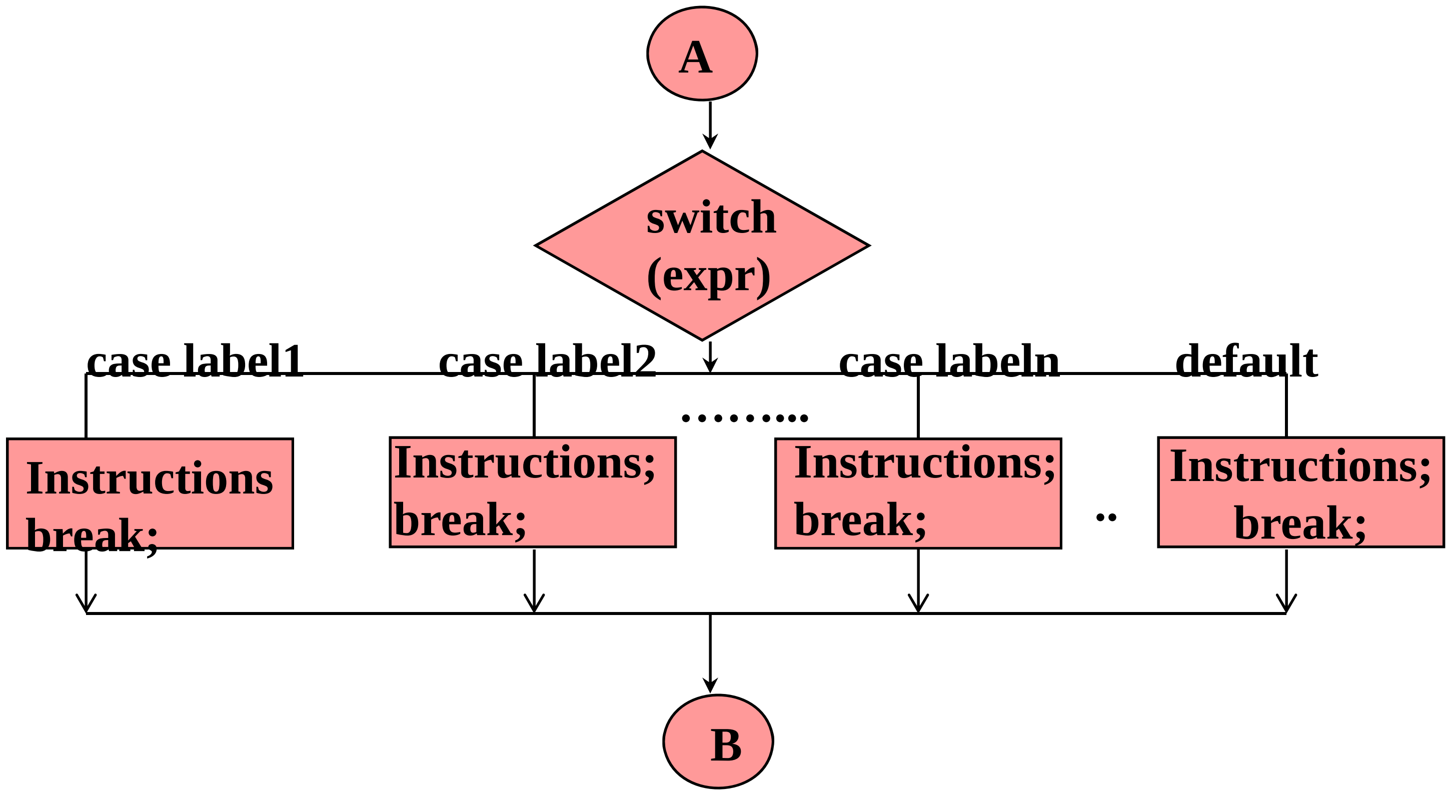
- The switch_case Instruction is the second type of instruction with the decision logic structure.
  - It is made up of several or more sets of instructions, only one of which will be selected if a case label matches the label for that set of instructions.
  - switch_case instruction is used to decide which one execution path among many to choose, while IF instruction chooses one path out of two alternatives.
- Format of switch_case instruction is given below:

switch_case Logic Structure

- switch (EXPRESSION) {

```
        case label1:instructions to execute if expression = labe11;
                      break;
        case label2: instructions to execute if expression = labe12;
                      break;
          :
        case label*n*: instructions to execute if expression = labe1n;
                      break;
        [default: instructions to execute if expression
              matches none of labels 1 to n above;
                      break;]
        }
```

switch_case Logic Structure



FLOWCHART Diagram for the switch_case Logic structure

switch_case Logic Structure

- Give the if instruction Equivalent to the switch_case instruction.

switch_case Instruction Example

- Example 6.7: Write a program that keeps track of the number of houses in each of the five zones labeled A, C, K, L, Q in a city. It reads the zone of a given house, increments the appropriate zone count and prints the number of houses in each zone.

switch_case Instruction Example

swich_case Instruction Example

7. Repetition Logic Structure

- Objectives
- 1. Use three types of Loop instructions in problem solving (while, do-while, for instructions)
- 2. Use nested loops in problem solutions touching on recursion as well.
- The Repetition Logic Structure
- Repetition logic structure allows a sequence of instructions to be executed continuously as long as a condition is satisfied.
- E.g. loop problems: Counting, accumulating sum
- 3 types of loop instructions are used:

