Computer Programming in C/C++
66111
Department of Computer Engineering

Luai M. Malhis, Ph.D.

Spring 2010

Computer Programming (66111)

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

Introduction To Computer System

Computer System

Definition of a computer

- The computer is an electronic machine that performs the following four general operations:

1. Input
2. Storage
3. Processing
4. Output.

Computer Components

- A computer consists of two main components :
- **Hardware** the mechanical, magnetic, electronic, and electrical components making up a computer system
- **Software**: which are written programs pertaining to the operation of a computer system and that are stored in read/write memory.
- Following is an overview of the main hardware and software components in a computer

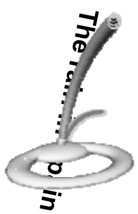
Computer Hardware

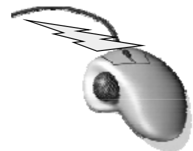
- Input devices
- System unit
- Output devices
- Storage devices
- Processing Unit:
  The CPU and
  Main Memory

Input Devices

- Enter data to be processed
  - Keyboard
  - Scanners
  - Mouse
  - Trackball
  - Touch screen
  - Microphone
  - Game Controller
  - Digital camera

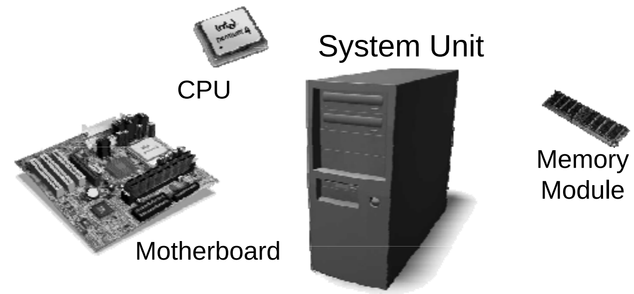




System Unit

- Cabinet that houses all components
- Motherboard
- CPU
- Memory modules

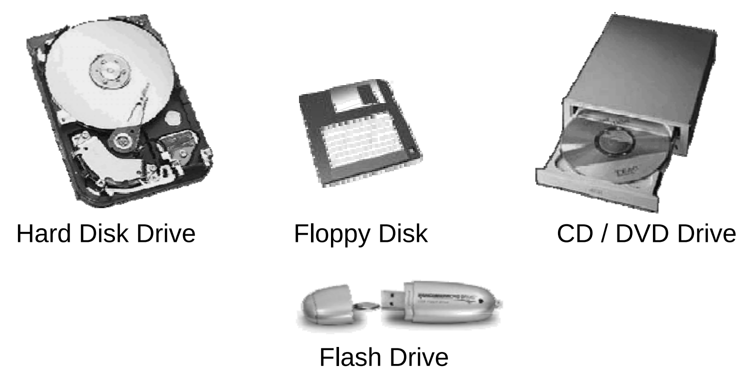


Output Devices

- Enable us to see or hear the processed information
  - Monitor
  - Speakers
  - Printers

Storage Devices

- Enable us to store data or information to be accessed again



The Central processing Unit

- The CPU contains three parts:
- *Arithmetic Logic Unit* - ALU is where the "intelligence" of the computer is located. It performs all arithmetic operations such as addition, subtraction, multiplication and division. The ALU performs logical operations i.e. makes decisions by determining if a number is greater, less, or equal to the other number. An operation completes in nanoseconds, which is a billionth of a second.
- *Registers*: which are small storage devices holds instructions and operands needed by the ALU during operation execution.
- 3. *Control Unit* - This is the part of the unit, which directs information to the proper places in your computer, such as calculation of information by the ALU unit or to store and print material.

The Memory Unit

- The Main Memory:
- Two types of memory contained on a chip are ROM (Read Only Memory) or RAM (Random Access Memory).
- *ROM* memory is installed on a computer by the manufacturer and can not be altered. ROM is the memory that determines all the basic functions of the operation of rge computer such as startup, shut down, and placing a character on the screen.
- *RAM* is temporary memory, which stores programs during execution and also hold all information displayed on the monitor. RAM is read/write memory and it is much larger in size than ROM. Data disappears from the RAM when the computer is turned off or power is off.

Computer Software

- **Software** - programs that enable the hardware to perform different tasks
- Application software
  - Tools for getting things done

Computer Software

- System software
  - Essential for platform operation and support

Computer Platforms: PCs and Macs

PC
- CPU – Intel, AMD
- Operating system – Microsoft Windows

Mac
- CPU – Motorola
- Operating system – Apple Mac OS

Application Software

- Used to accomplish specific tasks other than just running the computer system.
- May consist of a single program, such as an image viewer;
- A small collection of programs (often called a software package) that work closely together to accomplish a task.
- Independent programs and packages that have a common user interface or shared data format, such as Microsoft Office.

Programming languages

- The machine language, which is the only languages understood by CPU. While easily understood by the CPU, the machine language is almost impossible for humans to use because they consist entirely of 0’s and 1’s.
- A assembly language contains the same instructions as a machine language, but the instructions and variables have names instead of being just 0’s and 1’s.
- High Level language Closely resemble human language Examples of high level languages are : Pascal, Fortran, Basic, Java, and C/C++. Programs written in high-level languages are translated into machine language by a compiler.

C/C++ Programming Lnaguage

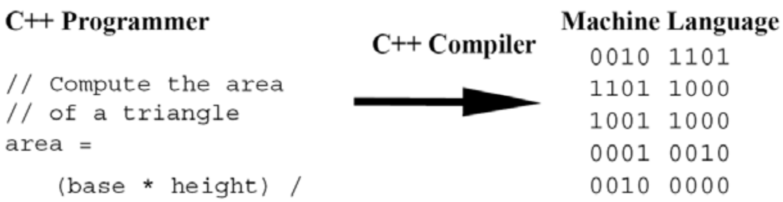
- History of C
  - Evolved from two other programming languages
    - BCPL and B
      - “Typeless” languages
  - Dennis Ritchie (Bell Laboratories)
    - Added data typing, other features
  - Hardware independent
    - Portable programs
  - 1989: ANSI standard
  - The C Lnaguage is Then developed to contain classes and other object Oriented features and named as C++.
  - Many Other Lnaguages currently developed that uses a synatx and symantics like C.
  - C/C++ is traditionally the first language a programmer learns.

How C++ Works

Programs are written by humans.

Programs are run on computers.

C++ programs are written by humans and translates into machine language by the C++ compiler.



What is a program

- A program is a set of instructions that a computer follows.
- Example: computing the Area of Rectangle
  Get base Get Height
  Area = 0.5 * base * height
- Steps to writing a program:
  Step 1. Think! (This is not optional.)
  Step 2. Organize your thoughts
  Step 3. Write them down in English
  Step 4. Translate them into C++

Program Construction

- **Text Editor**

This is used to create the program in C++ form. Since this is the start or source of the other forms this is called a *source file. (Source files end with* .cpp. -- also used C and .cc.)

- **Compiler**

This translates the *source file into a machine dependent file called an object file. The object file contains the instructions in a way that the* machine can understand. The source file is in the C++ language (high level code) while the object file is in machine language (low level code.)

- **Link**

  This is used to associate the object file with other necessary files to generate an EXE file Which contains the machine language.

Data vs. Information

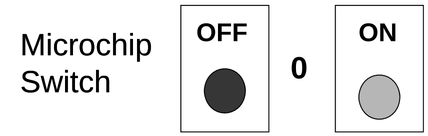
- Data vs. Information:
  - Data is a representation of a fact or idea
    - Number
    - Word
    - Picture
    - Sound
  - Information is data that has been organized or presented in a meaningful fashion.

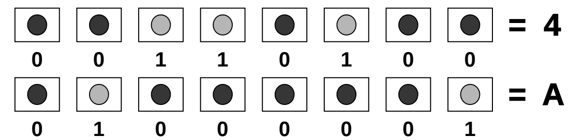
Computers are Data Processing Devices

- Four major functions:
  - Input data
  - Process data
  - Output information
  - Store data and information

Bits and Bytes: The Language of Computers

- Bit
  - Binary digit
  - 0 or 1
- Byte
  - Eight bits
- ASCII
  - Each byte represents a letter, number or special character





How Much is a Byte?

NAME	ABBREVIATION	NUMBER OF BYTES	RELATIVE SIZE
Byte	B	1 byte	Can hold one character of data.
Kilobyte	KB	1,024 bytes	Can hold 1,024 characters or about half of a typewritten page double-spaced.
Megabyte	MB	1,048,576 bytes	A floppy disk holds approximately 1.4 MB of data, or approximately 768 pages of typed text.
Gigabyte	GB	1,073,741,824 bytes	Approximately 786,432 pages of text. Since 500 sheets of paper is approximately 2 inches, this represents a stack of paper 262 feet high.
Terabyte	TB	1,099,511,627,776 bytes	This represents a stack of typewritten pages almost 51 miles high.
Petabyte	PB	1,125,899,906,842,624 bytes	The stack of pages is now 52,000 miles high, or about one-fourth the distance from the Earth to the moon.

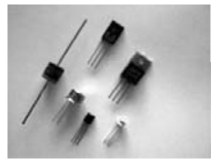
Binary Language

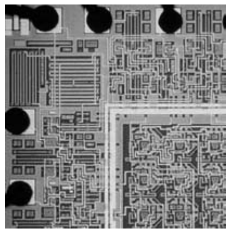
- Computers work in binary language
- Consists of two numbers: 0 and 1
- Everything a computer does is broken down into a series of 0s and 1s
- Switches: Devices inside the computer that can be flipped between these two states: 1 or 0, on or off

Switches

- Non-mechanical devices in computers that open and close circuits
- Types of electronic switches:
  - Vacuum tubes
  - Transistors:
    - Semiconductors
  - Integrated circuits

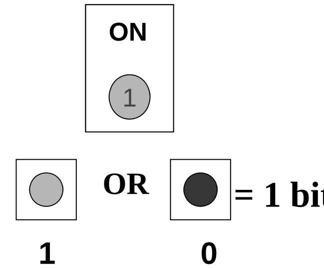
Vacuum Tube


Transistors


Integrated Circuits

Switches Representing Data

- The on/off state of a switch represents one bit of data
- Bit (binary digit)
  - On = 1
  - Off = 0



The Binary Number System

- Describes a number as powers of 2
- Also referred to as base 2 numbering system
- Used to represent *every* piece of data stored in a computer: all of the numbers, letters, and instructions

The Binary Number System

- Number systems are organized ways to represent numbers
- Each number in one system has a corresponding number in another.

	128 2x64	**64** 2x32	**32** 2x16	**16** 2x8	**8** 2x4	**4** 2x2	**2** 2x1	**1**	
Binary	0	1	0	1	1	0	0	1	
Base 10	0 +	64 +	0 +	16 +	8 +	0 +	0 +	1 =	89

01011001 = 89

Binary Base 10

Understanding Decimal Numbers

- Decimal numbers are made of decimal digits: (0,1,2,3,4,5,6,7,8,9)
- But how many items does a decimal number represent?
  - $8653 = 8x10^3 + 6x10^2 + 5x10^1 + 3x10^0$
- What about fractions?
  - $97654.35 = 9x10^4 + 7x10^3 + 6x10^2 + 5x10^1 + 4x10^0 + 3x10^{-1} + 5x10^{-2}$
  - In formal notation -> $(97654.35)_{10}$
- Why do we use 10 digits, anyway?

Understanding Binary Numbers

- Binary numbers are made of binary digits (bits):
  - 0 and 1
- How many items does an binary number represent?
  - $(1011)_2 = 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 = (11)_{10}$
  - $(1010010) = 64+16+2 = (82)_{10}$
  - $(100010001) = 256+ 16+1 = (273)_{10}$

Convert *from* Decimal *to* binary

For each digit position:

1. **Divide decimal number by the base (e.g. 2)**
2. **The *remainder* is the lowest-order digit**
3. **Repeat first two steps until no *divisor* remains.**

Example for $(13)_{10}$:

	Integer Quotient		Remainder	Coefficient
13/2 =	6	+	½	$a_0 = 1$
6/2 =	3	+	0	$a_1 = 0$
3/2 =	1	+	½	$a_2 = 1$
1/2 =	0	+	½	$a_3 = 1$

Answer $(13)_{10} = (a_3\, a_2\, a_1\, a_0)_2 = (1101)_2$

The Growth of Binary Numbers
N is the number of bits in the binary number

n	2^n
0	$2^0=1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
4	$2^4=16$
5	$2^5=32$
6	$2^6=64$
7	$2^7=128$

n	2^n	
8	$2^8=256$	
9	$2^9=512$	
10	$2^{10}=1024$	
11	$2^{11}=2048$	
12	$2^{12}=4096$	
20	$2^{20}=1M$	Mega
30	$2^{30}=1G$	Giga
40	$2^{40}=1T$	Tera

Understanding Octal Numbers

- Octal numbers are made of octal digits: (0,1,2,3,4,5,6,7)
- How many items does an octal number represent?
  - $(4536)_8 = 4x8^3 + 5x8^2 + 3x8^1 + 6x8^0 = (1362)_{10}$
- What about fractions?
  - $(465.27)_8 = 4x8^2 + 6x8^1 + 5x8^0 + 2x8^{-1} + 7x8^{-2}$
- Octal numbers don't use digits 8 or 9

Convert an Integer *from* Decimal *to* Octal

For each digit position:

1. **Divide decimal number by the base (8)**
2. **The *remainder* is the lowest-order digit**
3. **Repeat first two steps until no *divisor* remains.**

Example for $(175)_{10}$:

	Integer Quotient		Remainder	Coefficient
175/8 =	21	+	7/8	$a_0 = 7$
21/8 =	2	+	5/8	$a_1 = 5$
2/8 =	0	+	2/8	$a_2 = 2$

Answer $(175)_{10} = (a_2\, a_1\, a_0)_2 = (257)_8$

Understanding Hexadecimal Numbers

- Hexadecimal numbers are made of <u>16</u> digits:
  - (0,1,2,3,4,5,6,7,8,9,A, B, C, D, E, F)
- How many items does an hex number represent?
  - $(3A9F)_{16} = 3x16^3 + 10x16^2 + 9x16^1 + 15x16^0 = 14999_{10}$
- What about fractions?
  - $(2D3.5)_{16} = 2x16^2 + 13x16^1 + 3x16^0 + 5x16^{-1} = 723.3125_{10}$
- Note that *each* hexadecimal digit can be represented with four bits.
  - $(1110)_2 = (E)_{16}$

Converting Between Base 16 and Base 2

$3A9F_{16} = \underline{0011}\ \underline{1010}\ \underline{1001}\ \underline{1111}_2$

3 A 9 F

- **Conversion is easy!**
  - **Determine 4-bit value for each hex digit**
- **Note that there are 2^4 = 16 different values of four bits**
- **Easier to read and write in hexadecimal.**
- **Representations are equivalent!**

Converting Between Base 16 and Base 8

$3A9F_{16}$ = 0011 1010 1001 1111_2

3 A 9 F

35237_8 = 011 101 010 011 111_2

3 5 2 3 7

Convert from Base 8 to Base 2

1. **Regroup bits into groups of three starting from right**
2. **Ignore leading zeros**
3. **Each group of three bits forms an octal digit.**

Number System Conversion Table

Dec	Bin	Oct	Hex
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Representing Letters and Symbols

- There are codes that dictate how to represent characters in binary format. Most of today's computers use the **American Standard Code for Information Interchange (ASCII code)** to represent each letter or character as an 8-bit (or 1-byte) binary code.
- The ASCII code represents the 26 uppercase letters and 26 lowercase letters used in the English language, along with a number of punctuation symbols and other special characters, using 8 bits. Eight bits is the standard length upon which computers are built.
- In the ASCII The representation for A is 41H (65) , B is 42H (66), a is 61H (97) and b is 62h (98). The Complete table is shown next slide

ASCII Chart

Character	Decimal Number	Binary Number	Character	Decimal Number	Binary Number
blank space	32	$0010\ 0000_2$	^	94	$0101\ 1110_2$
!	33	$0010\ 0001_2$	_	95	$0101\ 1111_2$
"	34	$0010\ 0010_2$	`	96	$0110\ 0000_2$
#	35	$0010\ 0011_2$	a	97	$0110\ 0001_2$
$	36	$0010\ 0100_2$	b	98	$0110\ 0010_2$
A	65	$0100\ 0001_2$	c	99	$0110\ 0011_2$
B	66	$0100\ 0010_2$	d	100	$0110\ 0100_2$
C	67	$0100\ 0011_2$	e	101	$0110\ 0101_2$
D	68	$0100\ 0100_2$	f	102	$0110\ 0110_2$
E	69	$0100\ 0101_2$	g	103	$0110\ 0111_2$
F	70	$0100\ 0110_2$	h	104	$0110\ 1000_2$
G	71	$0100\ 0111_2$	i	105	$0110\ 1001_2$
H	72	$0100\ 1000_2$	j	106	$0110\ 1010_2$
I	73	$0100\ 1001_2$	k	107	$0110\ 1011_2$
J	74	$0100\ 1010_2$	l	108	$0110\ 1100_2$
K	75	$0100\ 1011_2$	m	109	$0110\ 1101_2$
L	76	$0100\ 1100_2$	n	110	$0110\ 1110_2$
M	77	$0100\ 1101_2$	o	111	$0110\ 1111_2$
N	78	$0100\ 1110_2$	p	112	$0111\ 0000_2$
O	79	$0100\ 1111_2$	q	113	$0111\ 0001_2$
P	80	$0101\ 0000_2$	r	114	$0111\ 0010_2$
Q	81	$0101\ 0001_2$	s	115	$0111\ 0011_2$
R	82	$0101\ 0010_2$	t	116	$0111\ 0100_2$
S	83	$0101\ 0011_2$	u	117	$0111\ 0101_2$
T	84	$0101\ 0100_2$	v	118	$0111\ 0110_2$
U	85	$0101\ 0101_2$	w	119	$0111\ 0111_2$
V	86	$0101\ 0110_2$	x	120	$0111\ 1000_2$
W	87	$0101\ 0111_2$	y	121	$0111\ 1001_2$
X	88	$0101\ 1000_2$	z	122	$0111\ 1010_2$
Y	89	$0101\ 1001_2$	{	123	$0111\ 1011_2$
Z	90	$0101\ 1010_2$	|	124	$0111\ 1100_2$
[	91	$0101\ 1011_2$	}	125	$0111\ 1101_2$
\	92	$0101\ 1100_2$	~	126	$0111\ 1110_2$
]	93	$0101\ 1101_2$			

Computer Programming

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

The C Language Basic

C Language Elements

- Key Words - reserved words with special purpose that are part of the C/C++ language
- Programmer Defined Symbols - words or names that have been defined by the programmer. May be variables, or constants.
- Operators - Tell the computer to perform specific operations (ex: +,-, .. > &&).
- Punctuation - begins or ends a statement (;)
- Syntax - grammar rules for writing a C statement.

