Chapter 00: Review of Algorithm Development

Welcome to a study of C++ Programming Fundamental! Mastery of the material covered in the course is useful towards successful completion of your degree in computer science (CS) as well as a career in the discipline; most CS professionals are comfortable C++ programmers. This chapter is intended to be an easy read, so if you are already familiar with the material (as you should be), just gloss over to the next section. The chapter proceeds via the following captions:

- Overview of Computer Hardware
- Overview of Computer Software
- Rudiments of Algorithm Development
- Rudiments of Program Development
- Summary and Concluding Remarks

0.1 Overview of Computer Hardware

In this and several other courses that you will pursue, you will be giving instructions to the computer. In order to gain mastery in this, knowledge of the internal workings of the machine is useful. With this in mind, this section covers the following:

- Brief History of Computer Technology
- The Architecture of a Contemporary Computer
- Introduction to the Binary System
- Introduction to the Octal System
- Introduction to the Hexadecimal System
- Character Representation in the Computer
- Representing Negative Numbers
- Representing Small and Large Numbers

0.1.1 Brief History of Computer Technology

In order to appreciate the development of computer technology, a brief history is necessary:

- 1834: Charles Babbage designed an analytic machine with the following components:
    - Store – a memory unit consisting of counter wheels
    - Mill – an arithmetic unit capable of performing addition, subtraction, multiplication, and division
    - Operation Cards Feeder
    - Variable Cards Feeder
    - Output – a punch card device
- 1936: Zuse introduced the concept of binary numbers
- 1939: ABC computer was developed by John Atanasof & Clifford Berry
- 1946: ENIAC (Electronic Numerical Integrator & Computer), the first general purpose electronic digital computer was completed by John Mauchly and John Eckert. It was based on the decimal system.
- 1952: Von Neuman and his colleagues completed the IAS (Institute of Advanced Studies in Princeton) computer. This computer is the prototype of a subsequent general purpose computers, hence the term Von Neuman Machine.
- Since the Von Neuman model, we have had six generations of computers as summarized in figure 0-1.

Figure 0-1: Computer Hardware Generations

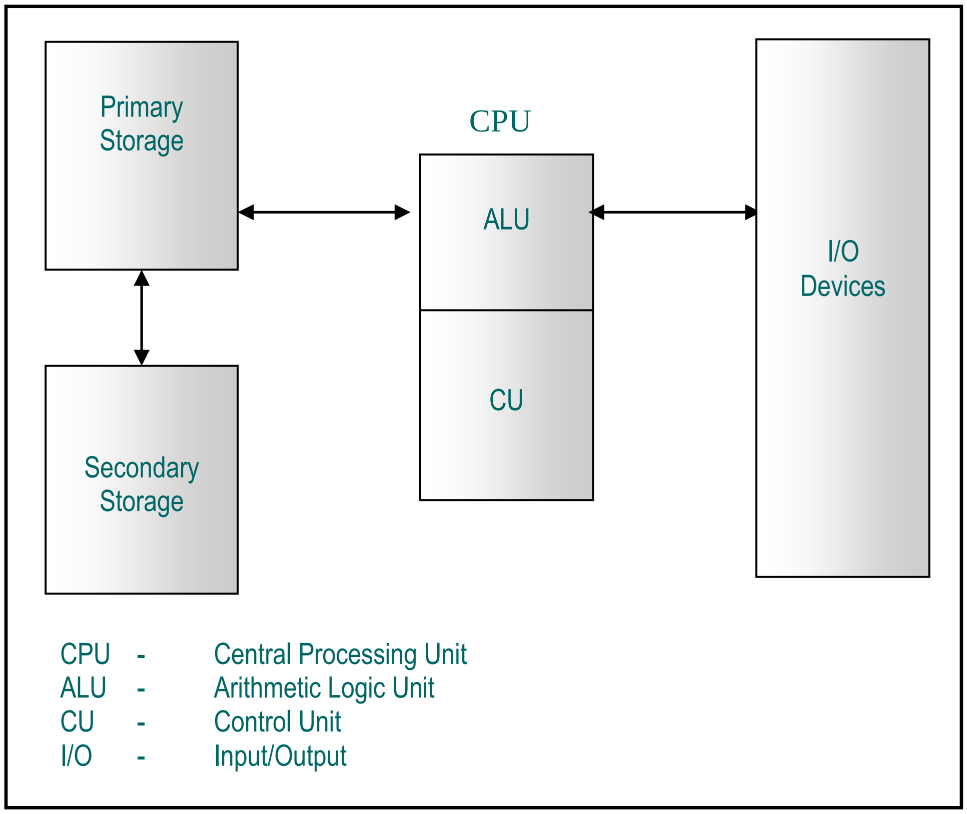
Generation	Approx. Date	Technology	Speed (OPS)
1	1946-1957	Vacuum Tube	40,000
2	1958-1964	Transistor	200,000
3	1965-1971	Small Scale Integration (SSI)	1M
4	1972-1977	Large Scale Integration (LSI)	10M
5	1978-1993	Very Large Scale Integration (VLSI)	100M
6	1993-present	Extra Large Scale Integration (XLSI)	above 100M