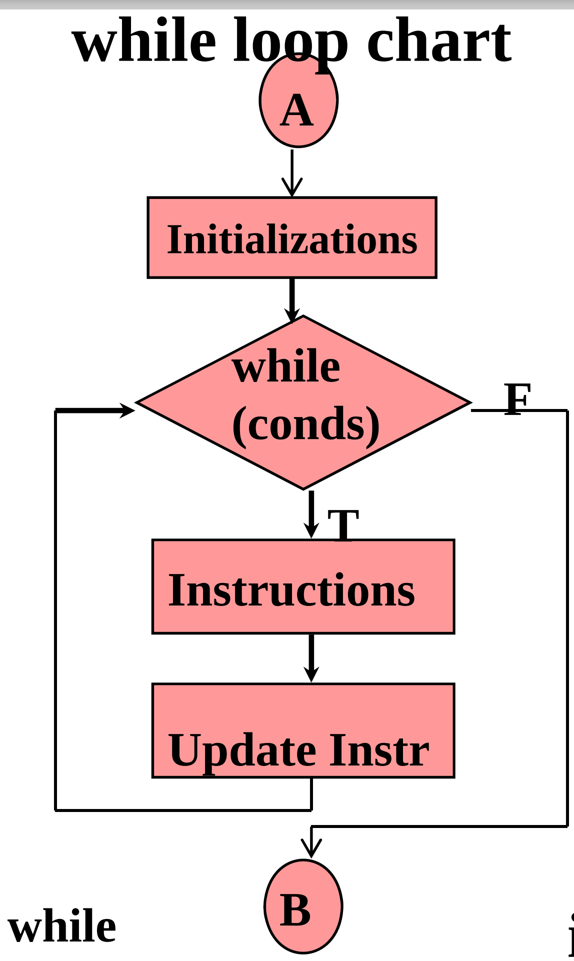
while Instruction

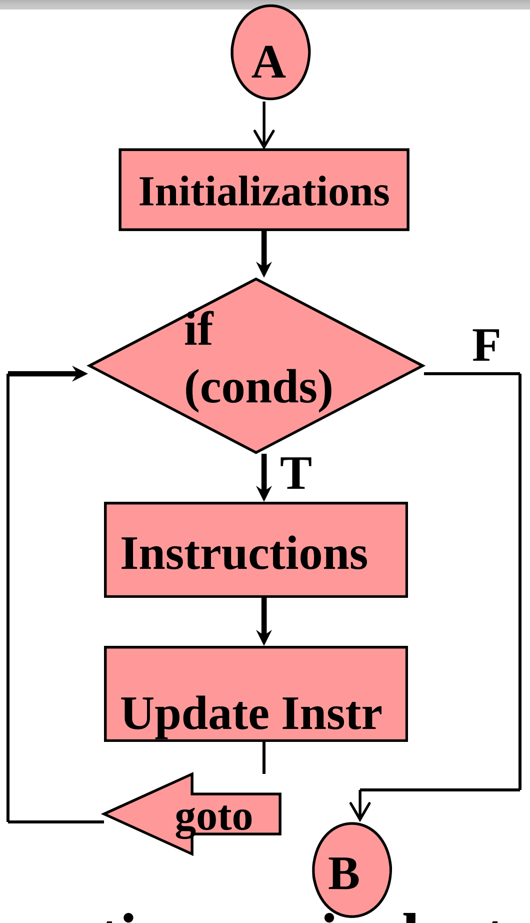
- 1. while Instruction
- Tells the computer to (a) test a <condition> and while that condition is true (b) to repeat all instructions between the while (begin) bracket “{“ and (end) “}”.
  - Initialization instructions;

```
while (condition(s))
{
                INSTRUCTION;
                INSTRUCTION;
                             :
                Update Instructions;
}
```

while Instruction

while loop chart





if instruction equivalent of while

while Instruction

- **While loop can be used for both event-controlled and counter-controlled loop**
- **Important parts of a loop structure are:**
  - **1. Initialization of variables (control and accumulation variables)**
    E.g., count = 0; and Sum = 0
  - **2. Testing of the control variables (for termination condition). E.g., while (count < 10)**
  - **3. Updating the control variable (to advance to next data item). E.g., count ++**

while Instruction (counter controlled)

- Example 7.1: A class of ten students took a quiz. The marks (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz using a program. [Complete this rough solution]
  - 1. Declare Variables
  - 2. Init_Module {Sum=0, Counter = 0}
  - 3. while (Counter < 10)
    {
    - Read_Module(mark, sum)
    - /*reads and adds to sum */

while Instruction Example Problem

    - Counter = Counter + 1
    - }
- 4. Finish_Module /* does the following instrs. */
    - average = sum/10
    - Print_Results_Module (average)
- 5. } /* end of main program*/

- Note that the ReadModule and Finish_Module need to be defined properly in a full program.

while Instruction With Sentinel

- Example 7.2: Write a program that counts the number of houses belonging to each of the five zones A, C, K, L, Q in a city. The zone of each house is entered for reading and a sentinel value of ‘0’ is used to mark the end of data.

while Instruction Example Problem

while and EOF marker

- Example 7.3: Write a program that counts the number of houses belonging to each of the five zones A, C, K, L, Q in a city. The zone of each house is entered for reading and the last data line is marked with end-of-file marker.