Some Definitions

- Statement - instruction for the computer to perform, usually ends in a semicolon (;).
- Variable - name given to a memory location that stores data that may change
- Constant - data that does not change like numbers 12, -14, 16.5 or letters 'A'

Programming Errors

- **Syntax errors: violation of the syntax (grammar rules) of the programming language**
- **The compiler gives an error message if the program contains a syntax error**
- **Run-time errors: errors detected when the program is run**
- **The system usually gives an error message during execution for run-time error**
- **Logic errors: program compiles and runs normally, but does not perform properly. Caused by an error in the logic of the program**

Special Characters

Character	Name	Use
//	double slash	to indicate a comment, everything to right is ignored.
/* */	slash asterisk	to enclose a comment
#	pound sign	to indicate preprocessor directive
< >	brackets	to enclose a file name for a preprocessor directive
( )	parentheses	to enclose parameters for a function or change precedence
{ }	braces	to enclose a group of statements
" "	quotes	to enclose a string of characters
;	semicolon	to end a statement

Comments

- Important part of the program.
- Non-executable (not compiled) statements
- Describe the purpose of the program or parts of the program
- Can be indicated by the double slash or the slash asterisk combination
  - // everything to the right of the double slash until the end of the line is ignored
  - /* encloses comment and requires a closing */ to end comment.
- Provide documentation

Programming Process

- Define the problem (most important step).
  - Purpose
  - Input
  - Processing
  - Output
- Design an algorithm (often in pseudo code(English))
- Check logic
- Write code, enter code, compile code
- Correct any syntax errors
- Run code with test data, correct any errors

Example

- Suppose you want to calculate the area of a circle using the radius that a user enters.
- Define the Problem
  - purpose: the program is to calculate the area of a circle for a given radius
  - input: radius
  - process: area = 1/2 pi radius^2
  - output: area
- Algorithm -
  - Display a message asking for radius // cout << "enter radius";
  - Input the radius // cin >> radius;
  - Calculate the area= $0.5\pi r^2$ // area = 0.5 *3.14 raduis*raduis
  - Display the radius and the area // cout << "the area is "<< area;
- Check Logic - does this algorithm fulfill the purpose.
- Write code, enter the code, and compile it.

Variables and Constants

- Data can be stored in RAM (Random Access Memory) to be used as needed.
- Variables and symbolic constants are names for these memory locations.
- Variables refer to memory locations in which the value stored may change throughout the execution of the program.
- Constants refer to memory locations in which the values do not change.
- Use a declaration to set aside memory space.

Identifiers (Variables)

- Identifiers are names (or symbols) used by the programmer to refer to items such as variables, constants, functions.
- Identifiers (variable) should be descriptive of what they stand for. Ex sum, total or area
- The "Name" used for identifiers must follow specific guidelines in C/C++ to be valid.

Identifier Naming Rules

1. The identifier cannot be a keyword, e.g. int, float, if, while, etc.
2. The identifier must be comprised of only letters (A-Z, a-z), numbers(0-9), the underscore (_) and the $
3. The first character must not be a digit
4. C/C++ is case sensitive so total is not the same identifier as Total

Valid and Invalid Names

- X: is valid name
- Xy2: is valid name
- 1class: is invalid because it starts with digit
- Num two: is invalid because it contains space
- For: is valid
- for: is invalid because it is a keyword
- X%y: is invalid because it contains %
- Total_Score: is valid
- area: is valid name
- _Info2for: valid
- @num: invalid

Data Types

- To allocate the memory space for a variable you must state the type of data that is being stored as well as the identifier.
- The classification of data types:

1. Integers (whole numbers),
2. Real numbers (with fractional parts)
3. Characters (ASCII code) may be letters, numbers or any other symbol.

Key words for Data Types

- In C/C++ There are 6 basic keywords used to define variables of the different data types

Integers: Example (decleration)

- **short -** integer (size 2 bytes) // **short** x;
- **int** - integer (size 4 bytes) // **int** y2;
- **long** - integer (size 4 bytes) // **long** abc;

Floating Points:

- **float** - floating point value: i.e. a number with a fractional part. (size 4 bytes) // **float** area;
- **double** - a double-precision floating point value. // **double** w;

Symbols(letters):

- **char** - a single character. (size 1 byte) // **char** z;

-----In this class, we will mainly use **char**, **int** and **double** when declaring variables.

Declarations

- Declarations are statements that tell the computer to allocate memory space and the identifier will be used to refer to that space.
- All variables must be declared before they can be used!
- Variable declarations have the format

1. data type Name(identifier);
2. int someNumber;
3. double radius;
4. char let;

Assignment Statement

- The assignment statement is used to store values in memory locations
- The general syntax is

  identifier (variable) = expression;
- Where expression may be simple or complex expression (equation).
- The expression evaluated first then the result is stored at the identifier.
- Later will discuss expressions in more details

Assignment statement (2)

- The assignment statement can be used to initialize variables. Examples are:

  int num1 = 15; // initialize to constant value

  int num2 = num1; // initialize to variable

  char = 'A'; // initialize to character

- Note the use of quotations with charters to differentiate it from variables

  double sum = 13.2; // initialize double values

  int x = 13.2; // store only 13

  int z = 'A'; // converts char to int stores 65 in z.

  double f = 12; // stores 12.0 in to f

Input Statement

- Allows data entered by the keyboard to be stored in variables.

<u>The general syntax :</u>

cin >> variable; // note the use of >>

- Examples are: int abc; cin >> abc;
- Can enter multiple values in one statement.

  cin>>length>>width;

- cin skips all white spaces blanks, newlines, and tabs.

  example cin >> x>>y; skips all spaces between x and y.

- cin requires the use of the pre-processor directive

  #include <iostream.h> as first line in the program file

Output Statement

- Used to display text and data to the screen.
- The general syntax is

  cout << exp; // note the use of <<

- Examples are:
- cout << 5; // display constant value
- cout << "Hello World"; // display text
- int num =5; cout << num; // display variable num
- int val; cin >> val; cout << val+2; // evaluate expression and display result on the screen.
- cout << "the area is " << 5 *2; // text + value;
- cout << "the house" << endl << "is full"; prints

  the house

  is full

- Use endl to stop printing on current line.

Expression Definition

- An expression in C/C++ is a C statement that may contain constants, variables and operators.
- An expression is two types **simple** and **complex**
- <u>Simple expression</u> is either constant value such as integer 12, double13.4 or character 'A' or a value of a variable such int x= 5; the value of x.
- <u>Complex expression</u> contains simple expressions and operator to be applied on it or them. Examples: 5+7, x*2+7/2, X*2> Y. Complex expression are build from other simple or complex expressions.

- Every expression must have a value: if the expression is constant it value is the value of the constant (12, 13.5. 'A'). If variable the value stored in the variable (int x =12, value 12). Complex evaluate expression to compute value (x + 2).

Expression Types and Values

- Expression may contain many simple, complex expressions and many operators applied on these expression that results in a single value.
- An expression can be either of two types: Arithmetic or Logical
- Arithmetic expression applies an arithmetic operator to an operands (expressions) (+,-,*,%, … more later)
- Logical expression applies logical operator to an operands (expressions) (>, < ,&& ||, more later)
- The Final value of an expression is either logical or arithmetic depending on last operator executed. If the last operator is logical the expression final value is either “true” 1 or “False” 0. If the last operator is arithmetic the expression final value is a arithmetic (integer or double)
- For any expression if the arithmetic value is zero its logically “false” otherwise it is logically “true”.

Expression Evaluation

- Operand: means the integer or floating-point constants and/or variables in the expression.
- There are two kinds of numeric values:
  - Integers (0, 12, -17, 142)
  - Floating-point numbers (3.14, -6.023e23)
- Operators: are things like addition, subtraction multiplication, greater than and less.
- The value of an expression will depend on the data types and values and on the operators used
- Additionally, the value assigned to a variable in an assignment statement will also depend on the type of the variable.

Arithmetic Operators

- Operators can be combined into complex expressions

```
result  =  total + count / max - offset;
```

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Precedence rules
  - Parenthesis are done first
  - Division, multiplication and modulus are done second
    - Left to right if same precedence (this is called associativity)
  - Addition and subtraction are done last
    - Left to right if same precedence
- Operator types:

Binary: operates on two operands : *6.5 * num*

Unary: operates on one operand: *-23.4*

Sample Expressions

- Operators on doubles:

  unary: - and binary: +, -, *, and /

  Constants of type double: 0.0, 3.14, -2.1, 5.0,

  Sample expressions:

  – 0.4 * income - children * 500
  – (A4.0 / 3.0) * 3.14 * radius * radius * radius

- Operators on integers:

  unary: - and binary: +, -, *, / and %

  Constants of type *integers*: 0, 1, -17, 42

  Sample expressions:

  – 5 + 4 * 2
  –Int x =10; x/2

int Division and Remainder

Integer operators include

integer division (/) and

integer remainder (%):

/ is integer division: <u>no</u> remainder, <u>no</u> rounding

299 / 100 → 2

6 / 4 → 1

5 / 6 → 0

% is mod or remainder:

299 % 100 → 99

6 % 4 → 2

5 % 6 → 5

A Cautionary Example

int radius;

double volume;

double pi = 3.141596;

volume = (4/3) * pi * radius *radius * radius;

Result is (1) * pi * radius *radius * radius;

result is 3.141596 * radius *radius * radius

val= (3/4)* radius

result is 0 * radius = 0.0 * radius

Order of Evaluation

Precedence determines the order of evaluation of operators.

Is a + b * a - b equal to (a + b) * (a - b) or a + (b * a) - b ??

And does it matter?

Try this:

4 + 3 * 2 - 1

(4 + 3) * (2 - 1) = 7

4 + (3 * 2) - 1 = 9

7

9

Operator Precedence Rules

Precedence rules:

–1. do ( )'s first, starting with innermost

–2. then do unary minus (negation): -

–3. then do "multiplicative" ops: *, /, %

–4. lastly do "additive" ops: binary +, -

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses "on the same level" (i.e., not nested), they are evaluated left to right.
*, /, or %	Multiplication Division Modulus	Evaluated second. If there are several, they re evaluated left to right.
+ or -	Addition Subtraction	Evaluated last. If there are several, they are evaluated left to right.

Associativity Matters

•Associativity determines the order among consecutive operators of equal precedence

•Does it matter? Try this: 15 / 4 * 2

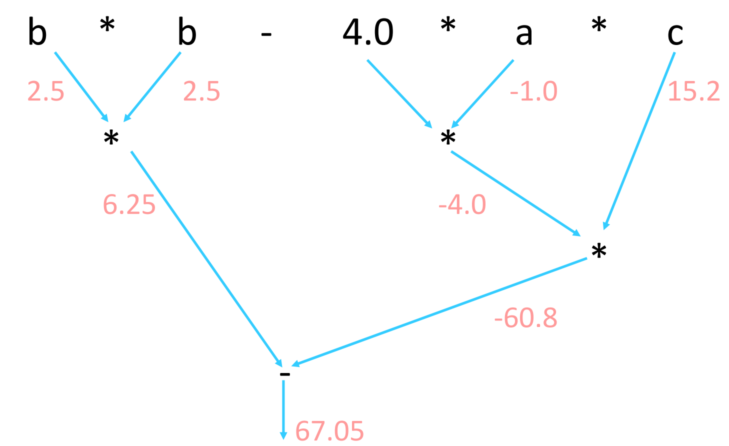
(15 / 4) * 2 = 3 * 2 = 6

15 / (4 * 2) = 15 / 8 = 1

•Most C arithmetic operators are "left associative", within the same precedence level *a / b * c* equals *(a / b) * c*

Depicting Expressions

assume a = -1.0; b = 2.5; and c = 15.2 then



Data Conversions

- Sometimes it is convenient to convert data from one type to another
  - For example, we may want to treat an integer as a floating point value during a computation
- Conversions must be handled carefully to avoid losing information
- *Two types of conversions*
  - *Widening conversions* are generally safe because they tend to go from a small data type to a larger one (such as a `short` to an `int`)
  - *Narrowing conversions* can lose information because they tend to go from a large data type to a smaller one (such as an `int` to a `short`)

Data Conversions

- In C#, data conversions can occur in three ways:
  - Assignment conversion
    - occurs automatically when a value of one type is assigned to a variable of another
    - only widening conversions can happen via assignment
    - Example: aFloatVar = anIntVar
  - Arithmetic promotion
    - happens automatically when operators in expressions convert their operands
    - Example: aFloatVar / anIntVar
  - Casting

Data Conversions: Casting

- *Casting* is the most powerful, and dangerous, technique for conversion
- Both widening and narrowing conversions can be accomplished by explicitly casting a value
- To cast, the type is put in parentheses in front of the value being converted
- For example, if `total` and `count` are integers, but we want a floating point result when dividing them, we can cast `total`:

```
result = (float) total / count;
```


Conversions in Assignments

```
int total, count, value;
double avg;
total = 97 ;    count = 10;
avg = total / count ;   /*avg is 9.0!*/
value = total*2.2;      /*Wrong Result*/
```

implicit conversion to double

implicit conversion to int – drops fraction with no warning

Explicit Conversions

Use a cast to explicitly convert the result of an expression to a different type

Format: **(type) expression**

Examples ***(double) myage***

(int) (balance + deposit)

This does not change the rules for evaluating the expression itself (types, etc.)

Good style, because it shows the reader that the conversion was intentional, not an accident

Using Casts

```
int total, count ;
double avg;
total = 97 ;    count = 10 ;
    /* explicit conversion to double (right way)*/
 avg = (double) total / (double) count;     /*avg is 9.7 */
 avg = (double) total /count;
 avg = total/ (double) count;
/* explicit conversion to double (wrong way)*/
 avg = (double) (total / count) ;           /*avg is 9.0*/
```

Advice on Writing Expressions

- Write in the clearest way possible to help the reader
- Keep it simple; break very complex expressions into multiple assignment statements
- Use parentheses to indicate your desired precedence for operators when it is not clear
- Use explicit casts to avoid (hidden) implicit conversions in mixed mode expressions and assignments
- Be aware of types: *Every* variable, value, and expression in C has a type (int, double or char)

Relational Operators

Logical expressions are C statements that when evaluated result in *true* or *false* values. In C *true* is represented by any numeric value not equal to 0 and *false* is represented by 0

<u>Relational Operators</u>

Relation operators allow us to compare two expressions or variables. Below is a list of these relational operators in order of precedence.

> Is greater than

< Is less than

> = Is greater than or equal to

< = Is less than or equal to

= = Is equal to // This is a mathematical equals

! =Is not equal to // An exclamation point means not in C++

Logical Operators

There are three types of logical operators which can be used to combine Boolean expressions into compound Boolean expressions.

The operators are: !(not) , && (and) , || (or)

The following table summarizes these operators

x	y	x && y	x || y	!x
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Short Circuits &&

- Short circuit evaluation looks at a compound expression and evaluates it until it reaches a conflict a final result of the expression
- A n expression filled with ANDs

  p && q // where p, q are boolean expressions.
- if p is false, then the expression is false and therefore, the evaluation will stop, i.e. p is not evaluated.
- If p is true, then evaluate q and the result of the overall expression depends on the evaluation of expression r.

int x =4; (x==4) && (x = 3); cout << x; what is printed?

int x = 4; (x > 4) && (x = 3); cout << x; what is printed?

Short Circuit ||

If we have A long expression filled with Ors like

p || q // where p, q are Boolean expressions

- if p is true, then the expression is true and therefore, the evaluation stop there.
- if p is false q is evaluated and the value of the overall expression depends on expression r being true or false.
- Example:

int x = 4; (x ==4) || (x =3); cout << x; what is printed?

Int x =4;   (x > 4) || (x = 3); cout << x; what is printed?

Summary and examples

Conditional AND (&&) and Conditional OR (||)Would not evaluate the second condition if the result of the first condition would already decide the final outcome.

This argument extends for expression of the from

P && q && r &&…….

or

P || q || r || …..

Logical and Arithmetic Operator Precedence

1. **Parenthesis ()** **Highest precedence**
2. **Unary ! not and – (negative) (cast)**
3. ***, /, % multiply, divide remainder**
4. **+, - plus and minus**
5. **>, <, >=, <= less, greater, less than, greater than**
6. **==, != equal and not equal**
7. **&& (AND)**
8. **|| (OR)**
9. **= (assignment)** **Lowest precedence**

Mixing Expressions

- It is possible to include logical and arithmetic operators in the same expression.
- The result of evaluating such expression is logical or arithmetic depending on the final operator being performed.
- If the last operator is logical then the result is either true or false.
- If the last operator is arithmetic the final value is arithmetic.