0.1.2 The Architecture of a Contemporary Computer System

The basic Von-Neumann structure still lives on with some enhancements. The basic components of a computer system are **Primary Storage**, **Secondary Storage**, **Central Processing Unit** (CPU) and **Input/Output Module** (see figure 0-2). The components are linked by data buses (and controlled from the CPU).

Following is a summary of the role of each component in the computer system. You will learn much more about these components in your computer organization course, but for now, an overview will suffice.

Figure 0-2: Basic Architecture of a Computer System



0.1.2 The Architecture of a Contemporary Computer System (continued)

Primary Storage:

- This unit is also called the main memory, or core memory. The latter term has some historical significance – main memory used to be effected by use of ferromagnetic loops referred to as core.
- Primary storage consists of electronic storage units (registers made up of cells), which can be accessed randomly. For this reason, it is also referred to as *random access memory* (RAM).
  All data used by the CPU in current work resides there in primary storage. The data may be volatile or nonvolatile. By volatile, we mean that data is stored electronically; therefore loss of power means loss of memory. By nonvolatile, the data is also stored electronically, but in such a way that a power loss does not result in data loss.

Secondary Storage:

- Data not required immediately by the CPU are stored in secondary storage and fetched (by the CPU) when required.
- Examples of secondary storage units are magnetic disks, magnetic drums, magnetic tapes, magnetic cassettes, magnetic cartridges, optical storage devices, solid-state drives (SSD), compact disks, microfilm and microfiche.

Input/Output Module:

- Input devices are devices that allow communication to the computer by the user. Traditionally, punch cards were used. Currently input media are in the form of keyboards, and optical character recognition (OCR) devices, mouse, voice, joysticks, etc...
- Output devices allow the user to obtain information from the computer. Output media include printers, monitor (traditionally called visual display unit — VDU), storage secondary devices, voice, etc.
- Some devices allow for both input and output, hence I/O. These include VDU and all the secondary storage units mentioned earlier.

Central Processing Unit:

- The CPU is the heart and head of the computer. It governs the operation of the entire system.
- The CPU manages the execution of all commands via its *control unit* (CU). These include interrupts, subroutine calls and I/O requests.
- The user communicates to the CU of the CPU via the *operating system*. The CU then issues machine commands on its behalf.
- The CPU manages all arithmetic and logical manipulations via its *arithmetic logic unit* (ALU).
- **Note:** All internal workings are done in binary and then converted to hexadecimal or decimal for the user, by the operating system.
- The CPU maintains link with all components of the system via data buses. Signal transfer is regulated by the CU.

0.1.3 Introduction to the Binary System

All data stored on a computer is represented in binary form. The operating system is responsible for converting data from binary to decimal and vice versa. The operating system works in concert with compilers and translators to convert source code to machine code. This will be further explained in more advanced courses.