while and EOF marker

(2) do-while

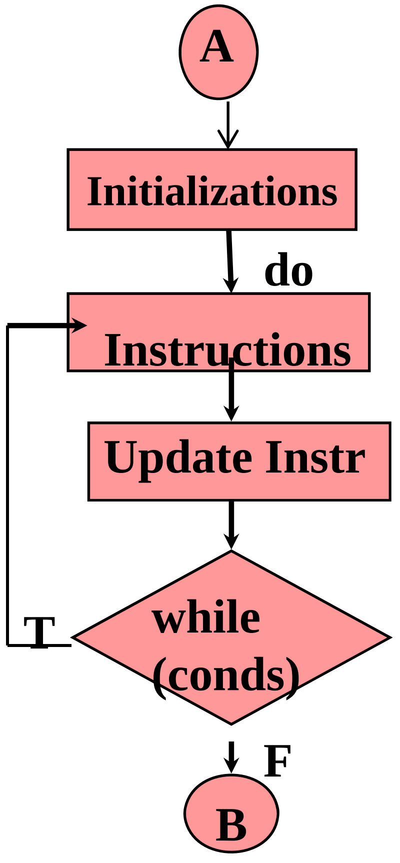
- Tells the computer to repeat the set of instructions between the “do” and the “while” keywords as long as the condition is TRUE.
- Differences between “do-while” and “while” instructions
  - 1. Test for loop termination condition is done at the beginning with “while” loop but at the end with “do-while” loop.
  - 2. With the “do-while”, the loop must execute at least once, with the “while” loop, zero iteration is possible.

(2) do-while

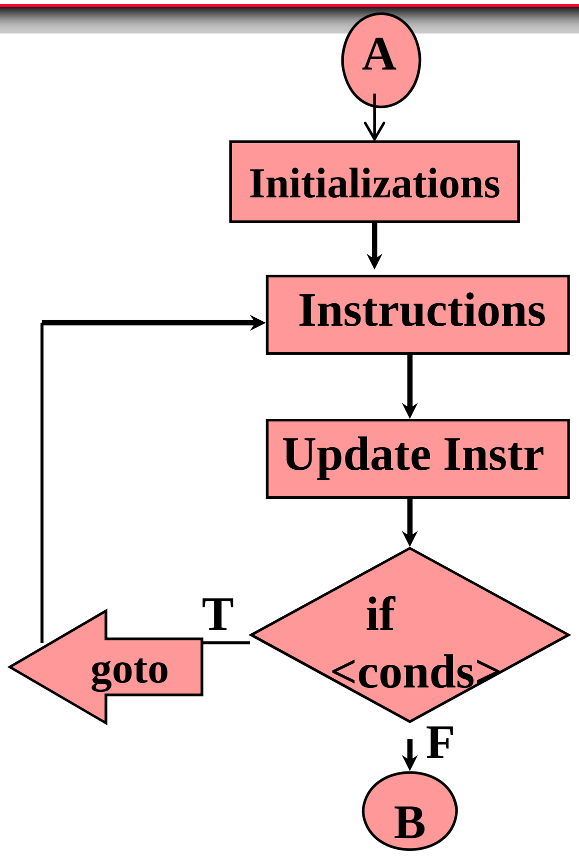
- With “do-while” loop, the loop instructions are processed at least once before the termination condition is tested.
- Thus, for problems that may need zero iterations (number of times the loop is processed), “do-while” loop should not be used (E.g., no data read)
- Format is:
  - Initializations;

```
Initializations;
do {
  Instruction;
  Instruction;
    :
  update instruction
} while (CONDITION(S));
```

(2) do-while



REPEAT/UNTIL



IF (DECISION) EQUIVALENT

(2) do-while

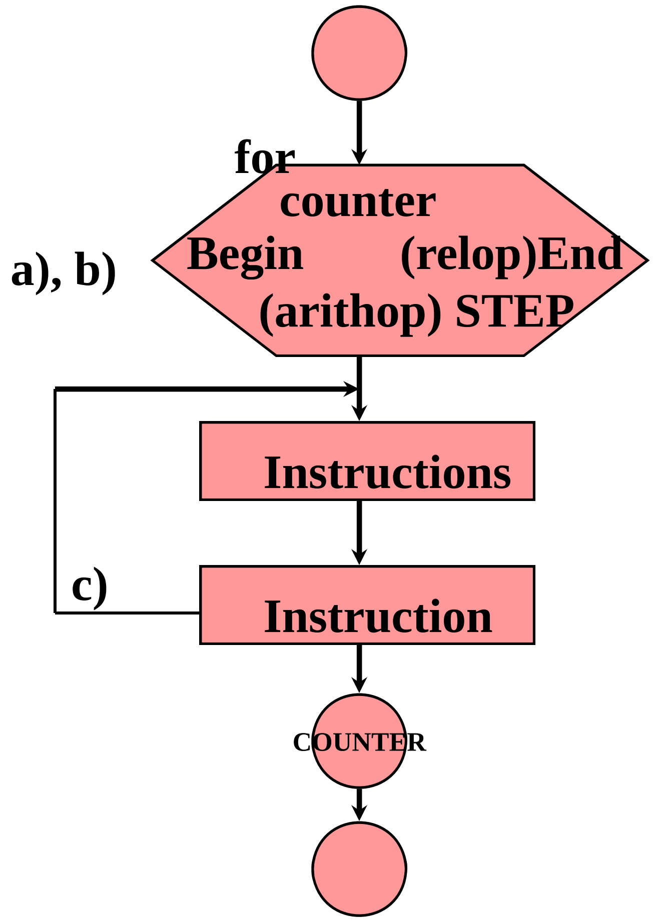
- Example 7.4: A computer class took a quiz. The scores (integer in range 0 to 100) for this quiz are available to you and the last data line is marked with a sentinel value of -1. Determine class average using the do-while loop structure.