Mixed Expression

- Special care must be taken when evaluating such expressions with the order of precedence. Example int x =5; y =7, int z;

  z = x+3 > y. In the precedence rules above the > operator is evaluated last hence result of expression is logical and value stored in z is 1.

  z = x +(3 > y). The last operator evaluated is +

  Since 3 < y is zero. Then z = x+0. This is an arithmetic expression and in 5 is stored in z.

Assignment Revisited

- You can consider assignment as another operator, with a lower precedence than the arithmetic operators

First the expression on the right hand side of the = operator is evaluated

```
answer  =  sum / 4 + MAX * lowest;
           4       1   3     2
```

Then the result is stored in the variable on the left hand side

Examples

Given int i= 1, j =3, k =4; Evaluate the following expressions. Assume each expression is independent of the others

- 2*i || i<j
- k = i+2 == 3
- 3 && k==4
- i = 3 && i < 3
- (k =2) && k == 2
- !k || k > 0
- K < 10 + 2 *(k =5)
- k/3+5.0/3+k==4

Examples

- Given int x=1, y =2, z=14;

  Then the value of the expression

  (x >= 1 && y==3 || z < 12) is 0 “false”

- Given int x,y=12; x = (y==12) + 25%4;

  What is the value of x. 2

- double w = (int) 13.5 + (int) (21/4 +3.5);

  What is the value of w 21.5

Examples Continue

- To test if X is outside the range 5…20 which of the following is correct

  a. (5 > X >20) b. (X<5 || X >20)

  c. (X>=5 && X<=20) d. (X<5 && X >20)

- Given int x=4; int y=3; x = x/y;

  what is the value of x

  a. 0 b. 0.75 c. 1.25 d. 1

Short Hand Operators

Syntax:

Variable Op.= Expression;

Evaluated as

Variable = Variable Op. (Expression);

Assignment operator	Sample expression	Explanation
+=	c += 7	c = c + 7
-=	d -= 4	d = d - 4
*=	e *= 5	e = e * 5
/=	f /= 3	f = f / 3
%=	g %= 2	g = g % 2

Examples

Int x =10; int y =20; int z = 30;

- x+= 5; equivalent to x = x + 5;
- y *= z-12; equivalent to y = y *(z – 12);

  important: not y = y * z -12;
- z /= x+y/4; equivalent to z = z /(x+y/4);
- x % = y; equivalent to x = x%y;
- x+1 += 12-z+4; // illegal operation

Increment and Decrement Operators

Operator	Called	Sample expression	Explanation
++	preincrement	++a	Increment **a** by 1, then use the new value of **a** in the expression in which **a** resides.
++	postincrement	a++	Use the current value of **a** in the expression in which **a** resides, then increment **a** by 1.
--	predecrement	--b	Decrement **b** by 1, then use the new value of **b** in the expression in which **b** resides.
--	postdecrement	b--	Use the current value of **b** in the expression in which **b** resides, then decrement **b** by 1.

Fig. 4.13 The increment and decrement operators.

There is no difference between post and pre increment on the variable itself. However, the difference in the final value of expression in which pre and post increment are found.
Post increments the variable *after* it is used in evaluating the expression and Pre increments the variable *before* it is used in evaluating the expression

Difference between Pre and Post

Pre and Post increments/decrements with respect to the value of the variable.

Pre

int number = 5; // declares number to be 5

++number; // increments number to 6.

Post

int number = 5; //declares number to be 5

number++; // increments number to 6

Pre and post increment/decremented with respect the value of the expression.

Pre

int number = 5, b;

b = number++;   b = 5, number = 6

Post

int number = 5, b;

b = ++number;   b = 6, number = 6

Examples

- Given int x =5; int y =10; then what is the output assume each group of statements are independent
- x++;   cout << x;
- cout << ++x;
- cout << x++;
- --x;   y++ ;   cout << x+y;
- cout << --y – x--;   cout << x << y;
- cout << y++ + x++; cout << x;   cout << y;

Precedence and Associativity

high

low

Operators	Associativity	Type
( ) ++ --	left to right right to left	parentheses unary postfix
++ -- + - (*type*)	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
= += -= *= /= %=	right to left	assignment

Pre and Post Operators Summary

- Syntax : (Variables)++;   (Variables)--;
  ++(Variables);   --(Variables);
- Can not do: (expression)++; (X + 1) ++; (5)++;
- Which value of the variable used in evaluating the expression? Depends on pre or post
- Pre: update variable first then use new value in evaluating the expression.
- Post: Evaluate expression first using old value then update variable.
- Exmaple:

int x =5, y =6; int z = x++ ---y;   z =10, x =6, y =5;

int z = 13; cout << --z; cout << 5+z++; cout<< 5+++z;

12   18   19

Math Functions

- An expression in C/C++ may need to perform a mathematical functions like compute the square root of a value or the sin of a degree
- C provides build in functions to perform common operations.
- To use these functions we must insert

  #include <math.h > at the of program file.
- These functions are called by writing

  functionName (argument); or

  functionName(argument1, argument2, …);

Math Functions Examples

- Examples:

  cout << sqrt(900.0); **//** would print 30.

  int x =5; int y = 2; int z;

  z = x + exp(2.0) + sqrt(900)/2; 5 + 7 + 15 (27);
- All functions in math library return a **double.**

  Function arguments can be
  - Constants: sqrt(4);
  - Variables: sqrt(x);
  - Expressions: sqrt(sqrt(x)) ; or sqrt(3 - 6x);

Method	Description	Example
`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0` `ceil( -9.8 )` is `-9.0`
`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`
`exp( x )`	exponential function *ex*	`exp( 1.0 )` is `2.71828` `exp( 2.0 )` is `7.38906`

`fabs( x )`	absolute value of *x*	`fabs( 5.1 )` is `5.1`
		`fabs( 0.0 )` is `0.0`
		`fabs( -8.76 )` is `8.76`

`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0` `floor( -9.8 )` is `-10.0`
`fmod( x, y )`	remainder of *x/y* as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`

`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0` `log( 7.389056 )` is `2.0`
`log10( x )`	logarithm of *x* (base 10)	`log10( 10.0 )` is `1.0` `log10( 100.0 )` is `2.0`
`pow( x, y )`	*x* raised to power *y* (*xy*)	`pow( 2, 7 )` is `128` `pow( 9, .5 )` is `3`
`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0`
`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0` `sqrt( 9.0 )` is `3.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`

Computer Programming

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

The Selection and Iteration Statements

Selection statement (if Statement)

- Gives the ability to choose which set of instructions are executed according to a condition.
- Choose among alternative courses of action

  Syntax :**if (*condition*)**

  ***statement*;**

  The if statement allows you to evaluate a *condition* and only carry out the statement if the *condition* is true (not zero).

  – Example:

  Read student's grade

  If student's grade is greater than or equal to 60
  Print "Passed"

```
int grade; cin >> grade;
if ( grade >= 60 )
   cout << "Passed";
```


`if/else` Selection

Different action is taken depends on conditions being true or false

Example:

Read studen's garde

if student's grade is greater than or equal to 60
print "Passed"
else print "Failed"

```
if ( grade >= 60 )
   cout << "Passed";
else  cout << "Failed";
```

Another example:

if (hours <= 40.0) pay = rate * hours;
else pay = rate * (40.0 + (hours - 40.0) * 1.5);

Nested If Statements

There are no restrictions on what the statements in an if statement can be. For example an if statement can contain another if statement.

```
if (x < 0)
   if (y != 4)
       z = y * x;
   else
       z = y / x;
else
   if (y > 4)
       z = y + x;
   else
       z = y - x;
```

In the code above first if statement contains another if else construct and the else statement contains another if else construct. Please note if no braces are used always the else statement matches the closest if.

More Examples

If Statements are Independent of each other

int day;   cin >> day;
if (day == 1)   cout << "Sunday";
if (day == 2)   cout << "Monday";
…………….
if (day == 7) cout << "Saturday";
if (day < 1 || day > 7) cout << “Unknown Day";

In the code above all the if statement must be evaluated. However, the cout statement is executed for only one of them depending on the value of day entered.
This wastes computation time.

if … else if …. else construct

int day;   cin >> day
if (day == 1) cout << "Sunday";
else if (day == 2) cout << "Monday ”;
……………………..
else if (day == 7) cout << "Saturday";
else cout << “Unknown day”;
statement

The code above is more efficient because when one statement evaluates to true the rest of statements are skipped to the statement after last else. Evaluation is done in the order of the statements.

Another Example

- Compute tax based on income

Income	% Tax
< 20,000	No tax
>= 20,000 and < 35,000	20%
>= 35,000 and < 50,000	25%
<= 50,000 and < 100,000	30%
>= 100,000	35%

int income;
cin >> income;
if (income < 20000)   printf(“No tax.”);
else if (income < 35000)   cout << “tax = “ << 0.20 * income;
else if (income < 50000)   cout << “tax = “<< 0.25 * income;
else if (income < 100000) cout << “tax =” << 0.30 * income;
else   cout << “tax =” << 0.35 * income;

Compound Statements

– Set of statements within a pair of braces

```
if ( grade >= 60 )
   cout << "Passed.\n";
else {
   cout << "Failed.\n";
   cout << "You must take this course again.\n";
}
```

– Without braces,

```
cout << "You must take this course again.\n";
```

always executed

Block

– Set of statements within braces {   statements   }

Common Mistakes

- One common mistake can occur when the == (equality) operator is confused with the = (assignment) operator.

int n; cin >> n;
if (n = 3) cout << "n equals 3"; // statement is always true;

- independent if statement: more than one statement may be executed? int n; cin >> n;

```
if (n > 0)  cout <<  "positive";
if (n%2) cout <<"odd";
if (n < -10) cout << "not zero";
else cout << "zero";
```

what is the output

if n = 0, 10, 17, -7 -50;

Common Mistakes (2)

The null statement: if (condition);

```
int num;
cin >> num;
if (num >0);
  cout << num;
```

In this case the value of num will always be printed on the screen regardless of its value. The reason is that the "cout << num;" statements is outside the if selection.

Code Examples

- if only // read double prints negative if value is less than 0

```
double db;
cin >> db;
if (db < 0)
   cout << negative;
```

- If … else   // read two numbers and print the smallest

```
int x, y;
cin >> x >> y;
if ( x < y)
     cout << x;
else
     cout << y;
```


Code Examples (2)

if …   else if …   else construct

```
// Code to print case of letter:
char  ch;
cin >> ch;
 if ( (ch >= 'a') && (ch <= 'z'))
      cout << "Small Letter";
 else if ((ch >='A') && (ch <= 'Z'))
      cout << "Capital Letter";
else   cout << "None Letter";
```

The Switch Statement

– Test expression for multiple values
– Series of **`case`** labels and optional **`default`** case

```
switch ( expression ) {
  // only works if expression evaluates to integer value
   case value1:     // taken if expression value is value1
        statements
         break;       // necessary to exit switch

   case value2:
   case value3:      // taken if expression value is value2 or value3
        statements
        break;

  default:        // taken if expression value matches no other cases
     statements
     break;     // break here not necessary with last option
}
```


Examples

Read in day as a number 1,2,3,….,7 and
print it as text Sunday, Monday, Tuesday, ……., Saturday.

```
int day;
cin >> day;
switch (day) {
   case 1 : cout << " Sunday"; break;
   case 2 : cout << " monday"; break;
    case 3: cout << "Tuesday"; break;
    …………….
   case 7 : cout << " Saturday"; break;
   default : cout << "unknown day";
}
```


Examples (2)

You can have multiple statements per case

```
Int x, y;
cin >> x >> y;
   Switch (x<y){
   case 1:   cout << "x is smaller than y";
              cout << x;
             cout << y;
             cout << " The sum of x and y is " << x+y;
             break;

   default :       // case 0   makes no difference
             cout << "x is greater than or equal to  y";
             cout << x + y;
}
```


Missing Breaks

```
int n;
cin >> n;
switch (n) {
   case 1: cout << "one ";
   case 2: cout << "two ";
   case 3: cout << "three "; break;
   case 4: cout << "four ";
   default: cout << "good bye";
}
```

// In the code above if n == 1 then one two three are printed.
If n == 2 two three are printed.
If n ==3 three is printed.
If n ==4 four good bye are printed.

Default location

char gender;

cin >> gender;

switch (gender) {

default: cout << “Uknown gender”; break;

case ‘M’:

case ‘m’: cout << “Male”; break;

case ‘F’:

case ‘f’: cout << “Female”; break;

}

The example above illustrates that

The default statement does not have to be the last statement in the cases block. It could be placed anywhere first, last or in between.

the last case may or may not have break;

Multiple case values

char gender;

cin >> gender;

switch (gender) {

default: cout << “Uknown gender”; break;

case ‘M’:

case‘m’: cout << “Male”; break;

case ‘F’:

case ‘f’: cout << “Female”; break;

}

The example above illustrates that if the same set of statement to be executed for more than a single case value, the case values are written following each other. Then the statement to be executed follow that last case. In the example above we print Male if the input is ether ‘M’ or ‘m’. Print Female if the input is either ‘f’ or ‘F’

Iterations

Definition: Iteration is a repetition structure in which a set of statements are repeated while some condition remains true

- Example

  while there are more numbers to read

  Read number and perform processing

- **while** loop repeated until condition becomes false

- Example

```
int product = 2;
while ( product <= 1000 )
   product = 2 * product;
```


Loop Definition

- Loops allow a group of statements to be executed over and over again.
- All loops must have:
  - loop-control variable(s)
  - body - block of statements to be executed repeatedly
  - a way for the loop to be terminated.
- Three basic loop mechanisms: while, for, and do-while.

While Loop

- Has the Syntax;

```
initial condition;
 while (conditional expression)
 {       statement(s)     // body of loop
 }
```

- The statements comprising the body of the loop will be executed until the conditional expression evaluates to false.
- Therefore one of the statements in the body should modify the loop-control variable(s) so the loop terminates.
- While loops are pre-test loops, the condition for repetition is tested before the loop is executed.

While Loop (cont.)

- The while loop may not be executed if initial loop condition is false.
- The initial condition is optional it sets up the condition based on which the loop may or may not executed.

Loop types:

(1) Count-controlled repetition

Loop repeated until counter reaches certain value

Number of repetitions known

Example:

```
int n = 0;
while (n < 10) {
        cout << "hello";
        n++
}
```


While loop continue

(2) Sentinel value:

Loop ends when certain value reached.

Example

```
int radius;  cin >> radius; // initial condition
 while ( radius <= 0 ) {
      cout<<" Zero is not a valid radius! \n"
           <<"Please re-enter the radius.\n";
      cin>>radius;
  }
```

The loop is only executed if an invalid value is entered. An invalid value is radius <=0. Loop is ended when user enters valid value for radius.

Examples

- Read 10 int number and compute their average:

```
int x;
 int count =0;
 int  sum =0;
while ( count < 10)  {
     cin >> x;
    sum += x;
    count++;
}
  cout << "The average is " << (double) sum/ count;
```

Examples (2)

Keep reading int values until their average exceeds 1000

```
 int x;                    int count =0;
double  sum =0;       double average = 0;
while (average <= 1000) {
   cin >> x;
    sum += x;
   count++;
   average = sum/ count;
}
  cout << "The average is " << average;
```


Examples (3)

Read characters until '#' is entered.
Print the count of small and capital letters entered.

```
int  countsmall =0;     int countcapital =0;
char c;     cin >> c;
while (c != '#') {
    If( c >='a' && c <= 'z')   countsmall++;
    If( c >='A' && c <= 'Z')  countcapital++;
   cin >> c;
   }
cout << "the count of small is " << countsmall << endl;
cout << "the count of capital is " << countcapital;
```


Common Mistakes

- Not reaching the termination condition - loop never ends. It is an infinite loop.
  `int x = 1; while ( x > 0) cout << "hello";`
- Missing braces - only first statement is executed as body of the loop.
  `int x = 0; while ( x < 10) cout << "hello"; x++;`
- A semicolon at the end of while line
  `while (condition);`
  No statement is executed - it is an empty loop and an infinite loop once it starts. Like the null statement in the If structure.
  `int x =0; while ( x  >0); cout << "hello"; x++;`

Do-While Loops

- The do-while loop is a post-test loop.
- The condition is tested after the loop is executed.
- The loop is always executed at least once.

```
do {
        statement(s);    // body of loop
} while (condition);
```

Example:

```
int x =0;
do {
        cout << x;
    } while (x !=0);
```

This loop prints 0 before it stops.

Common Use

A common use is printing menu continuously on screen:

```
 int n1, n2, result;
Do {
        cout<<"Please enter S to subtract two values“ << endl
                << “enter A  to add two values” << endl
                << “or enter  Q to quit”;
        cin> > operation;
        switch (operation) {
                       case ‘A’ : cin >> n1 >>  n2;
                                    result = n1 +n2;
                                   cout << result;
                           break;
                       case ‘S’: …..
            }
   }     while (operation != ‘Q’);
```


Another Example

Keep doing area calculations for rectangles while user wishes

```
void main ( ) {
     int length, width; char answer;
     do {
             cout<< “please enter length: ”;
              cin >> length;
              cout << “please enter width:”;
              cin >> width;
              cout << “the area is “ << length * width << endl;
             cout<<“Would you like another calculation enter”
                   <<  y or Y  any other character to quit”;
             cin>> answer;
             } while (answer == ‘y’ || answer == ‘Y’);
     }
```


For-loop

- Pretest loops
- Mostly count-controlled
- Have the format

```
     for(initialization; test; update)
        {      statement(s)     }
```

```
//  suppose you want to read 5 numbers and print their sum
     int j, num,  sum = 0;
     for( j = 0; j < 5; j++) {
      cin >> num;
      sum += num;
    }
   cout<<sum;
```

- Note that j is only used to control the number of imes this loop executes,   you could write for (j = 1; j <=5; j++) … or for (j = 5; j > 0; j--) …..   would give the same results.