In assembly programming and communication protocol writing, the system programmer relates to the computer at a very low level – close to machine code. This will be further expanded in more advance courses.

Two digits make up the binary system — 0 and 1. Sequencing starts from 0 to 1. Whenever a power of 2 is reached, a new binary digit (bit) is introduced and the sequencing starts over. Thus:

Binary:	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111
Dec.:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

A simple way to appreciate this is to recognize that each bit position represents a power of 2, starting at 2^0.

Example 0-1: $100110110_2 = 310$

$100110110_2 = 0(2^0) + 1(2^1) + 1(2^2) + 0(2^3) + 1(2^4) + 1(2^5) + 0(2^6) + 0(2^7) + 1(2^8)$
$= 2 + 4 + 16 + 32 + 256 = 310$

We represent fractions by introducing a binary point. Each bit positioned to the right of the binary point represents $1/2^p$ where p represents the pth position to the right of the binary point.

The number represented as $B_n\ B_{n-1} \ldots B_1\ B_0\ .\ P_1\ P_2 \ldots P_m$ in binary, evaluates to the decimal equivalent of

$$B_n(2^n) + B_{n-1}(2^{n-1}) + \ldots + B_1(2^1) + B_0(2^0) + P_1(2^{-1}) + P_2(2^{-2}) + \ldots + P_m(2^{-m})$$

Example 0-2: $1011.0101_2 = 11.3125$

1011.0101_2 evaluates to the decimal equivalent of
$1(2^3) + 0(2^2) + 1(2^1) + 1(2^0) + 0(2^{-1}) + 1(2^{-2}) + 0(2^{-3}) + 1(2^{-4})$
$= 8 + 0 + 2 + 1 + 0 + 0.25 + 0 + 0.0625 = 11.3125$

Conversion from decimal to binary is equally simple: repeatedly divide by 2 and read the remainder digits in reverse order. [Illustrate]

Example 0-3: $78 = 1001110_2$

78/2 = 39 R 0	39/2 = 19 R 1	19/2 = 9 R 1	9/2 = 4 R 1	4/2 = 2 R0
2/2 = 1 R 0	1/2 = 0 R 1			

0.1.4 Introduction to the Octal System

In the *octal* system, the base is 8; we therefore have digits are 0, 1, 2… 7.

A number represented as: $0_n\ 0_{n-1} \ldots\ 0_2\ 0_1\ 0_0 \,.\, P_1\ P_2\ P_3 \ldots\ P_m$
evaluates to the decimal equivalent of:
$(0_n * 8^n) + (0_{n-1}\, 8^{n-1}) + \ldots\ 0_2{*}8^2 + (0_1{*}8^1) + (0_0{*}8^0) + (P_1{*}8^{-1}) + (P_2{*}8^{-2}) \ldots\ + (P_m\ 8^{-m})$

Conversion from octal to decimal is therefore quite straightforward.

Example 0-4:

$25_8 = (2 * 8^1) + (5 * 8^0) = 21$
$100_8 = (1 * 8^2) + (0 * 8^1) + (0 * 8^0) = 64$
$100.45_8 = 64 + (4 * 8^{-1}) + (5 * 8^{-2}) = 64 + 0.5 + 0.781 = 64.578$

Conversion of decimal to octal is similar to conversion from decimal to binary: Repeatedly divide by 8 and read the remainder in reverse order.

Example 0-5: $64 = 100_8$

64 /8 = 8 R 0 8/8 = 1 R0 1/8 = 0 R 1

Conversion from octal to binary is achieved by replacing each octal digit with three equivalent binary digits.

Example 0-6:

$25_8 = 010101_2$

Conversion from binary to octal is simply the reverse of the conversion from octal to binary: Starting at the most significant bit (i.e. the rightmost whole-number bit, and the leftmost fractional bit), every three bits correspond to one octal digit.