do-while

(3) for Instruction (Automatic Counter Loop Control)

- This "for" instruction decrements or increments the control variable each time the loop is repeated
- e.g. for loop
- The initialization, termination value, testing and update of the control variable all occur in the one loop instruction
- Format:
  - for (counter = begin value; counter (relational_operator) end value;
  - counter=counter (arithmetic operator) step)
  - {
  - instruction 1;
  - instruction 2;
  - :
  - instruction n;
  - }

(3) for Instruction (Automatic Counter Loop Control)



Show the Decision Equivalent

(3) for Instruction (Automatic Counter Loop Control)

- **Example 7.5: A computer class of 15 students took a quiz. The scores (integers in the range of 0 to 100) for this quiz are available to you. Determine the class average on the quiz with a program.**

(3) for Instruction (Automatic Counter Loop Control)

Nested Loops & indicators

- Nested loop instructions are loop instructions inside an outer loop
- Nested loops do not have to be same types of loop structures
- Event-controlled loops, programmed with while & do-while loop structures make use of *indicators*.
- Indicators are logical/control variables set inside a program when a certain condition occurs (e.g., end-of-file or no more data in the file, or an error occurs like invalid data)
- Indicators are also called flags, switches, dummy or trip values

Example Problem With Nested Loop

- Example 7.6: Write a program and a flowchart that utilizes nested looping to produce the following table of values:

A	A+2	A+4	A+6
3	5	7	9
6	8	10	12
9	11	13	15
12	14	16	18
15	17	19	21

Example Problem With Nested Loop

Example Problem With Nested Loop

Example Problem With Nested Loop

- Example 7.7: Write a program and a flowchart to display the following checkerboard pattern using nested looping.

```
  *  *   *   *  *  *  *  *
     *  *   *   *  *  *  *  *
 *  *   *   *  *  *  *  *
     *  *   *   *  *  *  *  *
```

Example Problem With Nested Loop

Example Problem With Nested Loop

Recursion

- Recursion is a type of loop structure where a module or a function calls itself
- Some problems are naturally recursive, e.g., factorial
- A recursive solution should have a base case for termination
- Any problem that can be solved recursively can also be solved iteratively
- Recursive approach carries more overhead in terms of memory space needed during execution and processor time.

Recursion

- So, why use Recursion?
  - For some problems that are naturally recursive, providing and maintaining a recursive solution is easier
  - In terms of performance, recursive solution takes longer and consumes more memory
- Example 7.8: write a recursive program to obtain 10!

Recursion

Recursion

8. Arrays

- Objectives
- 1. Develop problem solution using a more complex data structure, arrays which will enable table look-ups, sequential and others.
- 2. Discuss string processing and functions.
- If we want to store 5 assignment marks for a student, we can create the variables Asn1, Asn2, Asn3, Asn4, Asn5
- If there are 10 to 20 assignments to record, this approach becomes clumsy

Arrays

- If we also have to record a matrix of 20 assignment marks for say 100 students, with a different variable for each assignment mark, we have:

  Stu1_Asn1, Stu1_Asn2, ……., Stu1_Asn20
  Stu2_Asn1, Stu2_Asn2,…….., Stu2_Asn20
  :   :   :   :
  Stu100_Asn1,Stu100_Asn2,….,Stu100_Asn20
- Approach is inconvenient and prone to error
- A way out? ---- Use ARRAY data structure

Arrays

- An array data structure allows us to use the same variable name for the 20 assignment marks for one student.
- Using an array, we replace Asn1, Asn2, ……,Asn20 with a single variable Asn subscripted as: Asn[1], Asn[2], .., Asn[20].
- Thus, an array is a data structure allowing more than one memory location to be designated for a single variable.
- Each element of the array variable is referenced using its subscript

Arrays

- Arrays are useful for many data values of the same type, e.g., all ages, all grades etc.
- Arrays are easier to read and use in program statements than having different variables.
- To use arrays in a program, they have to be declared and the size of the array (number of elements) needs to be included. Format is shown below:
- E.g., to declare a one-dimensional array in a program

```
        datatype  arrayname[size];
e.g.,  int  assn1[7];
```

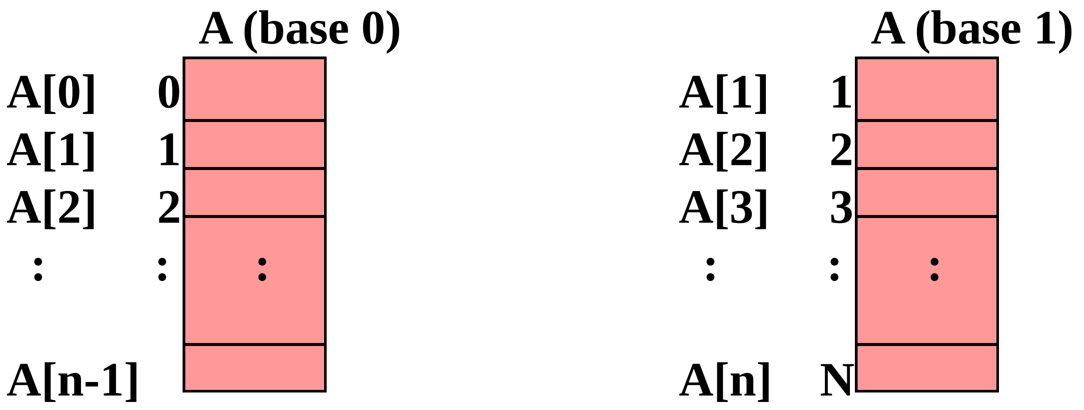
- ❑ Two ways to declare the size or dimension of an array