Common Mistakes with for loop

- Missing braces – only the first statement is repeated.

```
  int sum =0;
  for (int I =1; I <=5; i++)     // note that 1+2+3+4+5 is 15
        cout << I;
       sum +=I;
 cout << “           the sum is “ << sum;
prints: 12345       the sum is  5;     why?
```

- Updating the loop control variable in the body of the loop – the variable is updated twice.

```
 for ( int I =0; I < 10; i++)  {
      cout < i << endl;
      i++;
  }
```

Prints: 0 2 4 6 8   why?

Omitting some of the loop parts

One, two, or all of the expressions may be omitted from the for loop.

Sample 1: for(; x < 10 ; x = x+2)

the initial value of x is taken from earlier part of code.

Sample 2: for(; x < 10 ;)

the loop control variable must be updated in the body of the loop to avoid an infinite loop.

Sample 3: for(; ;) **// an infinite loop like while(1)**

Loop updates and control must be done inside loop body
we will handle such loops later using break statement

What Loop should you use

- Any while loop can be converted into: do while loop or for loop and vice versa.
- The loop to use depends on the problem?
- Should the loop always execute at least once?
  - Yes ➔ do-while No ➔ while or for
- Should the loop be count controlled or sentinel controlled.
  - Count ➔ for is the most common
  - Sentinel➔ while or do-while is most common
  - For loops fit more naturally with count controlled
  - While loops more naturally with sentinel controlled
  - Do while are rarely used.

Examples

keep reading and printing chars until '#' is read.

```
// while loop
   int c ; cin >> c;
   while (c != '#') { cout << c; cin >> c;}
// for loop
   char c;   cin >> c;
   for (;c !='#';) { cout << c; cin >> c;}
// do while
  char c;  do {
                cin >> c;  cout << c;
               } while (c!='#');
```


Example using for

Problem: Read 10 integers and print the smallest

Solution: (algorithm)

read first number consider it as smallest
then read next number and compare it with smallest
change smallest if the next number is smaller than smallest
Repeat the process until 10 numbers are read.

```
Code :  int num, smallest;
        cin >> num; smallest = num;
        for (int i =1; I <=10;  i++) {
                 cin >> num;
                 if (num < smallest)
                 smallest = num;
}
cout << " the smallest number is << smallest;
```

Example using while

Write code that keeps reading int numbers until 0 is entered find the sum of all odd numbers and the product of all even numbers.

Algorithm:…………

```
Code: int num;  int sum = 0; int  product =1;   // initial values
   cin >> num;
   while (num !=0) {
        if ( num%2)                 // if (num%2 ==1)
                sum + = num;
        else                        // if (num% 2 ==0)
                product *= num;
        cin >> num; }
cout << "the sum of all positive numbers is  "<< sum << endl;
cout << "the product of all negative numbers is "<< product << endl;
```

Nested Loops

Some problems require the use of a loop inside another loop.

Example keep reading integers until 0 is entered and for each read integer n compute the sum of values from 1 to n inclusive

- Design and test outer loop
- Design and test inner loop

Code: for inside while

```
int num, sum;       cin >> num;
while (num > 0) {         // outer loop
   sum =0;
   for (int i =1; i < = num; i++)     // inner loop
        sum += num;
   cout << sum << endl;
   cin >> num;
}
```


More Nested Loops

For Loops can be "nested" inside another for loop

```
      for (int j = 1; j < 10; j++) // outer loop
            for (int k = 1; k<= 10; k++) // inner
                cout << j << 'x' << k << "=" << j*k; << endl;
```

For each iteration of the outer loop the entire inner loop is executed

```
Another example:  int i; j, sum;
  for ( i =1; i <=3; i++)
  {   cout << i << endl;
     cin >> j;    sum =0;
      for (j = 1; j <= n; j++)
         sum +=j;
 } loops read j and prints sum form 1 to j repeated 3 times
```


Inter loop dependence

inside loop is control dependent on outside loop

```
   int outer, inner;
  for (outer=1; outer<=5; outer++) {
       int sum =0;
      for (inner=1; inner <= outer; inner++)
           sum += inner;
       cout << sum <<  "     ";
     }
```

The output: 1   3   6   10   15

Notice not using braces because the entire inner loop is considered as one statement with respect to the outer loop.

Using "Break" in Loops

- It is a way to stop loop execution.
- It should be used very cautiously as it makes the code more difficult to understand.
- It is usually part of an if structure inside the loop

```
if (cond)  break;
```

Example: Keeps reading and printing characters until small letter is entered

```
char c;
 while (1) {
    cin >> c;
     if ( c >= 'a' && c <='z'))   break;
     cout << c; }
```


Common Use

- When the programmer writes a for loop or a while loop, but wants to stop the loop when some value is reached.

```
char c; while (1) { cin >> c; if ( c =='#') break; …….}
int x; for (;;)     { cin >> x; if ( x%2) cout << x; else  break;}
```

- Example 1:

```
for (int i =0;   I < 100 ; i++) {
        cout << i;
        if (I == 9) break; }      // be carful  if (I = 9) break;}
```

Prints: 0123456789 // Prints 0

- Example 2:

```
int x =10, sum =0;
   while (--x) { sum += x; if (x ==5) break;}  // if (x =5)
   cout << sum;
```

Prints: 35 // 9

Break in Nested Loops

- When break is used in an inner loop, it only interrupts that loop, the iterations of outer loops would continue.
- When break is used in an outer loop, the inner loop is also ended.

Example:

```
int j =0;
    while (j < 3){
        for (int I = j; I < 10; i++) {
       cout << I;
       if (I == 5) break; }
       cout  << endl;  j++;  }
```

Prints: 012345
12345
2345

Using Continue in Loops

- It is usually used with an if structure inside the loop
- Tells the loop to skip over the remaining statements in the body and go execute the update statement
- Example 1:

```
for (int I = 0; I < 10; I++) {
        if (!(I %2))          // if (I%2)
              continue;
      cout << I;
 }
```

Prints: 13579 // 02468

Break and Continue Examples

```
int i =0;
while (i++) {
  if (i == 2)
     continue;
  if( i == 8)
      break;
  if( i < 5)
      cout << i << “,”;
}
cout << “I = “ << i;
```

Prints: I = 1

```
int i =0;
while(++i) {
    if (i == 2)
       continue;
    if (i == 8)
       break;
   if ( i < 5)
       cout << i << “,”;
}
cout << “I =“ << i;
```

Prints: 1,3,4, I=8

Break and Continue Examples 2

What is printed?

```
for ( int i =0; i < 20; i++) {
   if  (i ==5||i == 11)
          continue;
   if (I == 13 || I == 16 )
          break;
   if (I >= 4 && i < 8)
          i=9;
   cout << I << “,”;
}
```

Prints 0,1,2,3,9,10,12,

Be carful when using continue in a while loop?

```
int I = 0;
while (I < 10) {
    if (I %2 )
       continue;
    cout << I;
       I++;
}
```

Prints 0 then goes to infinite loop

Prime or not Prime Example

Write code to keep reading one integer numbers and quits if number greater than 100 is entered.

For each entered number print prime or not prime:

```
int  num ;
while (1) {
   cin >> num;
   If(num >= 100)  break;
   for (int i = 2;  i < num; i++)
        if (num%i ==0) break;
   if( i == num) cout << num << “ is prime” << endl;
   else cout << num << “ is not prime” << endl;
}
```


Summation Example

Write the code to read x and y then compute the following equation :

$$\sum_{I=1}^{I=X} 2*(Y+I^2), \quad \text{where x and y > 0.}$$

Solution:

```
int x,y;
cin >> x>>y;
for (int I =1, int sum =0;  I <= x; I++)
      sum += 2 * ( Y + I * I);
cout << “The sum is “ << sum;
```

Write Code Examples 1

Keep reading int numbers until 0 is entered, then
for each entered number n
if n > 0 compute the sum of values between 1 and n
if n < 0 compute the product of values between -1 and n
Solution:

```
int n;    int product =1;    int sum = 0;
 cin >> n;
 while (n) {
   If (n > 0)    for (int i = 0;  i < = n; i++) sum += i;
   else          for (int i = -1;  i >=n; i--) product *= i;
   cin >> n; }
```


Write Code Examples 2

keep reading numbers stops when negative number is read. find the sum of all positive odd numbers.

```
int num , sum =0;
 for (;;) {
cin >> num;
If (num < 0) brerak;
else If (num %2 && num > 0)
   sum+= num;
else
   break; }
```

Read characters until non letter ins entered convert small to capital

```
char c;
while(1) {
cin >> c;
If (c > ='A' && c <='z')
   continue;
else If(c>='a' && c <='z')
  cout <<  (char) c-32;
else break; }
```


Write Code Examples 3

- Print all small letters.

```
 for ( char c = 'a'; c <= 'z'; c++)
      cout << z << endl;
```

- Read int number n and print its factorial

```
int n, product =1;
for (int I =n; I >=1; I--)
    product * = I;
```

- Print all numbers 0 to 1000 that are even and divisible by 3

```
for (int I = 0; I <= 1000;I++)
     if(I%2 == 0 && I%3 ==0)
           cout << I << endl;
```


Summary

- If statement can be used independent of else
- Else statement must be associated with an if
- In if … else if … else construct: when one is true no more checking is done
- Any loop can be converted to other type
- For loops are count controlled
- While loops are sentinel
- Be carful with the null statement in if and loops
  if (x > 0); cout << x;   while (x > 0); x--;

Computer Programming

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

Review Questions and Problems

(1) double x = (int)(4/5 +(float)(5/2) + (3.0*3)/2);
what is the value of x.

a. 7.8 b. 7.0 c. 6.0 d. 6.5

(2) Given *int x= 4, y= 3; double a = x/y; cout << a*;
what is the output

a. 0.0 b. 1.33 c. 1 d. 0.75

(3) Given: cout << 5.5 + 4/7 – (double) 4/ (5%3);
what is printed

a. 3.5 b.1.0 c. 4.5 d.-1.0000

(4) float x = 3.0 + (3/6) + (3.0/2) + (float)(4/8) ;
what is stored in x;

a.5.0 c. 4 d. 5 e.4.5

(5) Given int x=5, y =10;
cout << (x==y || (x < y) && (x !=y)); what is printed

a. 0 b. 1 d. syntax error

(6) Given double z =2.3; int x =13;
int w = (int) (x + 6.7) + z + z >= 2.0; *valueof **w is**?*

a. 22 b. 23 c. 22.3 d. 1

(7) Given int a =4, int b=5; what is printed?
cout << (double) (0.5 + 2 < b + (5/2 > a/2));

1.5 b. 1 c. 0 d. NOA

(8) Given int x =5; int y =10; int z;

```
if ( x -= 5) z = x;
else if (x++) z = x;
else z = 2*x;
cout << z;
```

what is printed?

a. 0 b.1 c.2 d.5 e.10

(9) Given: int x =6; int y=5; int z=2;

```
if ((x-1) == y++)    z =y++ * --x;
else   z = x++ * ++y;
```

what is the value of z

a. 25 b. 20 c. 30 d. 42

(10) Given int a = 4; int b = 5; int c = 0;

```
if (a=5) c =a * b;
else c = b;
if (a < b) c += 10;
else        c += b ;
```

the value of c is :

a. 35 b. 30 c. 10 d. 15

(11) int x =2;

```
switch (x) {
        case 1:  cout << 1<<" ";   break;
        case 2:
        case 4: cout << 2 <<" ";
        default: cout << 3 <<" ";   break;
}
```

What is printed?

a. 2 3 b. 2 c. 3 d.123

e. nothing

(12) Given for (int x =0; x!=10;x++) {

```
    cin >> x;
    cout  << x << endl;
}
```

The last printed value is?

a. 9 b. 10 c. 0

d. unknown value e. infinite loop

(13) Given int sum =0, j = 3;

```
for (int i = j <7; j-i; i++) {sum += i+j; }  cout << sum;
```

What is printed after executing the loop:

a. 0 b.5 c.9 d. infinite loop

(14) Given: int x =0, int y = 0;

```
while (x < 10) {y = y + x++;  x+=2;}  cout << y;
```

What is printed?

a. 22 b. 18 c. 55 d.10

(15) Given int x =0;

```
while (x <5)
    x+=2;
cout << "hello" << endl;
```

How many times the word hello is printed?

a. 0 b. 1 c. 2 d. 5

(16) Given int x = 10;

```
for ( ; x > 0; x--) {
     cout << "hello" << endl;
     x = x -2;
}
```

How many times hello is printed

a. 0   b. 4   c.10   d. infinite

(17)

```
int sum =0;
for(int i=1;i<=10;i=i*2)      sum=sum+i;
cout<<sum;
```

What is printed ?

a. 0   b.4   c.8   d. 15

(18) Given int a=3, b=5;

```
while(++a < b++)
   if(++a == --b)
      if (b-- == --a)
         a=a+10;
```

what is the value of a?

a. 4   **b.** 14   **c.** 5   **d.** 3

(19) Given :

```
int x =0 ;
do{ x +=2 ;    cout<<x;
    if (x++ == 3) break;
}while(x);
```

What is the number of iterations in the last loop?

a. 0   b. 1   c. 2   d. infinit loop

(20) Given int t1=0, t2 =0;

```
for (int j =0; j < 4;j++)
     for (int k = 3; k  > 0; --k)
     {
        ++t1;
     }
          t2++;
     cout <<t1+t2++;
```

a. 13   b. 14   c. 16   d. 17

e. none of the above

- Write c code to print all values between -10 and 10 inclusive that are even and divisible by 3

```
for (int i = -10; i <=10; i++)
     if(i% 2 == 0 && i%3 == 0)  cout << I << endl;
```

- Read 2 integer numbers X and Y to print ($XY + X^{-Y}$) without using any math function in "math.h"

```
int x,y; cin >> x >> y;          int p =1, value ;
for (int i =0; I < y; i++) p *= x;
value = X * Y + 1/p;
cout << value;
```

Write a program that keeps reading integers until -1 is read your program must then print
(1) the count of even numbers
(2) the count of 3 digit numbers
(3) the average of all read numbers

Solution:

```
// initialize variables
  int  num, counte =0, count3=0, count =0;
  double sum =0.0;
   // next slide is the  loop
```



```
while(1) {
   cin >> num;
    if (num == -1) break;
    count++;
    sum += num;
    if(!(num%2)) counte++;
    if(num>99 && num<1000) count3++;
}
cout << sum/count << endl;
cout << counte << endl;
cout << count3 << endl;
```


- Keep reading integer numbers until one positive number > 0 is entered. Then print the divisors of the positive number.
- Ex: if **-10 -1 12** is entered then prints 2 3 4 6

```
int num;
  do { cin >> num;
       } while (num <=0);
  for (int i = 1; i <= num/2;i++)
       if ( num%i ==) cout << i << endl;
```


- Keep reading integer numbers and calculating their sum. Your program should stop when a ***3 digit number*** is entered .

  You must print the average of all entered numbers. Note *3 digit number* is a number between 100 and 999 inclusive.

Solution #1: Last number is included in average

```
int num, count =0;  double sum =0;
do{  cin >> num; sum+= num; cou++;
    } while (num < 100 || num > 999);
    cout << "average is " << sum/count;
```

Solution# 2 // last number is not included in average

```
int num, count =0;  double sum =0;
cin >> num;
while (num < 100  ||  num > 1000)
{ sum+= num; count++;  cin >> num;}
cout << “average is “ << sum/count;
```

Solution#3 // can or not include last number

```
int num, count =0;  double sum =0;
while(1) { cin >> num;   sum+ = num; count++;
           if (num > 99 && mum < 1000) break;}
cout << “average is “ << sum/count;
```


Write a complete program that keeps reading characters until the character '!' is read. For each character read your program needs to do the following.

if the read character is capital letter or small letter it Prints "L".

if the character is a digit (0,1,----9) it Prints "D".

for any other character it prints "*".

Then your program also prints "more capital" if the number of capital letters read is larger than the number of small letters read. Otherwise it prints "more small".

Example if input is: A c d 9 4 1 D > K ^ d 5 s!

Output is : L L L D D D L * L * D L

more small


```
char c;  int counts =0, countl =0;
for(;;){
   cin >> c;
   If (c >=‘a’ && c <= ‘z’) {cout << ‘L’ ; counts++;}
  else If (c >=‘A’ && c <= ‘Z’) {cout << ‘L’ ; countl++;}
  else if (c >=‘0’ && c <=‘9’) cout << ‘D’;
   else   cout << ‘*’;
}
If (counts > countl) cout << “\n more small”;
else cout << endl << “more capital”;
```


- Keep reading numbers of type integer until two equal consecutive numbers are entered. At the end print the average of all entered numbers, the count of even numbers, and the count of negative numbers. Assume that at least two numbers will be entered.
- Example input -2 -10 7 9 -11 30 6 7 7.

```
int countn =0; counte, count =0;
int last, cur;
cin >> cur;   last = cur-1;
// continue next page
```

```
while(1) {
  if (last == cur) break;
  else if (cur < 0) countn++;
  else if (cur%2 ==0) counte++;
 sum+=cur; count++;
 last = cur;
 cin>> cur
}
cout << countn << counte;
cout << “average is “ << sum/count;
```


- Write code that keeps reading numbers and stops when a negative number is entered. Based on the numbers entered, the program should print the smallest and largest numbers entered.

Solution:

```
int num, small, large;
cin >> num;  small = large = num;
while (num >0) {  cin >> num;
       if( num < small) small = num;
       if (num > large) large = num;}
cout <<“the smallest numbers is”<<small<< endl;
cout << “the largest numbers is” << large;
```

Computer Programming

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

Arrays: One and Two Dimensional

Arrays

- So far we have only used scalar variables:
  variable with single value
- But many things require set of related values: student grades in exam, letters in a aname.
- Arrays are used to store a collection of related values. Example 100 int values.
- Instead of declaring an individual variable to to each element in the array. We declare one special variable to all to all the elements.