Arrays

- 1. Static Arrays: allowed by many programming languages.
  - Size and dimension declared at the beginning and never changes during the execution of the program
- 2. Dynamic Arrays: Number of array locations is a variable which can be increased or reduced during the execution of the solution (using malloc in C).
  - More flexible but more time consuming during program execution

Arrays

- The first array element (the base element) is numbered zero (has subscript 0) in some languages like C, but numbered 1 in some others.
- If the base element is 0, the second element is 1; and if the base element is 1, the second element is 2.



Arrays

- By using the assignment instruction, we can assign the value of a constant, a variable, or an expression to an element.
- One Dimensional array is the simplest array structure. Conceptually, a one dimensional array represents an array variable that has only one column of elements.
- E.g. of a one dimensional array: 10 assignment marks for student Maggie

Arrays

- **Example 8.1: Read and print the marks for 10 assignments obtained by student Maggie as well as her average assignment marks.**

Arrays

Array Variables in Functions

- **An array variable can be used any where any simple variable can be used in all types of instructions including function calls but a complete array cannot be returned using a function return value.**
- **An array parameter passing all elements of the array, in a function call simply includes the name of the array variable without specifying the dimension or size.**
- **However, an array's dimension needs to be specified in the function prototype and function header. Its size may also be specified if passed as a parameter in the function call.**
- **E.g., a function ReadData reads data into a 1-dimensional array of seven assignment marks. The function prototype and header for ReadData are respectively:**
- **void ReadData(int [], int);**
- **void ReadData(int assn [], int size)**

Arrays (one-dimensional parallel)

- **Example 8.2: Write a program that computes the assignment average for assignments 1 and 2 in a small class of seven students whose names and ids are Maggie (id 1050), John (id 1051), Ken (id 1052), Joy (id 1053), Pat (id 1054), Tim (id 1055) and Tom (id 1056). The program reads their ids, computes and prints the average mark obtained by each student id as well as asn1 and asn2 averages.**

Arrays (one-dimensional parallel)

Arrays

- Asn1[1] and Asn2[1] both relate to Student[1]; and Asn1[5] and Asn2[5] both relate to Student[5]
- To declare these three arrays, we use
  int Student[7], Asn1[7], Asn2[7];

Array Example

Array Examples

Array Examples

- Example 8.3: We want to use arrays to summarize the results of data collected in a survey. Forty students were asked to rate the quantity of the food in the student cafeteria on a scale of 1 to 5 (1 means awful and 5 means excellent). Place the forty responses in an integer array and summarize the results of the poll using a C program.

Array Examples

Array Examples

Two-Dimensional Arrays

- While a one-dimensional array has only one subscript indicating the number of rows, a two-dimensional array has two subscripts indicating (number of rows, number of columns).
- A two dimensional array can be used to store a table of values with more than one column (e.g., a Matrix).
- To declare a 2-dimensional array, use:

  datatype   arrayname[num_row][num_column];
- The two parallel arrays for Asn1[7] and Asn2[7] we defined earlier on, can be stored in one two dimensional array as:(write answer here)

Two-Dimensional Arrays

- **Example 8.4: Write a program that computes the assignment average for assignments 1 and 2 in a small class of seven students named Maggie, John, Ken, Joy, Pat, Tim and Tom. The average mark obtained by each student is also computed and printed. Solve using two dimensional array where necessary.**

Two-Dimensional Arrays

Two-Dimensional Arrays

Two-Dimensional Arrays

Multidimensional Arrays

- These are arrays with three or more dimensions
- With three dimensional array, three subscripts are needed and three nested loops are used.
- An example of a 3 dimensional array is given in the course book section 8.2
- int Cube[row][column][depth];

String Processing

- **A string in C is an array of characters declared as:**
  char variable[number of characters];
- **The last character of the string is the null character '\0'**
- **Thus, a string with 20 characters has the 20th as '\0'**
- **E.g., char studentname[20] can hold only one student name with up to 19 alphanumeric characters.**
- **Now, if we want to declare a variable to hold 10 student names, it is declared as a 2-dimensional array:**
  char studentname[10][20];
- **Names can also be initialized at declaration as:**
- char studentname[10][20] = {"John Smith", "John Adams", "Mary Goods", "Peter Kent", "Chu Lee", "Paul Best", "Okee Ndu", "Pat Madu", "Andrew New", "Mark Ogods"};

String Processing

- Library functions for string input and output include:
- gets (stringvariable);
- fgets (stringvariable, length, filepointer);
- puts (stringvariable); fputs(stringvariable, length, filepointer);
- sscanf(string_to_readfrom, format specifiers, variablelist);
- sprintf(string_to_printto, format specifiers, variablelist);
- Library functions for string copying, concatenation, comparisons and others include:
- strcpy(s1, s2) , strncpy(s1, s2, numchars) , strcat(s1,s2) , strncat(s1,s2,n) .
- The list of string functions in C library <stdlib.h> for I/O are summarized in section 8.4, while functions for copying and other operations are summarized in section 4.3 of book.