Array Declaration

- Array declaration has the format:
  data type arrayname[size];
  where size is an integer constant > 0 that represents the maximum number of values (elements), that you want to store in the array.
- Each element of the array is like little varaible
- All elements of the array are of the same type.
- To reach an element use arrayname[index]
  where index in the range 0 to size -1

Example

- Suppose I want to store grades for 100 students.
  Then I need to do either:
- Declare 100 scalar variables score1, score2, ... score100.
  In this case I need 100 cin statements to read the graders from the keyboard:
  cin >> score1
  cin >> score2
  ....
  cin >> score100

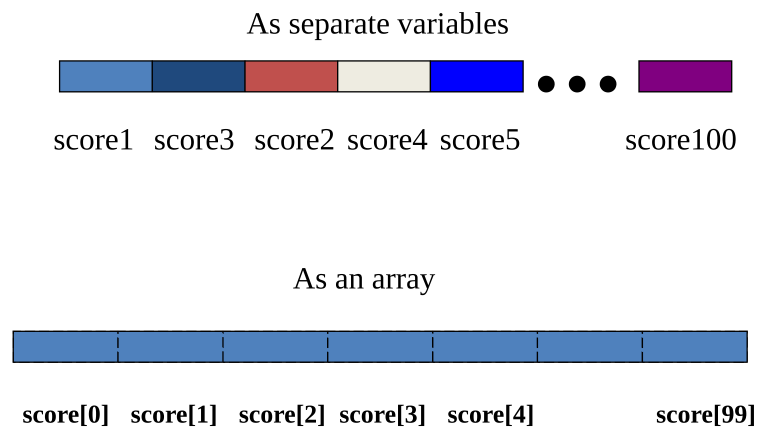
This is a tedious process and makes programming boring and difficult.

Example continue

- Instead, I could declare the array, int score[100]; which allocates space for 100 integer values.
- Using a loop and one cin statement I can read the 100 scores into the array as follows:
- for (int i =0; i < 100; i++)

  cin >> score[i];
- This technique allows me to declare an array of very large size and be able to process the array using loops.

Schematic of Memory



Some Constraints

- Array name must be valid variable name.
- Array size must be constant value > 0.
- Examples of valid array declarations:
  double salaries[100]; char name[30];
- Examples of invalid array declarations:
  int x; double salaries[x]; // not constant
  char name[-5]; // constant < 0
  int players[]; // must have a constant
  double average[5.5]; // must be integer constant

Accessing Array Elements

- The individual elements of an array may be accessed by the array name and the subscript or index.

  score[3], score[78], score[99]
- Subscripts in C++ begin with zero, so the first element of the array with size n has the index or subscript 0 and the last element has the index or subscript n-1.
- Any subscript outside this range (< 0 or >= n) is an invalid access to array element.

Array Types

- All array elements are of the same type.
- We can declare an array of integers
  int IA[40];
- We can have an array of doubles
  double DA[50];
- We can have array of characters
  char CA[30];

Special case: strings array of characters that ends with special characters called null '\0'

- You can have an array of any other data type:
  float, short, long,….

Storing Elements in an Array

- Array elements may be input one at a time

```
score[0] = 78; // assignment
score[5] = 80;
cin >> score[81];
cin >> score[66];
```

- Or use a loop:

```
for (int j = 0; j < n; j++)
    cin>>score[j];   // from the keyboard
for (int j = 0; j < n; j++)
    score[j] = j *2;  // using an expression
for (int j = 0; j < n; j++)
    score[j] = 70; // same value in all locations
```


Initialization at Declaration

Array elements can be initialized at time of declaration.

float temps[4] = {78.5, 79.8, 85.4, 86.2};

If you are initializing the elements at the point of declaration the size declarator may be omitted.

float temps[] = {78.5, 79.8, 85.4, 86.2};

the size will implicitly be set equal to the number of elements specified (four here).

float temps[]; // is invalid must give size if not initialized

float temp[4] = {1,6.5,88.2,19,72,31}; // invalid size is too small

Initialization at declaration continues

- int x[10] = {4,5,6}; Store 4,5,and 6 in the first 3 locations respectively and 0 in the rest
- char name[3] = {'M','A','Y'}; array of characters
- Char name[4] ={'M','A','Y','\0'}; string
- char name[10] = {'M','A','Y'}; // string all elements at locations 3,4..9 are filled with '\0'
- char name[10] = "MAY"; string like above
- double values[5] = {1.2, 4,5, 6.2}; locations 3, 4 are filled with 0.0
- char name[] = "Ahmad"; // size is 6
- char name[5] "Ahmad"; // invalid size is small

Declarations and Initializations Continue

- int x[3] = {10,5,13,4,6}; invalid size is too small for specified elements.
- int x[10] ={4,5,,6,7}; is invalid only last values in the array may be omitted. The correct declaration in this case is int x[10] = {4,5,0,6,7};
- int X[10]; x = {4,5,6,7}; is invalid because declaration and the initialization are done in two separate statements .
- char x ="abc"; // x is a single char not array

Strings

- Strings are arrays of characters.
- Strings have some special properties that numeric arrays do not have.
  - The size should allow for the null character '\0'
  - Input may be an entire array at a time

    cout<<"Please enter a filename"<<endl;

    cin>>filename

    char name[6] = "Ahmad";   //valid

    char name[10] = "Ahmad"; //valid

    char name[5] = "Ahmad"; // invalid size is small

Outputting Data from an Array

- Array elements can be outputted individually

  cout<<score[0]<<" "<<score[5];
- Or using a loop

  for (int j = 0; j < n; j++)

  cout<<score[j]<< endl;
- All elements of arrays cannot be outputted by just using the array name.

  cout<<score;   // will not work!

  exception to this if the array is a string

  char B[10] = "ahmad"; cout << B; // output entire string in one cout statement

Array Declaration Examples

- Define arrays of types int, char, double and floats each of size 10 elements.

  Solution: int A[10];   char B[10];

  double C[10];   float D[10];
- Declare and initialize an integer array of 100 elements that contains {6,12,4,9,15,0,0….0}.

  Solution: int IA[100] = {6,12,4,9,15};
- Declare and initialize an array of 100 doubles that contain {0.0, 2.0, 4.0, 6.0,…198.0}.

  Solution: double DA[100];

  for (int i= 0; I < 100; i++) DA[i] = I * 2;

Examples Continue

- Define an arrays of characters size 26 elements. Store all small letters in the array.

  solution 1: char CA1[26];

  for (int i = 0; i < 26; i++)

  CA1[i] = (char) ('a'+i);

  solution 2: char CA2[26];

  for (char ch ='a'; ch <= 'z'; ch++)

  CA2[ch – 'a'] = ch;

- Declare an array of 10 characters store "world" in it. solution: char st[10] = "world";

Examples Continue

- Declare and initialize a string with the word "palestine". Array size is selected automatically solution char st2[] = "palestine";
- Declare an Array of 100 doubles Then read values into the array from the keyboard. Then multiply each read value by 2 and print the final result in the array starting at location 99 then 98 … down to 0.

  Solution:

```
double A[100];
for (int i =0; i < 100; I++) cin >> A[i];    //read data
for (i = 0; i < 100; i++) A[i] *= 2;        //process data
for (i = 99; i >=0; i--) cout << A[i] << endl; //output data
```


Caution!

- C/C++ has no checking on the bounds of the array. Outside the range 0 to n – 1
- Therefore you c++ allows you to store elements with subscripts larger than the size declarator. However, these values could be corrupted because the memory spaces have not been officially reserved. If you try to access such elements you may get **run time error.**
- Retrieving data stored in out-of-bounds elements is compiler and/or platform dependent.

Array Processing

Array elements can be used in assignment statement, output statement, and to perform operations on them, in a similar manner as individual variables.

```
Int  IA[10] = {1,2,4,};  int x, y =12;
x = y * IA[2];
IA[4]= sqrt(IA[3]) + x;
for ( IA[0] =5, int j =1; j < 10; j++)
    IA[j] = IA[j-1] * 2;
for (int sum =0, int j=0; j<100;j++)
     sum += IA[j];
```

Examples

Declare an array of grades for 50 students. Ask the user to enter grade for each student compute and display the average.

```
int grades[50];     int i, sum =0;
for (i=0;i<50;i++) {
    cout << "Enter grade:";      cin >> grades[i];
    sum = sum + grades[i]; }
cout << "Average is:" << sum/50.0 ;
```

More processing: int fail =0; int pass =0;

```
for ( i=0; I < 50; i++)  if (grade[i] >= 60) pass++;
cout <<pass<<" passed";
for ( i=0; I < 50; i++)  if (grade[i] < 60) fail++;
cout << fail << " failed";
```


Examples 2

Given an array of 100 int values called A, compute the smallest, and the largest values in the array.

Solution:

```
int smallest =0; int largest = 0;
for ( int i = 0; i < 100; i++) {
   if (A[i] < smallest)  smallest = A[i];
   if (A[i] > largest) largess = A[i];
}
cout << "The smallest value is " << smallest << endl;
cout <<  "The largest value is " << largest;
```


Examples 3

Read 120 int values from the keyboard store all negative values in an array called negative and all positive values in an array called positive. Select appropriate array size for negative and positive arrays.

Soultion:

```
int positive[120]; int negative[120]; int num;
int i,j,k; i = j = k =0;
for (; i < 120; i++) {
   cin >> num;
    if (num >= 0) positive[j++] = num;
    else              negative[k++] = num;
}
```


Examples 4

- Given an array of 1000 characters called AC, compute the count small letters in the array.

```
for ( int csmall =0, int I = 0; I < 1000; i++)
     if ( AC[i] >= 'a' && AC <= 'z") csmall++;
```

- Given an array of int values of size 100 (AI) print the average of all 3 digit values in the array.

```
double sum =0; int count =0;
for (int I =0; I < 100; I++)
     if (AI[I]>99  &&  AI[I]<999){sum += AC[i];  count++};
cout << "The average is" << sum/count;
```

Examples 5

- Define an array of 100 int and store the sequence:

  1 2 3 5 8 13 21 ….

  int A[100]; A[0] = 1; A[1] = 2;

  for (int i =2; i <100; i++)

  A[i] = A[i-1] + A[i-2];

- Write a program to do each of the following:

  Given an array of 15 double values called DA.

  Print the index of the smallest value in the array

  The larges tod value.

  Print the sum of all values that are > 100.0

  Replace all negative values with 0.0 in the array.

  Print only array elements that are even and 2 digits

Examples 5 continue

```
int i =0, locsmall = 0, small = DA[0], sum =0, large = DA[0];
 for (int i =0; i < 15; i++) {
   if (DA[i] < small) { small = DA[i]; locsmall =i;}
   if (DA[i] %2  && DA[i] > large) large = DA[i];
   if (DA[i] > 100) sum+= DA[i];
   if(DA[i] < 0) DA[i] = 0;
   if(DA[i] > 9 and DA[i] < 100 && DA[i]%2 ==0)
       cout << DA[i] << endl;
}
cout << “loc = “ << locsmall   <<  “ sum = “ << sum << endl;
cout << Largest value in the array << “large”  << endl;
```


Two Dimensional Arrays (Matrices)

- **Two subscripts `a[ i ][ j ]` are used to reference an element in the matrix**
- **When declared must define number of rows and columns example: `double B[5][6];`**
  - Matix with rows and columns
  - Specify row, then column
  - “Array of arrays” ex a[3][4];
    - **`a[0]`** is an array of 4 elements
    - **`a[0][0]`** is the first element of that array

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Array name   Row subscript   Column subscript

Matrix Declaration and Initialization

- Define a matrix of types int, char, double and floats each of size 6X4 elements.

  int im[6][4];   char cm[6][4];

  double dm[6][4];   float fm[6][4];

- Define a 4x4 int matrix where each cell contains the sum of row and column indices

```
int  a[4][4];
for (int i =0; i < 4; i++)
    for (int j =0; j < 4; j++)
        a[i][j] = i+j;
```

Matrix Initialization

- A matrix can be initialized at declaration:

  Default of 0 (int), or 0.0 (double) or '\0' (char) for not specified values like the one dimensional array

  – Initializes grouped by row in braces

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

Row 0	1	2
Row 1	3	4

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

Row 0	1	0
Row 1	3	4

When initializing a matrix at declaration the number of columns must be specified but number of row may be not specified.

Initialization at Declaration

int M[3][3] = {{1,2,3},{4,5,6},{7,8,9}};

int [][3] = {{1,2},{3},{4,5,6}}; specifies a matrix with three rows and three columns missing values are filled with 0. {1,2,0},{3,0,0},{4,5,6}

int A[][] = {{1,2},{3},{4,5,6}}; is invalid declaration because number of columns not defined

int A[2][] = {{1,2},{3},{4,5,6}}; is invalid declaration because number of rows must be at least three

int A[4][4] = {{1,2},{3},{4,5,6}}; valid declaration fills missing values with 0s. {{1,2,0,0},{3,0,0,0},{4,5,6,0},{0,0,0,0}}

Reading and Printing Matrix Elements

- When reading elements from the keyboard must read it one item at a time: Int A[10][20];

```
for (int r =0; r < 10; r++)        // for each row
  for (int c =0; c < 20; c++)      // for each column in  a row
       cin>> A[r][c];              // read elements
```

- When printing elements on the screen must print it one item at a time: Int A[10][20];

```
for (int r =0; r < 10; r++) {          // for each row
   for (int c =0; c < 20; c++)         // for each column in  a row
       cout << A[r][c] << " ";         // print element
   cout << endl;                       // print end line
}
```


Read and Print Examples

Give int M2[6][4];

Read elements and store in matrix row0 then row1 ...

```
for (int r= 0; r< 6; r++)
    for (int c = 0; c < 4; c++)
        cin >> M2[r][c]
```

Print the elements in the matrix row0 then row1 …

```
for (int r = 0; r < 6; r++) {
    for (int c = 0; c < 4; c++)
        cout << M2[r][c];
    cout << endl; }
```

Read and Print Examples 2

Read Elements and store column0 then column1 ....

```
int M2[6][4];
for (int c = 0; c < 4; c++)
    for (int r = 0; r < 6; r++)
        cin >> M2[r][c];
```

Print the Matrix elements column0 then column1 ....

```
for (int c = 0; c < 4; c++) {
    for (int r = 0; i < 6; r++)
        cout << M2[r][c];
    cout << endl;
}
```


Important Notes

- int A[][]; or int A[][4]; or int A[5][]; are invalid declarations; must specify both rows and columns when declared only (not initialized at declaration)
- Given int A[Row][Col]; Each element is specified using row and column indices. Range 0..Row -1 , 0 .. Col -1
- Example: Given int B[3][5]; Then B[1][3] references the element in 2nd row and 4th column
- To traverse all elements in the matrix need two loops. One loop traverse the rows the other loop traverse the columns.

Matrix Examples

Given 10X10 matrix M of type int.

Print the values of first and last elements in M.

```
cout << "first " << M[0][0];
cout << "last " << M[9][9];
```

Print the value of the first element in row 3 of the matrix.

```
cout << M[3][0];
```

Print the value of row 2 column 4

```
cout << M[2][4];
```

Store 110 in element at 4th row and 5th column

```
M[3][4] = 110;
```

Add any two locations and and store result in another location

```
M[2][4] = M[1][2] + M[6][3];
```


Matrix Example 2

Store 0 in the diagonal elements in the matrix.

```
for (int i = 0; i < 10; i++) M[i][i] =0;
```

Store 5 in the last element of every row // last column

```
for (int i = 0; i < 10; i++) M[i][9] = 5;
```

Print in the first element of every column // first row

```
for (int i = 0; i < 10; i++) cout << M[0][i] << endl;
```

Print the elements in row 5.

```
for (int i = 0; i < 10; i++) cout << M[5][0] << endl;
```

Find the sum all the elements in column 6

```
for (int i = 0, int sum =0; i < 10; i++) sum+=M[i][6];
```

Matrix Examples 3

Compute smallest, largest and average value in M.

```
int small = M[0][0]; int large = M[0][0]; int sum =0;
 for (int r =0; r < 10; r++)
     for (int c= 0; c < 10; c++) {
         if(M[r][c] < small) small = M[r][c];
         if(M[r][c] > large) large = M[r][c];
         sum + = M[r][c]; }
cout << "smallest - largest" << small << "-" << large;
cout << "the average value " << sum/100.0;
```


Matrix Examples 4

Compute the count of negative values, the sum of all even values and product all values > 10.

```
int countn =0,  sume =0; int product10 =1;
  for (int r =0; r < 10; r++)
    for (int c= 0; c < 10; c++) {
        if (M[r][c] < 0) countn++;
        if(M[r][c] %2 == 0) sume+= M[r][c];
        if(M[r][c] > 10) product10 *= M[r][c]; }
 cout << "count negative =" << countn << endl;
 cout << "sum even =" << sume << endl;
 cout << "product greater than 10 =" << product10;
```


Matrix Examples 5

- Swap elements in row 2 with elements in row 5

```
for (int c =0, int temp; c < 10; c++) {
     temp = M[2][c];
     M[2][c] = M[5][c];
     M[5][c] = temp; }
```

- Compute the sum of each row and store it in array

```
int Asums[10] = {};
for (int r =0; r < 10; r++)
     for (int c = 0; c < 10; c++)
         Asums[r] += M[r][c];
```


Arrays With Higher Dimensions

- Same arguments is extended to arrays of higher dimensions
- int A[3][4][5]; is an array of 3 dimensions
- int A[4][5][6][7]; is an array of 4 dimensions.
- When referencing an element in k dimensional array k subscripts must be used.
- For this course we only deal with one and two dimensional arrays.