Searching or Table LooKup Techniques

- Searching is one important application of arrays.
- Searching entails using a value to look up another value in a table of values. For example, 100 test scores are stored in an array score[100] and you want to answer the question regarding whether there is any 96% in the 100 scores.
- You can go about this look-up in two ways
    - 1. Sequential Searching
    - 2. Binary searching
- Example 8.5: Given n test scores and a search key score, write a sequential search program to return the position of the first element in the array equal to the key score.

Sequential Search

- The program for sequential search is:

```
• int main (void) {
• int  Score[10], key , k, I, n=10;
• scanf ("%d", &key);
  for (I = 0; I < n; I++)
     scanf("%d", &score[I]);
•  k = 0;
•     while (key != score[k] && (k < n))
                 k = k + 1;
   If (k>=n) printf ("element not found");
       else printf("element %d is equal to key %d", k, key);
   return  0;
   }
```

Sequential Search

- Works well for small or unsorted arrays. Inefficient for large arrays
- In the worst case, the algorithm will search through all n elements of the array before either finding the value or not finding it at all
- In the best case, the algorithm searches through only 1 element

Binary Search

- Binary search is faster, but only works on sorted arrays as it eliminates half of the elements in the array being searched during each iteration.
- Binary search compares the mid-element of remaining array list to the search key.
  - 1. Set the lower boundary at 0
  - 2. Upper boundary is set as the number of elements in the array minus 1 (that is the last element)

Binary Search

- 3. The loop is started and will continue as long as mid element if not the search key and the end of the list is not yet passed. That is while ((test[mid] != key) && (LB <= UB)). If LB > UB, it indicates the last element has been searched.
- 4. The mid-element number is calculated (truncated to integer value) as mid = (LB+UB)/2
- 5. Also, upper and lower boundaries are re-calculated. If search value is greater than value of mid element number, then lower boundary is set to one more than the midpoint, otherwise it is set to one less. (see solution 8.5 for details)

Binary Search

- 6. Once the search loop has ended, test to know whether the search key was found or not. If the lower boundary is greater than the upper boundary, it means element could not be found.

Binary Search Algorithm

Binary Search Algorithm

Binary Search Algorithm

Binary Search Algorithm

- **Example 8.6: Assume the following 10 test scores are sorted in an array test[10], find if a score of 84 is in this array using binary search.**

Test(k)	**-56**	**63**	**65**	**71**	**72**	**75**	**80**	**81**	**84**	**86**

Sorting Techniques

- Sorting is the process of putting the data in alphabetical or numerical order using a key field
- primary key is the first key by which data in a file is sorted, e.g., area code for a mailing list
- Secondary key is the second key by which data in a file is sorted within the primary key order.
- E.g., a mailing list sorted by area code can again be sorted in alphabetical order of name within each area code.

Sorting Techniques

- Sorting techniques include
  - 1. Selection Exchange Sort, bubble Sort, Quick Sort, Shell Sort and Heap Sort
- Best sorting techniques are determined by the number of comparisons and switches that take place for a file of n records in a specific order.

The Selection Exchange Sort

- To sort n records
- Maintain 2 sublists within n records
  - 1. List of sorted part (S)
  - 2. List of unsorted part (U)
- Initially number of elements in S=0 and number of elements in U = n
- 1. Find the smallest element in U and switch its position with the first element of U
  [now number of elements in S=1 and number of elements in U = n-1]

The Selection Exchange Sort

- 2. While number of elements in (U) > 1
  - Find the smallest element U and switch with first element of U. [Once switched this first element of U becomes the last element of S]
- Example 8.7: sort the following in ascending order using selection exchange sort
- 56
  80
  75
  63
  58
  79

The Selection Exchange Sort

```
int main(void) {
int  score[6], num_score, I, minpos, j, temp;
   for (I=0; I < num_score-1;  I++)
   {
       min = I;
       for (j = (I+1); j < num_score; j++)
       {
           if (score[min] > score[j])
                   min = j
       }           /* end of for j */
```

The Selection Exchange Sort

```
▪   if (min != I)
      {
          temp = score[I];
          score[I] = score[min];
          score[min] = temp;
      }
  }                  /* for I    */
  return    0;
}          /* of main */
```

The Bubble Sort

- Example 8.8: Sort in ascending order with bubble sort
- To obtain the S list from the U list, compare each element in U with the next element and switch if element is larger than next one
- 56
  80
  75
  63
  58
  79

The Bubble Sort Algorithm

```
int main (void) {
     int  score[6], numscore=6;
     int  temp, numleft ;
   for (numleft=numscore-2; numleft >= 0; numleft- -)
    {
     for  (j = 0;  j <= numleft;  j++)
          {
            if (score[j] > score[j+1])
             {       temp = score[j];
                     score[j] = score[j+1];
                     score[j+1] = temp;          }
          }  /* end of for j */
    }  /* end of for numleft  */
  return  0;
}    /*  end of main  */
```