Computer Programming

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

Pointers

Definition

- Pointers are variables used to hold (and to refer to) memory addresses of other variables.
- Memory addresses is the location of the first byte of memory allocated for that variable or array. Remember that the amount of memory allocated to a variable is dependent on the data type of the variable.
- You can learn the memory address of a variable or array by using the address operator, (the & symbol), before the variable name.

Definition continue

- A pointer variable is designated at time of declaration by a * before the variable name.
  - int *pntr; // or int* pntr;
- After declaration, a * immediately before a pointer name acts as an indirection operator to refer to the value stored in memory, and consequently, may be used to change what is stored in that memory location by an assignment statement.

```
 cout << *pntr;
*pntr += 5; //adds 5 to value pointed to by pntr;
```


Pointer Declaration

- Pointers are declared by specifying the type of location (variable) they point at..
- Syntax:

  type * name; // compiler allocates 4 bytes

  For all data types pointer size is 4 bytes
- For example: // compiler allocates 4 bytes for each of the following pointer declarations.

  int *p; char *tp; double *dp;
- When a pointer is declared it points to null (not valid memory location)

Pointer Initialization 1

- Before using a pointer to store or retrieve data from memory location it must be initialized to a valid memory location.
- Valid locations are either:

  exist: (static) variables of the same type

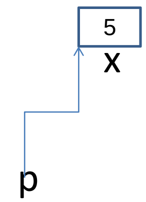
  new: (dynamic) new locations of same type
- Example: int x;   int *p;

  p = &x; // p now points to static location x, this location is accessed by x or *p

  p = new int; // p points to new location this location is accessed by *p only;
- To delete memory allocated by new use: **delete p**;

Pointer initialization 2

- Suppose `p' is a pointer, then `*p' is the value in memory location which `p' points to.
- Example:



int x =5; int *p;

// make p point to location x

p = &x; then

The location x can be accesses using *p or x.

cout << *p; // prints 5 on the screen

*p = 10; cout << x ; // prints 10

Pointer Initialization 3

- When a pointer is initialized it must point to a location that can hold data of the same type;
- Example: Given   int *p1; double *p2; char *p3;

  int x;   double y;   char z;
- The following is valid initialization:

  p1 = &x;   p1 = new int;

  p2 = &y;   p2 = new double;

  p3 = &z;   p3 = new char;
- The following is invalid initializations:

  p1 = &z;   or p1 = &y;   or p1 = new double;

  p2 = &x;   or p2 = &z;   or p2 = new char;

Pointer Initialization 3

- Pointer can only point to one location at a time;
- Example: Given int x =5 ,y = 10;   int *p;

  p = &x; cout << *p;   // prints 5

  p = &y; cout << *p;   // prints 10

  p = new int; cout << *p; //prints random val
- More than one pointer may point to the same location. Example Given: int x; int *p1,*p2;

  p1 = &x;   p2 = &x;   // p1 = p2 = &x;

  *p1 = 10;   then

cout << x;   or cout << *p1;   or cout << *p2;

All prints the same value 10 because they all reference the same location.

Pointer Examples 1

Given int x =5, y =10 ; int *p1, *p2;

what is the output of the following if valid?

```
p1 = x;      // invalid  must use p1 = &x;
y = *p2;    // invalid p2 must point to valid location
p1 = p2 = &x;    cout << *p2;       // 5
p1 =&y;   cout << *p1;    cout << *p2;  //10   5
p1 = &x;    x = y;  cout << *p1;  //  10;
p1 = &x;  p1++;    cout << *p1;  // causes run time
    error because p1 points to invalid location
p2 = &y; *(p2)++; cout << *p2;   // 11
```


Pointer Examples 2

Given: double *dp, d; char c ='A'; What is valid?

```
cin >> dp;                  // invalid use of dp
cin >> *dp;                 // invalid location of dp
dp =&d;    cin >> *dp;          // valid
d= &dp;   // not valid assignment need cast
dp = new double;    d =*dp;      // valid
*d=dp;       // invalid do not use * with variables
dp = &c;    // invalid  dp and c of different types
dp = new double;  *dp = c;    or c = *dp;   // valid
dp = &d;     *dp = *dp + c;       // valid
```


Pointers and Arrays 1

- Remember that an array name refers to a group of memory addresses, not just a single one.
- The array name holds the address of the first byte of memory allocated to that array.
- Therefore, an array name, without the index, may be considered a pointer to that array.
- Array name is a static pointer and can only point to the location assigned to it.
- Because the array name already acts as a pointer you do not need the address operator, &, to output the first memory address.

Pointers and Arrays 2

- The indirection operator, *, may be used with the static array name to store a value in the first array element.

```
*arrayname = 15;       //stores 15 in arrayname[0]
```

- Values can be stored in subsequent array elements by adding numbers to the array name and employing the indirection operator

```
*(arrayname + 1) =25; //stores 25 in arrayname[1]
*(arrayname + n) =67; //stores 67 in arrayname[n]
arrayname = arrayname +5;  // invalid can not make
arrayname point to any location other than first
*arrayname = *arrayname + 5;   // valid add 5 to
   arrayname[0]
```

Pointer Arithmetic

Integer values may be added to or subtracted from a pointer to move it to different locations

Example: Given int A[5]; int *p;

```
p = A;  // p = &A[0];   // p points to first elem of A
p = A; then  p = p+2;  // now p points to A[2];
p = &A[2];   p--;         // now p points to A[1];
p = A; p = p+5; // p points outside the array A
p = A; p +=2;  p[1] = 18;  // stores 18 in A[3];
p = A+5;  p[-1] = 20;  // stores 20 in A[4];
```

Important Note: With A index is absolute from A. With p index is relative to pointer location

Pointer Arithmetic 2

- One pointer may also be subtracted from another pointer.
- Example Given int A[10] = {1,4,6,10,12,20,22};

  int *p1 = &A[2]; int *p2 = &A[5];

  cout << p2 – p1; **prints 3** // number of elements between p1 and p2;

  cout << *p2 - *p2; **prints 14** // the difference between the values pointed to between p2, p1
- Pointers can not be added p1 + p2 has no meaning in C. // it is an int number

  p1 = p1 + *p2; // move p1 by the value *p2(+)

  p1 -= *p2; // move p1 by the value *p2 (-)

Other Pointer Operations

- Pointers can be initialized at time of declaration.

  float *fp= &fvar; // fvar must exist 1st
- Pointers, and memory addresses, may also be compared using the relational operators.

  p1 < p2 ; p1 != p2; p1 ==p2
- Static pointers can not be incremented
- Int A[10]; A= A+5; or A++; or A--; A[-1];

  are invalid operations
- Since A is a static pointer it must always point to the same location (first element of the array) and can not be moved.

Static Arrays and Pointers 1

int A[10] = {12,4,7,10, 13, 16};

int *p1 = A; Then

A[0], *A, p1[0] and *p1 all used to refer to the first element of the array (12 in this case).

int *p2 = A + 2; now p2 points to A[2]. Then

A[1] and *(A+1) , p2[-1], and *(p2-1) all have the value (the second element, 4 in this case)

Int *p3 = &A[4]; p3 = p3-2; Then

A[2], *(A+2), p3[0], and *p3 all refer to the third element in the array (7 in this case).

Static Arrays and pointers 2

Given int A[10] = {12,4,7,10, 13, 16}; int *p = A;

Then we can print array elements as follows:

for (int i =0; i < 10; i++) cout << A[i];

for (int i =0; i < 10; i++) cout << *(A+i);

for (int i =0; i < 10; i++) cout << p[i];

for (int i =0; i < 10; i++) cout << *(p+i); //not *p+i;

for (int i =0; i < 10; i++) {cout << *p; p++;}

Note that:

for (int i =0; i < 10; i++) {cout << *A; A++;} is invalid

Dynamic Arrays

- So far we have allocated arrays statically.

  Example int AI[20]. In which case array size is fixed at compile time to 20 and can not change.
- However, C allows us to declare a pointer and make it point to consecutive memory locations (array) at run time.
- Example int *ap; Then we write:

  ap = new int[30]; where we allocate array called ap and we can use ap to refere to any element in the array like ap[i]; i< 30; or *ap, or *(ap+i)

Dynamic Arrays 2

- Array size may be entered by the user at run time.
- Example: double *dp; int size;

  cout << "please enter array size:";

  cin >> size;

  dp = new double[size];
- You can access array elements using static method or dynamic method, dp[i] or *dp; dp++; …etc
- To delete memory allocated to the array we use the statement delete [] dp;

Dynamic Arrays 3

- Be carful: In dynamic arrays if array pointer is made, accidentally, point outside the array then the pointer can not be made to point again to the array.
- Example given int *p = new int[10]; int x;

  p = p+15; p now points outside the array and can not make it point to the array again.

  p = &x; p now points to memory location x and can not make it point to the array again.

  p = new int; p now points to new memory and can not be made to point the array again.

Example 1

- Given int x; int *p1, *p2; make p point to x;
  and make p2 point to new location.
  solution: p1 = &x;   p2 = new int;
- Given: double *dp, d; Which one is valid

a. dp =&d;   b. d = *dp;   c. d=&dp;

- Given int s1[5] = {3,5,6};   int s2[5];   int *p;
  which of the following is valid and why?

a. s2 = s1;   b. p= &s1[6];
c. s2 = *p;   d. p = s1+1;

Examples 2

- Given int x[]={0,2,4,6,8,10}; int *y;
  y = &x[2];   *y += 1;   *(y+2) +=2;
  what is the new array content: {0,2,5,6,10,10};
- Given float x[10]={2.5,3.5,4.5,20.0,0.0,6.5};
  float *p= x + (int) *x; float sum=0.0;   int i =0;
  while(*p && i < 10){ sum+=*p++; i++};
  cout<<sum;   what is printed?   24.5
- Given int x[6] = {1,5,3,4,0}; int *p= x+1;
  for(int i= *p; I > 0; i -=*x){cout<<"hello";}
  how many time hello is printed 5

Examples 3

- Given an Array A of 100 int values, and int *p;
  Write code to: Print array content using static pointer A.

```
for (int i = 0; i < 100; i++)
    cout << A[i];   // cout << *(A+i);
```

Print array content using pointer p;

```
for(int i =0, p = A; i < 100; i++)
  cout << p[i];   //cout << *(p+i);
// {cout < *p;   p++;}  This is not possible with A
```


Examples 4

Given an Array A of 100 int values, and int *p;

- Use p to replace all even values in A with 0.
  for(int i =0, p = A; i < 100; i++)
  { if (*p%2 == 0) *p = 0; p++;}
- Use pointer p to print all elements in the array in reverse location 99 ,98,…0
  for(int i =0, p = &A[99]; i < 100; i++)
  {cout << *p << endl; p--;}
- Use p to sum all elements in A until first 0.
  for(int sum =0, p = A; *p; p++) sum = *p;

Examples 5

Given an Array A of 100 int values, and int *p;

Declare two pointers p1 and p2

(1) Make p1 point to the first negative value in A.

(2) Make p2 point to the last even value in A.

(3) Find the count elems. between p1 and p2 **inclusive**.

(4) Find the sum of values between p1 and p2.

Solution: int *p1, *p2;

(1) p1 = A; while(1) { if (*p1>=0) p1++; else break;}

(2) p2 = A+99; while(*p2%2) p2--;

(3) int elemcount = p2 – p1+1;

(4) For (int sum=0, p = p1; p <=p2; p++) sum += *p;

Examples 6

Given an Array A of 100 int values, and int *p;

Declare two pointers p1 and p2 make p1 and p2 point to array locations index 20 and index 50 respectively, then (1) use pointer p to print all odd values between p1 and p2 **exclusively.** Then (2) copy all values between p1 and p2 into new dynamically allocated array.

(1) int * p1 = &A[20]; int *p2 = &A[50];

for (p = p1+1; p !=p2; p++) if (*p%2) cout << *p;

(2) int *t = new int [p2 – p1 -1];

for (p = p1+1; p !=p2;) *t++ = *p++;

Example 7

- Declare a dynamic array of N double values where N is entered by the user then read the N double values and store them in the array. Multiply each value in the array by 2. Then print the new content array in reverse.

Solution:

int N; cin >> N; int *p = new double [N];

for (int *t =p, int i =0; i < N; i++, t++) cin >> *t;

for (int *t =p, int i =0; i < N; i++, t++) *t+=2;

for (int *t =p+N; t >= p;) cout << *t-- << endl;

Important points

- Pointer declaration: type * name;
- Pointer must point to a location before it is used.
- Pointer type and location type must be the same.
- Pointers can be subtracted. But can not be added.
- Static arrays (pointer) point to the first element.
- Static pointers can not change location it points to.
- Dynamic pointers can point to any valid location.
- Dynamic pointer can not only point to two or more locations at the same time.
- Two or more pointers can point to same location.

Computer Programming

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

Functions

Introduction

- Computer programs that solve real-world problems are usually much larger than the simple programs discussed so far.
- To design, implement and maintain larger programs it is necessary to break them down into smaller, more manageable pieces or *modules*.
- Dividing the problem into parts and building the solution from simpler parts is a key concept in problem solving and programming.

Introduction Continues

- In C++ we can subdivide the program into blocks of code known as **functions**. In effect these are subprograms that can be used to avoid the repetition of similar code and allow complicated tasks to be broken down into parts
- Until now we have encountered programs where all the code (statements) has been written inside a single function called main(). Every executable C++ program has at least this function.

Function Definition

- A function Is comprised of <u>heading</u> and a <u>body</u>. Following is the syntax of function definition.

```
return type function name (data type parameter(s))
{
     statement(s);  //function body
}
```

```
void main ( )
{
     statement(s);
}
```

```
int main ( )
{   statement(s)
    return (exp);
}
```

Examples

```
void print2lines ( )
{ cout<<"*************************\n";
  cout<<"*************************\n";
}

int sum2int (int a, int b) // params are declared
{    int c;
     c = a + b;
     return (c);}
```


Function Return Type

- May be void, indicating that nothing will be returned, or any of the data types (int, float, double, char,…, or a pointer to type)
- When the return type is any thing other than void, a return statement must be part of the function body. In this case the function return a value (Value - Returning functions)
- When functions are part of a conditional expression they can not be void. Example
  If (pow(2,3) > 5) …..

Return Statement and its Placement

- Are included in value-returning functions
  (non-void functions).
- Terminate the execution of statements in a function. Any statements after the return statement are not executed.
- Send a single value back to the calling function may be main or another function.
- int f1(int x) { x = x *2; return x; cout << x;}
  the cout statement never executed

Function Placement

- a function may be placed before main or after main. If it defined after main, it requires function prototypes. Example:

```
int f1() { int y; ……;return();}   // impl.
double f2(int);                   // proto
void f3(char);                    // proto
void main(){…. F1…… F2…. F3};     // impl
double f2(int z) {……return();} // impl
void f3(char z) { ….};            // impl
```

Function Prototypes

- Act similar to a variable declarations.
- Occur before its called in the file (before main).
- Look similar to function heading except that it ends in a semicolon and it does not require the parameter name, just the data type.

```
void print2lines ( );
int sum2int (int, int);
```

- If the parameter name is included, it is ignored.

Function Format

```
//comments
# include library files
Global variables
function definition(s)
{
   statement(s)   //body
}
void main ( )
{
   statement(s)
}
```

```
# include library files
function prototype(s)
global variables
void main ( )
{
   statement(s)
}
function definition(s)
{
statement(s)  // body
}
```


Calling Functions

- A function call is the statement that tells the function to execute.
- When execution of the function has been completed, flow of the program returns to the statement immediately after the call.
- It has the format of
  function name (parameters);
- The parameters from the call are transferred to the function heading.
- Function calls for value-returning functions are often part of assignment statements.

Examples of Function Calls

```
#include <iostream.h>
int readint() {int x; cin >>x}
void printint(int x) {cout << x << endl;}
int sum2int( int x, int y){ return(x+y);}

void main ( ) {
   int a,b,c;
    a = readint();   b=readint();
    c = sum2int(a,b);
     printint(c);  }
```

More on Function Calls

- A function can call another function or even itself (recursive functions)
- A single C++ statement may call more than one function. Example: sqrt(x) + pow(x,y);
- Function calls can be part of a condition.

  if (sum2ints(6, 12) > 10) {statements}
  if (function()) {statements}
- Function could be part of an expression

  X + function() ……

  int y = function();

Parameters

- The parameters in the function heading are referred to as "formal parameters" or arguments.

  int add(int x, int y) { return(x+y);}
- The parameters in the function call are referred to as "actual parameters" or arguments.

  int w = 10; cout << add(4,w);
- When the function call is executed, the actual parameters are transferred to the formal parameters in order that they appear in the call.
- We will discuss some restrictions later.
- Formal parameters and actual parameters are different variables even if the have the same name

Passing Parameters

- **Pass by value** – a copy of the value from the actual parameter is sent to the formal parameter of the function. The function can not change the value of the actual parameter.
- **Pass by address (pointer)**- the formal parameters point to the actual parameters. The function can change the value of the actual parameter.
- **Pass by reference** – the same memory location is shared by actual parameter and the formal parameter. The function can change the value of the actual parameter.

Pass by Value

- Only the value of the actual parameter stored in the formal parameter.
- The actual parameters and the formal parameters are separate memory locations
- Example:

```
int f1(int x) {  x = x/2; cout << x;  return(x);}
void main() { int y =10;
              cout << f1(y) << endl;   // 5;
              cout << y;           //  10
}
```

Pass by Value 2

Pass by can be generalized to include passing the value of any expression to the function.

int f2(int x) { x = x/2; return x;}

In main() we can call the function as follows:

```
void main() {  int y =10;
        cout << f2(y);              // 5
        cout <<  f2(30);            // 15;
        cout << f2(25 * 2 + y);     // 30
        cout << f2(f2(y)/2 + 15);   // 10
    }
```


Passing by Address

- The address of a variable is passed to the function and the function access the variable using a pointer.
- Pass address of the variable using **&** operator
- Define the formal parameter as pointer in the function
- **Use** * operator to access and make changes to the value of the variable from inside the function.

void f3(int *x) { *x =5};

void main{ int y =10; f3(&y); cout << y;}

// it prints 5 because the function call changes the content of location y in main using pointer x.