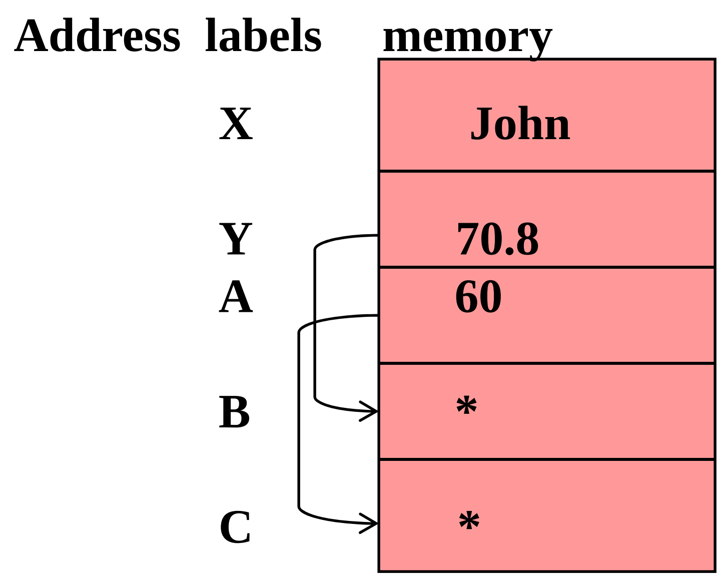
The QuickSort

- Using Quicksort, sort
  56 80 75 63 58 79

9. Pointers, Files, Records and others

- Objectives
- 1. Get introduced to more advanced data structures like pointers, files, records, stacks, linked lists and binary trees
- Pointers
- A pointer is a variable that can store only memory addresses as its value.

Pointers



Here, cells X, Y and A hold ordinary data values while cells B and C hold addresses of variables Y and A respectively.

Pointers

- A pointer variable needs to be declared before use and the format for declaring them is:
- type_of_data_it_points_to   *pointervariable;
- E.g, float *B;
-         int *C;
- Operations on Pointer Variables
  - 1. A pointer variable can be initialized to 0 or null
    E.g., B = 0;
    meaning that it is pointing to nothing but it exists.

Pointers

- 2. A pointer variable can be set to point to a variable by assigning the address of the variable using address operator (&).
  E.g.   B = &Y;
         C= &A;
- 3. We can read or write the data value being pointed to by a pointer variable through the pointer variable by using the indirection or dereferencing operator (*).
  E.g., print (*B)   will display 70.8
   *C = *C + 10 will replace the value in A with 70.

Pointers

- 4. A pointer may be subtracted from another pointer and a pointer may be incremented with an integer value.
- 5. A pointer can be assigned another pointer variable if they both point to values of the same type.

File Concepts

- So far, we have read data from the key board
- If we write a program to process one thousand student records, reading data from the key board, then, every time we need to run the program again, we have to start typing in all one thousand records.
- Approach in this case is inconvenient and prone to error.
- A solution to this problem is to pre-type our one thousand records in a disk file, save it and tell our program to read data from a disk file and not from the standard input device which is the key board.

File Concepts

- A file structure consists of a number of records with each record representing a real life entity.
- A record is made up of a sequence of fields or attributes (e.g., student id, name, major, gpa).
- Records in a file could be accessed either sequentially or randomly.
- Sequential access files store records in some order (usually in primary key order)
- For a file to be used in a program/algorithm, the following steps should be taken:

File Related Program Instructions

- 1. Declare a file pointer variable or logical variable name. That is, declare a pointer variable to point to variable of type FILE. Format is:
- FILE *filepointer;
- e.g., FILE *stnptr;
- 2. OPEN the file: This step associates the file pointer variable with a disk file which is to be opened for either read (r), write (w), update (r+) or append (a). Format for opening a file is:
  filepointer = fopen (“disk file name”, “mode”);
  E.g., stnptr = fopen(“stnrec.dat”, “r”);

File Related Program Instructions

- 3. Read/Write records from/into the file:
  - Read copies the next record from disk file into the internal memory variables for processing
    Format is: fscanf(filePointer, “format specifiers”, variable list to be read);
  - fprintf(filePointer, “format specifiers”, variable list to be printed);
  - fscanf(stnptr,”%s %s %s %f ”, studentid, name, major, &gpa);
- 4. CLOSE the file
  - tells the computer the file is no longer needed. Format is: fclose(filePointer);
  - E.g., fclose(stnptr);

File Related Program Instructions

- FILE END-OF-FILE (feof) Marker with files
- Data files contain feof marker to indicate there are no more data.
- When testing for feof marker in a file include the file pointer as parameter. E.g., while (! feof (stnptr))

- Loop structures can be used to read lines of records from a file sequentially as:

```
K=0;
fscanf(fptr,"%s %s", field1, field2);
while (!feof(fptr)
  {
            K++;
            fscanf(fptr,"%s %s", field1, field2);
  }
```

Record Structure

- A record has many fields identified using one variable name but the fields can be of different data types.
- E.g., record student has fields studentid, name, major (of type string), and gpa (of type real) and can be declared as follows:

```
•  struct        student_type {
•                    char studentid[15];
•                    char name[20];
•                    char major[15];
•                    float gpa;
•         }          /* of student record type */
```