Pass by Reference

- Reference is defining another name to previously defined variable. Reference declaration syntax:

  type & name = variable;

  int x =5, y=10;   int & z = x;
- Now both x and z are names for the same location.

  x → 5 ← z   // the location can be accessed either using the name x or the name z.

  cout << z;   //5   z= 20;   cout << x; // 20
- Int & z = y; // syntax error already defined z.
- Int & w; // syntax error must assign to variable;

Pass by Reference 2

- We can use references to pass variables to functions by defining another name in the function to the variable passed.
- Use reference name in the function to access and make changes to the value of the variable from inside the function.

void f3(int &x) { x =5};

void main{ int y =10; f3(y); cout << y;}

// it prints 5 because the function call changes the content of location y in main using reference x.

Function call Example

- Suppose your know the length of the sides of a rectangle and you want a single function to calculate both the perimeter and the area of the rectangle.
- Your function would need 4 parameters, rect_length, rect_width, perimeter, and area.
- The parameters rect_length and rect_width would be passed by value because you do not want the function to change them. The parameters perimeter and area would be passed by reference or by address because the function calculates them and store changes.

Example (cont.)

- The Function Definition would be

```
void calcAreaPeri(float rect_length,
   float rect_width, float &perimeter, float * area)
   {   perimeter = 2 * rect_length + 2 * rect_width;
     *area = rect_length * rect_width; }
```

- The function prototype would be

```
void calcAreaPeri(float, float, float&, float *);
```

- The function call would be: float p, a;

```
calcAreaPeri(4.6, 8.5, p, &a);
```


Example (cont.)

- Write main that calls the function

```
void main ( ) {       float len, width, p = 0, a = 0;
cout<<"Please enter the length and width. \n";
cin >>len >> width;    // len = 4   and width = 6
calcAreaPeri(len, width, p, &a);
cout<<"A rectangle of length "<< len    << endl;   //  4
cout <<  "and width of "          <<width << endl;   // 6
cout << " and has a perimeter of "<< p    << endl;   // 20
cout  <<" and has an area of"<< a<<   endl;         // 24
}
```


Summary of Passing Parameters

- With pass by value, the actual parameter may be a variable, constant , or an expression.
- Pass by reference the actual parameter must be variable name. use & in the function heading.

```
void Func(int & a , float b , char & c ) {    }
main ()   { int x, float y, char z;  Func(x,y,z);}
```

- With pass by address, the actual parameter must be the address of a variable and defined as a pointer in the function heading.

```
Func (int *a,  float b, char *c)  {   }
main ()   { int x, float y, char z;  Func(&x,y,&z);}
```

Passing Arrays to Functions

- An entire array can be passed to a function.
- In the function call, just use the array name without any indices or subscripts.
  - Often the number of elements in the array is passed as another argument of the function call.
  - Printarray(myarray, 30);
- In the function heading, the size of the array should be left blank by using empty brackets
  - void Printarray(float numbarray[ ], int size)

Some Details of Array Passing

- Previously in functions, we discussed that we could pass parameters "pass by value or "pass by reference" or pass by address.
- However, we cannot pass an entire array as pass by value to a function.
- By default an array is passed in a similar manner (to pass by value) as pass by reference.
- Any changes to array content in the function will be permanent.

Example of passing Arrays to Functions

```
void printarray(float B[ ] , int size)
 { for (int i = 0; i < size; i++) cout << B[i] << " ";}

void  mult2array(float B[], int size)
{  for (int i = 0; i < size; i++) B[i] *= 2; }
// all changes to array content will be present  to main,
void main ( ) {
   float A[10] = {2.5,6.9,7.2,13.1,26.5};
   printarray(A, 10); // 2.5   6.9  7.2  13.1  26.5
   mult2array(A,10);
    printarray(A,10);  // 5.2  13.8  14.4  16.2  53
}
```


Returning a pointer to memory

- Instead of returning a value from a function a pointer to a memory location may be returned provided that the memory location allocated dynamically (using new).

```
int * f(int a ) { int b; int * p = new int[10];  return(p)}
void main ()  { int *x; x = f();}
```

Now x points to new memory pointed to by p.

- Never return a pointer to a (temporarily) variable declared in the function heading or body. Because the memory location disappears when the function end.
- In the example above the function should not include the statement: return (&a) or return(&b).

Designing Functions

- When we design a function we should consider the following points:
- What do I want this function to do?
- What parameters do I need and their type?
- Should this function return a single value?
- if so what is the data type of the returned value?
- What local variables need to be defined within this function?
- Etc.

Example 1

- Write a function to print "Hello" on the screen n times where n is passed as parameters. And write main to call the function .

```
void printhello (int n) {
      for ( int i = 0; I < n ; i++)  cout << "hello"; }
void main() {     int x;
                  printhello(5);
                  cin >> x;  printhello(x);
                  printhello(x*2+19);
        }
```


Example 2

- Write a function that takes an int passed by value, a double passed by address and char passed by reference. In your function multiply the double by the int value and store result in the double value. Add the int to the char value and store the result in the char value. Also return the sum of all three variables. write main to call the function;

```
double comp (int a, double *p, char *c) {
   *p = *p * a;    c  = c + a;  return(a + *p + c); }
void main() { int x =5; double y = 12.5;  char z = 'A';
            double w = comp(x, &y, z); }
```


Example 3

- Write a function that takes two characters c1 and c2. Print all characters between them. Your function return the count of printed chacaters. Example if c1 = f; and c2 = p; prints aghijklmnop. returns 11

```
int printchar (char ch1, char ch2) {
  int count =0;
  for (char ch = ch1; ch <= ch2; ch++,count++)
       cout << ch;
 return(count); }
```

Example 4

- Write a function that takes an array of characters and array size. Your function returns a pointer to new dynamically allocated array that contains all small letters in the passed array.

```
char * getsmall (char A, int size) { int scount = 0;
 for (int i =0; I < size; i++)
     if (A[i] > ='a' &&  A[i] <= 'z')  scount++;
 char *p = new char[scount];  char *t =p;
 for (int i =0; I < size; i++)
     if (A[i] > ='a' &&  A[i] <= 'z')  *t++ = A[i];
return(p);   }
```


Example 5

Given: double **M[20][10].** Write a function that takes the matrix **M** as parameter. Your function computes and returns the smallest value in the matrix.

```
double small ( double A[20][10] { //must define # columns
    double small = A[0][0];
    for (int r= 0; r < 20; r++)
        for (int c = 0; c < 10; c++)
            if (A[r][c] < small )  small = A[r][c];
    return(small); }
void main() { double M[20][10]  = {{…},{….},{….}};
            cout << small(M);}
```


Global vs. Local Variables

- A variable that is declared within a block is "local" to that block and may only be accessed within that block. Block is enclosed within { }
- Therefore, a variable declared in a function definition (either heading or body) is local to that function.
- Several functions may use the same identifiers as variable names, but each is stored in a different memory space.

  F1() { int x;} F2() { int x}   main() { int x}
- Global variables are declared outside all functions and may be accessed from anywhere.

  int x; f1 ( ) { x =5;}   f2 () { x +=2;}   main() {x ..

Variable Scope

The variable is known from the point it is defined in a block and any sub block in that block.

A block is the code between {   ……….}

- Example 1: if (1) { int x;   x =5;}   cout << x; // syntax error

  The cout statement accesses x outside the block
- Example 2 : { int x =2 ;

```
        { int y =3;  cout << x;  cout << y;
              { int z = 12;  cout << z << x << y}
        }  // all good
        cout << y;  cout << x;  // syntax error on y
    }  cout << x << y << z;  // x,y and z not known
```

Computer Programming

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

Strings

String Definition

- Series of characters treated as single unit
- Can include letters, digits, special characters **+**, **-**, ***** and any character in the ASCII code
- String literal (string constants) Enclosed in double quotes, for example: **“Palestine"**
- Array of characters, ends with null character **'\0'**
- String name is a pointer that points to the first character of the string.
- String name can be static or dynamic as follows:

char ss[20] =“palestine”; // static pointer
char *ds = “palestine”; // dynamic pointer

String assignment

- Character array **char color[] = "blue";**

Creates 5 element **char** array **color** last element is**'\0‘**
Same as char color[5] =“blue”;
Alternative for character array

char color[] = {‘b’,‘l’,‘u’,‘e’,‘\0’ };

- Variable of type **char ***

char *colorPtr = "blue";
Creates pointer **colorPtr** to letter **b** in string **“blue”**

- **“blue”** is stored somewhere in memory.

String input output

- In addition to initializing string as one unit we can read and print a string as one unit:
- Reading strings: Read string content from Keyboard Given: **char word[20]; then cin >> word**;

Reads characters until whitespace (BLANK, TAB, ENDLINE) is reached. Then appends **‘\0’**.
Can read at most 19 characters.

- Printing strings: display string content on the screen

cout << word ;
prints characters until **'\0‘ is reached**

String Processing

- Given a string char * s = “ABCDE”;
- The statement cout << s; prints ABCDE
- The statement cout << s+2; prints CDE
- s = s+3; cout << s; is a valid operation since s is a dynamic pointer; prints DE on the screen
- for (;*s;s++) cout << s << “ “; prints:
  ABCDE BCDE CDE DE E
- for (;*s;s++) cout << *s << “ “; prints:
  A B C D E

String Processing 2

- Given a string char s[] = “ABCDE”;
- The statement cout << s; prints ABCDE
- The statement cout << s+2; prints CDE
- s = s+3; cout << s; is an in valid operation since s is a static pointer; cannot change s
- for (int i =0; s[i];i++) cout << s+i << “ “; prints:
  ABCDE BCDE CDE DE E
- for (int i =0; s[i];i++) cout << s[i] << “ “; prints:
  A B C D E

String Functions

- Set of build in function in C to manipulate strings. These functions are found in library <string.h>: must #include <string.h>
- Some of the most important functions are:
  - Copy one string to another
  - Compare two strings strings
  - Compute string length
  - Concatenate one string into another string
  - In the next few slides we will study these function
  - There are many more functions in string.h

String functions prototypes

`int strlen( char *s1);`	Returns the number of characters in string s without the null.
`char *strcpy(char *s1, char *s2 );`	Copies the string **s2** into the character array **s1**. The value of **s1** is returned.
`char *strcat( char *s1, char *s2 );`	Appends the string **s2** to the string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`int strcmp(char *s1, char *s2 );`	Compares the string **s1** with the string **s2**. The function returns a value of zero, less than zero or greater than zero if **s1** is equal to, less than or greater than **s2**, respectively.

String Length: strlen

```
int strlen( char *s1);
```

Returns the number of charters in the string without null.

Given char s1[10] =“ABCDEF”; char s2[]=“zyz”;

Cout<<strlen(s1); prints 6.

Cout << strlen(s2); prints 3.

Cout << strlen(s1+2); prints 4

Cout << streln(s2+streln(s2)); prints 0.

Cout << strlen(“12345”); 5

Cout << strlen(“”); prints 0

String Copy: strcpy

```
char *strcpy( char *s1, char *s2 )
```

Copies second argument into first argument. S1 must be large enough to store s2 with the null character.

Given char S1[10] = “ABCDEFG”; char S2 =“XYZ”; Then:

strcpy (s2,s1); is an invalid because s2 is too small for s1;

strcpy (s1,s2); “XYZ” is stored in s1 and s2 is not changed

strcpy(s1+2,s2); s1 becomes ABXYZ;

strcpy(s1,”123456789”); s1 becomes “123456789”

strcpy(s1+2,s2+2); s1 becomes ABZ

strcpy(s2+2,”LMN”); invalid operation

Strcpy (s1,strcpy(s2,”ABCD”); copies ABCD into s1 and s2

Concatenating strings: strcat

```
char *strcat(char *s1, const char *s2)
```

Appends second s2 to the end of s1. Must make sure s1 large enough to store all characters in s1, s2 and null.

returns pointer to s1.

Examples: Given char S1[10] = “ABCDEFG”;char S2 =“XYZ”;

strcat (s2,s1); is an invalid because s2 is too small for s2+s1;

strcat (s1,s2); “XYZ” is stored at the end of s1= ABCDEFGXYZ

strcat(s1+2,s2); s1 becomes ABCDEFGXYZ

strcat(s1+2,s2+2); s1 becomes ABCDEFGZ

strcat(s2,”LM”); is invalid operation becaue s2

strcat(“lm”,s2); is invalid operation because “lm” is constant.

stract (strcat(s1,”1”),”2”); s1 becomes ABCDEFG12

String Compare : strcmp

```
int strcmp( char *s1, char *s2 )
```

Characters represented as ASCII code.

Compares string character by character according to their ASCII code values. Example

S1 =“abc” s2 =“abcd”, s3 =“ABCDEF”, s4 =“123456”;

Returns Zero if the two strings are equal.

Returns Negative value if s1 is smaller than s2

Returns Positive value if s1 is greater than s2.

In examples above s1 < s2, s2 > s3, s4 <s1,s2, s3.

String compare continue

Given char s1[10] = "ABCD"; char s2[] ="ABM"; Then

strcmp(s1,s2); returns a value < 0; since C < M

strcmp(s2,s1); returns a value > 0; since M > C

strcmp(s2, "ab"); returns a value < 0 since A < a.

strcmp(s2,"ABM"); returns 0 since both strings are equal

strcmp(s1+2","CD"); returns 0 since both are equal

strcmp("abc","a") returns > 0 since b > null

strcmp("abc",strcpy(s1,"abc"); returns > 0;

strcmp("s1+2,strcpy(s1,"M"); returns 0

String Function Examples 1

Write code to read 100 strings print the average string size. Assume max string size is 20.

```
char st[21]; int sumall =0;
for (int i =0; i < 100; i++) {
    cin >> st;
    sumall += strlren(st);
}
cout << "average string size is "<< sumall/100.0;
```


String Functions Example 2

Write code to keep reading strings until the string "finish" is entered print the largest entered string. Assume max string size is 20.

```
char st[21]; char maxst[21] ="";
while(1) { cin >> st;
        if (strcmp(st,"finish") ==0) break;
        if (strlen(st) > strlen(maxst))
            strcpy(maxst,st);    }
 cout << maxst;
```


String Functions Example 3

Write code to read 50 strings concatenate them into 1 string. Assume max string size is 20. The compute the strlen of the new string and the count of 'a' in the new string.

```
char st[21]; char all[50*20+1] ="";
 for(int i =0; I < 50; i++) { cin >> st;  strcat(all,st)}
 cout << "the length of all is" << strlen(all);
 int counta =0;
 for (chat *p = all; *p; p++) if(*p == a) couta++;
```

String Functions Example 4

Write a function that takes string s1 and char c1 as parameters. Your function returns the number of times c1 is found in s1.

```
int find(char *s, char c1) {
    int count =0;
    while (*s) {
        if ( *s == c1)  count++;
        s++;  }
    return(count);
}
```


String Functions Example 5

Write function that takes s1, c1 and c2 as parameters your function returns a pointer to new allocated string that contains all characters between c1 and c2 inclusive.

```
char * extract(char *s, char c1, char c2) {
    char *p1 = s, *p2 = s, *p, *ns;
    while(*p1 != c1) p1++; while(*p2 != c2) p2++;
    ns = new char[p2 – p1 +2];
    for (char *t = ns, p = p1; p <=p2;) *t++ = *p++;
    return(ns);  }
```


Array of Strings 1

- We can allocate a matrix of characters and store each string in a given row of the matrix. Following is a code to read student names from the keyboard and store the names in a two dimensional array.

```
char students[40][15];
//40 students, Max length of a name is 14 letters
for (int I =0; I < 40; I++) {
    cout  <<"Enter Student Name:";
    cin >> students[i];
}
```

- // also we can print them one string per line

```
for (I =0; I <40; I++) cout << students[i] << endl;
```


Array of Strings 2

- Given char months[12][20] = {“January”, “February”, “March”, ...,”December”};

Print the number of months starts with M or J

Print the months that are larger than 5 characters

Print the number of months that end with y.

```
Int smj =0;  int ey =0;
for (int i = 0; i < 12; i++) {
  if (months[i][0]==‘M’||months[i][0]==‘J’) smj++;
  if (strlen(months[i]) > 5) cout<<months[i]<<endl;
  if (months[i][strlen(months[i]) -1] == y) ey++;}
```

Computer Programming

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

Structures

Definition

- Structure is a user defined data type, that is build of basic data types (char, int, double,….).
- Structure allow many variables of different types grouped together under the same name.
- To define a structure we use the following format:

```
struct name
{
 type member1;
 type member2;
…
…
};
```


Structure Definition Example

- We can Define a structure called person which is made up of a string for the name and an integer for the age and a double for salary.
- struct person

```
{
  char name[20];
  int age;
  double salary;
};
```


Defining Variable of A structure

- Previously we defined a data type called person.
- However, we must create a variable of that type to be able to use it.
- Following is sample variable declarations:

```
#include<iostream.h>
struct person{ char name[20];  int age;  double salary;}
void main() {
 person p1;          // variable of type person
 person *ptr;        // pointer of type person
 person PA[100]; // Array of persons }
```

Accessing member variables

- After defining a variable of type structure, we can access structure member variables using the '.' (call it dot) operator.
- In the previous example to access member fields of the structure we place a dot between the structure variable name (p1) and the name of a member variable (name, age or salary).

  p1.age , p1.salary, p1.name
- If the variable is a structure we use and arrow ->

  to separate pointer name and field name:

  ptr->age, ptr->salary, ptr->name;

Structure Initialization

- Like other data type we can initialize structure when we declare it. As far initialization goes structures obey the same set of rules as arrays. We initialize the fields of a structure following structure declaration with a list containing values for each filed.
- Assume we have the following structure definition:

  struct Employee{ int emp_id; char name[25]; char department[10]; float salary; };

Structure Initialization 2

We can initialize a variable of type employee when we declare it as follows

- Employee emp1 ={125,"basil","marketing",500.00};
- This initializes the emp_id field to 125, the name field to "basil", the department field to "marketing" and the salary filed to 500.0.
- We can use assignment statement to initialize one structure to another. Example:

  Employee emp2 = emp1;

Structure Initialization 3

- We can initialize a variable of type employee one field at a time using the assignment operator.

  Employee emp3.

  emp3.salary = 2470.28;

  strcpy(emp3.name, "Ahmad");

  strcpy(emp3.department,"sales");

  emp3.emp_id = 27;
- We can also use cin statement to read fields of a structure: cin >> emp3.name; cin >> emp3.emp_id;

  cin >> emp3.salary >> emp3.department;

Pointer to structures

- When declaring a pointer to structure, before we use the pointer to access member variables the pointer must be made to point to an existing structure or new structure. Example:

  Employee emp4 ={133,"nael","sales",1234.5};

  Employee *pt1, *pt2, *p3;

  pt1 = &emp4; // pt points to existing structure;

  pt2 = new Employee;

  pt3 = pt1;

Pointer to structures 2

- Then to access member variable use the "->" to separate pointer name and field name. Example

  strcpy(p1->name,"Nader");

  p1->emp_id = 1234;

  strcpy(p1->department,"Human Resources");

  p1->salary = 1765.5;

// Note: Employee *p5; p5->emp_id = 1234.6;

is invalid because p5 does not point to a an variable of type structure or to new structure.

Array of structures

- It is possible to define an array of structures. for example if we are maintaining information of all the students in some university. We need to use an array of structures to maintain information about all students.

```
struct info
{
    int id_no;              char name[20];
    char address[20];       int age;
};
```

Then, we can define an array of structure information as follows: info student[100];

Array of Structures 2

- Then can access member fields as follows:

  student[0].id_no = 212;

  strcpy(student[4].name,"walid");

  int x; cin >> x; cout << student[x].age;

- We can also declare and array of pointers to structure as follows: info *stptr[200]; Then to we can access member variables as follows:

  stptrp[10] = new structure;

  stptr[10]->age=19; strcpy(stptr[10]->name,"ab");

Example:

```
#include<iostream.h >
struct info
{   int id_no;   char name[20];  char address[20];    int age;  }
void main() {
     info std[100];   int I,n;
     cout << "Enter the number of students";      cin>> n;
     cout<<  "Enter Id_no, name, address,  and age");
      for(I=0;I < n;I++) {
      cin>> std[I].id_no >> std[I].name;
       cin>> std[I].address >> std[I].age;  }
      cout <<  "Student information";
     for (I=0;I< n;I++) {
         cout << std[I].id_no << "    "  << std[I].name << "  ";
          cout <<  std[I].address;  << "  " << std[I].age <<    endl;
}
```


Nested Structure

- A structure may be defined as a member of another structure. In such structures the declaration of the embedded structure must appear before the declarations of other structures. Example

```
struct date {   int day;     int month;     int year; };
struct info
{
   int id_no;                   char name[20];
   char address[20];       int age;
   date dob;
};
```

the structure student constrains another structure date as its one of its members.

Accessing member variable of Nested structure

- Given: info st1; info *ptr; Then

```
strcpy(st1.name,"Adel");
st1.id_no = 223344; ……..
st1.dob.day = 12;  st1.dob.month = 9;
st1.year = 1998;
ptr = &st1;   // make sure ptr point to structure
cout << ptr->name;   cout << pt->id_no;
cout << ptr->dob.day  << ptr->dob.month;
```


Pointer inside a structure

- A structure may contain a pointer to a variable to another structure. Programmers must be carful to allocate memory to these pointers before accessing them. In this case we must allocate memory to each structure then use the -> to access each member. Example:

```
struct data { int age; char *name; };  data ex; Then:
ex.age = 21;       ex.name = new char[30];
strcpy(ex.name,"abc");
data * ptr;  ptr = new data; ptr->name = new char[20];
ptr->age = 22;  strcpy(ptr->name,"abc');
```

Examples

- Given the following definition:
- struct info

```
{
      int id_no;
      char name[20];
      char address[20];
      int age;
};
```


Example 1

- Write code to declare a variable of type info call it s1 and give it the following values 123 for id , “wael” for name , “jenin” for address and 21 for age.
- Solution1: info s1;

```
s1.id_no = 123;     strcpy(s1.name,"wael");
s1.age = 21; strcpy(s1.address,"jenin");
```

- Solution 2:

```
info s1 = {123,"wael","jenin",21};
```


Example 2

- Define another variable of type info call it s2 and read information of s2 from the keyboard.
- Solution: info s2;

```
cin >> s2.id_no;   cin >> s2.name;
cin >> s2.address; cin >> s2.age;
```

- Print the content of s2 to screen.

```
cout << s2.id_no;    cout << s2.name;
cout << s2.address;  cout << s2.age;
```


Example 3

- Define a pointer to structure and make it point to s1. Then use the pointer to print s2.

```
info *ptr = &s2;
cout << ptr->id_no;    cout << ptr->name;
cout << ptr->address;  cout << ptr->age;
```

- Define a pointer to info and call it ptr; allocate new memoy to ptr and read info from the K.B.

```
info *ptr = new info;
 cin >> ptr->.id_no;   cin >> ptr->.name;
cin >> ptr->address; cin >> ptr->.age;
```

Example 4

- Declares an array called A of info of size 1000 then read the structure content from the K.B.

  info A[1000]; int n; cin >> n;

  for (int i = 0; I < n; i++) {

  cin >> A[i].id_no; cin >> A[i].name;

  cin >> A[i].address; cin >> A[i].age; }

- Suppose we have info *ptr = A; then

  for (int i = 0; I < n; i++, ptr++) {

  cout << ptr->id_no; cout << ptr->name;

  cout << ptr->address; cout << ptr->age;}

Passing structure to functions

- We can pass structures as arguments to a functions, and retrun structures from functions.
- A structure may be passed into a function as pass by value, reference and by address.
- You can also return a structure from a function or apointer to dynamically allocated structure in the function.
- A program example is to display the contents of a structure passing the individual elements to a function is shown next.

- # include < iostream.h >
- struct Employee{ int emp_id; char name[25]; char department[10]; float salary; };
- Employee readEmp() { Employee emp1;

  cin >> emp1.name; cin >> emp1. emp_id;

  cin >> emp1.department; cin >> emp1.salary;

  return(emp1); }

- void printEmp(Employee e)

  { cout << e.name; cout << e.emp_id;

  cout << e.department; cout << e.salary; }

- void main() {Employee em1; em1 = reademp(); printEmp(); }

Example

- Write a function that takes and array of **info** as parameter and array size; in your function return a stucture of the oldest students.

```
info largSt(info A[], int size)
{
      info st;  st = A[0];
      for (int i =1; i <  size; i++)
            if (st.age < A[i].age) st =  A[i];
      return(st);
}
```

Other Examples

- Given an array of structure info call it A, size 100

  Write code to do the following

  (1) Print the the count of students that start with the letter 'A' or 'a', Start and end with same letter.

  the count of names larger that 10 characters.

  print the name of students that are from Jenin.

  (2) sort the array A in according the name in ascending order. Then compute total salaries.

  (3) Sort the array A in descending order according to salary. Print employee name of the largest salary.

p1

```
int counta =0, countsl =0, countg10 =0, jen =0;
for (int i = 0; i < 100; i++)
{
   if (A[i].name[0] == 'A' && A[i].name[0] == 'a' )
     counta++;
  if(A[i].name[0] == name[strlen(A[i].name)-1]))
    countsl++;
  if(strlen(A[i].name) > 10)  countg10++;
  if (strcmp(A[i].name,"Jenin") jen++;
}
```


p2

```
Info temp;
for (int i =0; i < 99; i++)
      for (int j = i+1; j < 100; j++)
         if(strcmp(A[i].name,A[j].name) > 0){
              temp = A[i];
             A[i] = A[j];
             A[j] = temp;
         }
// compute total salaries.
int sum =0;   for (int I =0; I <100;i++) sum+= A[i].salary;
```


p3

```
info temp;
for (int i =0; i < 99; i++)
      for (int j = i+1; j < 100; j++)
         if(A[i].salary < A[j].salary)
         {
              temp = A[i];
             A[i] = A[j];
             A[j] = temp;
         }
cout << "the highest salary is " << A[i].name <<
"with salary" <<  A[i].salary.
```

Computer Programming

An-Najah N. University
Computer Engineering Department
Luai Malhis, Ph.D,

Input/Output files with C++

File IO

- Important points:
  - arrays provide capability for storing related values in a single entity
  - can access individual values using an index
  - can traverse through the indices, systematically access all values in the array
  - can pass the entire array as a single parameter
  - for efficiency reasons, arrays are treated as pointers (references) to memory

File IO Continue

- Arrays make it possible to store and access large amounts of data
  - requiring the user to enter lots of data by hand is tedious.
  - each execution of the program requires re-entering the data
  - *better solution:* store the data in a separate file, read directly from the file.
  - Requires only one entry by the user, can re-read as many times as desired

File Input

to read from a file, must declare an *input file stream*

- similar to the standard input stream, only input comes from a file
- in particular, can read values using >>
- defined in the <fstream> library
- Steps to read data from file shown next slide.

. . .

Reading data from file

```
#include <fstream>    // loads definition of the input
            file stream class (ifstream)
ifstream myin;          // declares input file stream
myin.open("nums.dat");   // opens the input  file
    stream using the file "nums.dat"
int numbers[100];
// reads numbers from the input file stream
//  and stores them in the array
for (int i = 0; i < 100; i++) {
myin >> numbers[i];//
}
myin.close();    // closes the input file stream
```

Averaging Grades

suppose we wanted to store grades in a file to compute their average, don't need to store the grades in an array – just read and process

```
OPEN INPUT FILE STREAM;
Set  COUNTER, SUM to zero.

READ FIRST GRADE;
while (there are more grdaes){
  ADD GRADE TO SUM;
  INCREMENT GRADE COUNTER;
  READ NEXT GRADE;
}

COMPUTE & DISPLAY AVERAGE
```

Averaging Grades 2

```
#include <iostream.h>
#include <fstream.h>

int main()
{
   ifstream myin;
   myin.open("grades.dat");

   int numGrades = 0, gradeSum = 0;

   int grade;
   // continue next slide
```

Averaging grades 3

```
myin >> grade;
 while (grade != -1) {
        gradeSum += grade;
       numGrades++;
     myin >> grade;
  }
  if (numGrades > 0) {
     double avg = (double)gradeSum/numGrades;
     cout << "Your average is " << avg << endl;
  }
  else {cout << "There are no grades!" << endl;}
 myin.close(); }
```

Grade display

```
#include <iostream.h>
#include <fstream.h>
const int MAX_GRADES = 100;
void ReadGrades(int grades[], int & numGrades);
int CountAbove(int grades[],int numGrds,int cutoff);
void main()
{
  int grades[MAX_GRADES], numGrades;
  ReadGrades(grades, numGrades);
  int cutoff;
  cout << "Enter the desired grade cutoff: ";
  cin >> cutoff;
  cout << "There are "
       << CountAbove(grades, numGrades, cutoff)
    << " grades above " << cutoff << endl;
}
```

Grade Display 2

```
void ReadGrades(int grades[], int & numGrades){
// Results: reads grades and stores in array
   ifstream myin;
    myin.open("grades.dat");
    numGrades = 0;
    int grade;     myin >> grade;
    while (grade != -1){
        grades[numGrades] = grade;
        numGrades++;
        myin >> grade;
   }
     myin.close();
}
```

Grade Display 3

```
int CountAbove(int grades[],
               int numGrds, int cutoff)
// Assumes: grades contains numGrades grades
// Returns: number of grades >= cutoff
{
    int numAbove = 0;
    for(int i = 0; i < numGrds; i++) {
        if (grades[i] >= cutoff) {
            numAbove++;
        }
    }
    return numAbove;
}
```

Improvement 1: end-of-file

- Each file has a special marker called EOF that marks the end of input

input via >> evaluates to a Boolean value

- if the input succeeds (the expected type of input is read in), the expression evaluates to true
- if the input fails (the wrong type of input is read in or end-of-file has been reached), the expression evaluates to false

thus, can have a while loop driven by an input statement

Read until end of file

```
void ReadGrades(int grades[],int&
  numGrades){
// Results: reads grades and stores in array
    ifstream myin;
    myin.open("grades.dat");
    numGrades = 0;
    int grade;
    while (myin >> grade){
       grades[numGrades] = grade;
       numGrades++;}
    myin.close();
}
```

Generalizing file names

we can generalize the program to read from an arbitrary file

- can prompt the user for the file name, read into a string

```
void ReadGrades(int grades[], int & numGrades)
// Results: reads grades and stores in array
//            numGrades is set to the # of
  grades
{   char  filename[80];
    cout << "Enter the grades file name: ";
    cin >> filename;
    ifstream myin;   myin.open( filename );
    numGrades = 0;  int grade;
  while (numGrades < MAX_GRADES && myin >>
  grade) {  grades[numGrades] = grade;
  numGrades++; }
  myin.close();
}
```

Guarding against file errors

we can test the ifstream to be sure that the specified file was opened correctly

- the ifstream has a Boolean value associated with it
- if the file exists (and not past end-of-file), the ifstream evaluates to true
- otherwise, the ifstream evaluates to false
- must clear the input file stream between attempts to open (in order to reset true/false value)

```
void ReadGrades(int grades[], int & numGrades)
// Results: reads grades and stores in array
//          numGrades is set to the # of grades
{  char filename[50];
    cout << "Enter the grades file name: ";
    cin >> filename;
    ifstream myin;  myin.open( filename );
    while (!myin) {
      cout << "File not found.  Try again: ";
      cin >> filename;
      myin.clear(); myin.open( filename ); }
  numGrades = 0;     int grade;
  while (numGrades < MAX_GRADES && myin >> grade)
  {   grades[numGrades] = grade;   numGrades++; }
  myin.close();
}
```

File Output

It is possible to direct program output to a file

- must declare an output file stream (ofstream)
- open using a file name (same as with ifstreams)
- write using <<
- close when done (same as with ifstreams)

File Output

```
#include <fstream>
// loads definition of the outputfile
  stream class(ofstream)
 ofstream myout;
//declares output file stream
myout.open("nums.out");
//opens the output file stream using the
  file"nums.out"
for (int i = 0; i < 100; i++) {
// writes numbers to output file,
    myout << i << endl;
}
myout.close();
// closes the output file stream
```

Updating the grade file

```
#include <iostream.h>
#include <fstream.h>
const int MAX_GRADES = 100;
void ReadGrades(int grades[], int &numGrades, char fname[]);
void ScaleUp(int grades[], int numGrades);
void WriteGrades(int grades[],int numGrades,char fname[50]);

int main()
{
  int grades[MAX_GRADES], numGrades;
  char fileName[50];
  ReadGrades(grades, numGrades,fileName);
  ScaleUp(grades, numGrades);
  WriteGrades(grades, numGrades, fileName);
  return 0;
}
```

```
void ReadGrades(int grades[],    int & numGrades, char fname[50]) {
// Results: reads grades and stores in array
//  fname is the name of the input file.
   cout << "Enter the grades file name: ";              cin >> fname;
   ifstream myin;              myin.open( fname );
    while (!myin) {
        cout << "File not found.  Try again: ";       myin.clear();
        cin >> fname;                          myin.open( fname )
    }
    numGrades = 0;        int grade;
   while (numGrades < MAX_GRADES && myin >> grade)
    {  grades[numGrades] = grade;         numGrades++;      }
   myin.close();
 }
// next slide the code for  ScaleUp and  WriteGrade
```

Updating the grade file Continue

```
void ScaleUp(int grades[], int numGrades) {
// Assumes: grades contains numGrades grades
// Results: a bonus is added to each grade
    int bonus;
    cout << "How many bonus points are there? ";      cin >> bonus;
    for(int i = 0; i < numGrades; i++)
    {     grades[i] += bonus;if (grades[i] > 100)       grades[i] = 100;    }
}

void WriteGrades(int grades[],     int numGrades, char fname[50])  {
    ofstream ofstr;           ofstr.open(fname);
    for (int i = 0; i < numGrades; i++) {    ofstr << grades[i] << endl;    }
     ofstr.close();
}
```

Copy a file to another file

would the following code suffice to copy the contents of a file?

i.e., does copy.txt look exactly like the input file? recall that `>>` ignores all whitespace

- thus, the previous program will copy all non-whitespace chars, but spacing will be lost

the `get` member function (from `<iostream>`) reads a character, including whitespace

- applied to an input stream
- one argument: a char

Copy Example 1

```
#include <iostream.h>  #include <fstream.h>
#include <string.h>
void main()
{
  char infile[40];
  cout << "Enter the input file name: ";
  cin >> infile;
  ifstream ifstr;    ifstr.open(infile);
  ofstream ofstr;     ofstr.open("copy.txt");
  char ch;
  while (ifstr.get(ch)) {    ofstr << ch;    }
  ifstr.close();   ofstr.close();
}
```

Faster Copy

the `getline` function (from `<iostream>`) reads an entire line of text, including whitespace two arguments: input stream and a string

```
#include <iostream.h>   #include <fstream.h>
#include <string.h>
void  main()
{  char infile[40];
  cout << "Enter the input file name: ";    cin >>
  infile;
  ifstream ifstr;      ifstr.open(infile);
  ofstream ofstr;       ofstr.open("copy.txt");
  char line[255];
  while (getline(ifstr, line)) {   ofstr << line <<
  endl; }

  ifstr.close();   ofstr.close();
}
```