Record Structure

- To declare a variable of record type, we need to first define the record structure type as we have done above for student record type, then secondly, we define a variable to be of this record type.
- To define a variable of student record type, we do:
  - struct student_type   studentvar;
- Now we can assign values to fields of the variable Student as follows:

  -
    ```
    scanf("%s %s %s %f",
    studentvar.studentid,studentvar.name,
    studentvar.major, &studentvar.gpa);
    ```

Record Structure

- Any other valid operations can be performed on these fields of the record (e.g., print, assignment etc.)
- We can also define an array of student records to store more than one student record as follows:
  - struct Record_type record_var[size] ;
  - E.g., struct Student-type Student[100];
  - To print the record for student number 51, we use:
  - printf ("%s %s %s %f", Student[51].name, Student[51].age, Student[51.major, &Student[51].GPA]);
- And to read a record variable from a file, we again specify the file pointer first before listing the fields of the record.
- typedef command can be used to rename a record structure.

Other Data Structures

- Data structure specifies the way data are stored in the computer memory
- Two types of Data Structures are
  - 1. Single Valued data types [or Ordinal types]
    - have ordinal values with a defined preceding or succeeding value
    - E.g. of ordinal types are char, integer, logical type
    - Real numbers do not have ordinal values

Data Structures

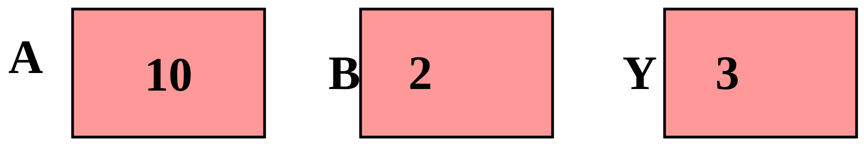
- 2. Structured Data Types
  - Strings, arrays and records
  - each variable has multi-values e.g., an array
- a. Stacks
- A stack is a list of numbers
- All additions and deletions are at one end (top of stack)

Data Structures

- A last-in, first-out procedure
- Operations defined on stack are
  - *Push* to add a value to the top of stack, and
  - *Pop* to delete a value from top of stack
- E.g., trace the following procedure on stack data structure and show the states of the stack before and after the procedure.
- It should be noted that before data is pushed onto stack, there has to be room to hold the data
- Also, before data is popped from stack, there is data available

Data Structures

- Push 4
  Push 5
  Push Y
  Pop A
  push 7
  Push 8
  push B



b. Linked Lists

- A linked list is a data type where each record points to its successor except for the last record
- Each record contains a field (the linking field) that contains the address of the next record in sequence.
- The link field of the first record points to the second record, that of the second record points to the 3rd record, etc. and that of the last record contains zero meaning it points to no record.
- It is easier to add or delete records from a linked list file than an array of records

Linked Lists

- With the linked list, deleted records are placed in an empty list and additions are placed in the records that had been deleted or at the bottom of the file
- Example is Figure 9.1 of course text.

Linked Lists

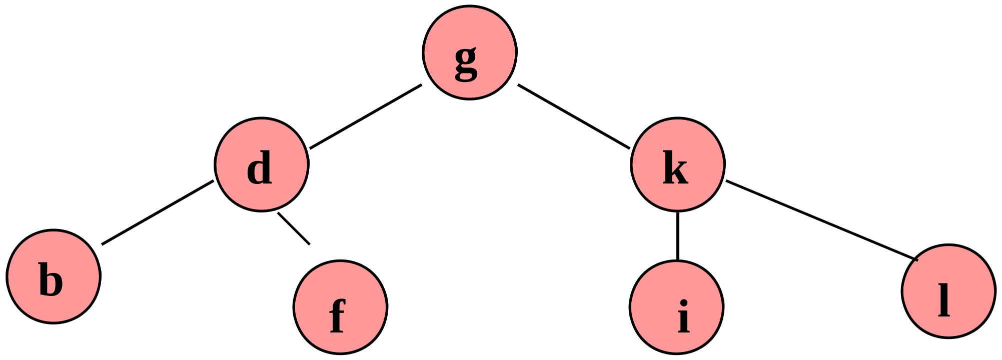
- Record of this type can be declared as follows:
- struct list_type

```
        {char id[15];
         char name[20];
         list_type      *listptr ;
       }
```

- struct   list_type   *listptr ;

c. Binary Trees

- A tree structure uses a top down or hierarchical structure for data
- Each record is stored as a node of the tree. E.g.



Binary Trees

- A parent node is at a higher level. The nodes at a lower level of a node are its children.
- E.g. g is the root node, d is the parent of b and f. b and f are children of d.
- A subtree consists of a chain of nodes.
- A branch is a path from root to leaf.
- A binary tree is a tree in which each node has at most two children
- Each record (a node) contains two link fields, one pointing to the left node (child) and the other pointing to the right child.

Binary Trees (Declaration

- To declare a variable of type binary tree, we do:

```
struct  btree_type {
                    char node_name[20];
                    btree_type     *leftPtr;
                    btree_type     *rightPtr;
          } btree_ptr  (pointer to btree_type)
```

- struct btree_type btree_ptr;

Binary Trees

Binary Trees (Creation Algorithm)

Binary Trees (Traversal Algorithms)

- Records stored as binary trees have to be processed and printed in order
- Processing of these records can be done using tree traversal techniques
- 3 tree traversal algrorithms are used
  - 1. Preorder (N L R)
  - 2. Inorder (L N R)
  - 3. Postorder (L R N)

Binary Trees (Traversal Algorithms)

- E.g. the order of the binary tree in Fig. 14.13 when processed in each of these methods are:
- 1. Preorder: g d b f k i l
- 2. Inorder: b d f g i k l
- 3. Postorder: b f d i l k g