Example 0-7:

$010101_2 = 25_8$
$1001101_2 = 115_8$
$1001101.1101_2 = 115.64_8$

0.1.5 Introduction to the Hexadecimal System

In the *hexadecimal* system, the base is 16; we therefore have digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Note the introduction of hexadecimal (hex) digits A – F to represent ten, eleven, twelve, thirteen, fourteen, fifteen. This is necessary since we cannot use 10, 11, 12, 13, 14 and 15 to represent these numbers, as it would cause ambiguity.

A number represented as: $H_n\ H_{n-1} \ldots H_2\ H_1\ H_0 \mathbf{.}\ P_1\ P_2\ P_3 \ldots P_m$
evaluates to the decimal equivalent of:
$(H_n * 16^n) + (H_{n-1}\ 16^{n-1}) + \ldots H_2{*}16^2 + (H_1{*}16^1) + (H_0{*}16^0) + (P_1{*}16^{-1}) + (P_2{*}16^{-2}) \ldots + (P_m\ 16^{-m})$

Conversion from hex to decimal is therefore quite straightforward.

Example 0-8:

$100_{16} = 1{*}16^2 + (0{*}16^1) + (0{*}16^0) = 256$

$101.16A_{16} = 257 + (1{*}16^{-1}) + (6{*}16^{-2}) + (10{*}16^{-3})$
$= 257 + 0.0625 + 0.0234 + 0.0024 = 257.0883$

Conversion of decimal to hex is similar to conversion from decimal to binary: Repeatedly divide by 16 and read the remainder in reverse order.

Example 0-9: $256 = 100_{16}$

256/16 = 16 R 0   16/16 = 1 R 0   1/16 = 0 R 1

Conversion from hex to binary is achieved by replacing each hex digit with four equivalent binary digits.

Example 0-10:

$100_{16} = 000100000000_2 = 100000000_2$

AF31.F 1010111100110001.1111_2

Conversion from binary to hex is simply the reverse of conversion from hex to binary: Starting at the most significant bit (i.e. the rightmost whole-number bit, and the leftmost fractional bit), every four bits correspond to one hex digit.

Example 0-11:

$100000000_2 = 0001\ 0000\ 0000_2 = 100_{16} = 100_H$
$1111_2 = F_{16} = F_H$
$0111\ 1001_2 = 79_{16} = 79_H$
$101101.011_2 = 0010\ 1101.0110_2 = 2D.6_{16} = 2D.6_H$

0.1.6 Character Representation in the Computer

Characters, letters, and symbols are assigned specific (predefined) values, known and understood by the computer. Three systems of data representation are prevalent — EBCDIC, ASCII and Unicode.

EBCDIC is an acronym for *Extended Binary Coded Decimal Interchange Code.*

- Four bits are used for zoning.
- There is also a four-bit numeric code.
- The zone bits and numeric codes are shown in figure 0-3. Note that lower case characters are facilitated.

Figure 0-3: EBCDIC Coding System

EBCDIC Codes

Zone Codes		Numeric Codes								
1000	a - i	0000	0							
1001	j - r	0001	A	a	J	j	1			
1010	s - z	0010	B	b	K	k	S	s	2	
1100	A - I	0011	C	c	L	l	T	t	3	
1101	J - R	0100	D	d	M	m	U	u	4	
1110	S - Z	0101	E	e	N	n	V	v	5	
0000	0 - 9	0110	F	f	O	o	W	w	6	
		0111	G	g	P	p	X	x	7	
		1000	H	h	Q	q	Y	y	8	
		1001	I	i	R	r	Z	z	9	

Examples

0	00000000	a	10000001
A	11000001	b	10000010
B	11000010	c	10000011
C	11000011	i	10001001
I	11001001	j	10010001
J	11010001	k	10010010
K	11010010	r	10011001
R	11011001	s	10100010
S	11100010	t	10100011
T	11100011	3	10101001
Z	11101001		
1	00000001		
2	00000010		
9	00001001		

Special Characters

SP	01000000	SUB	00111111	‘	01101011
¦	01101010	ESC	00010111	-	01100000
NULL	00000000	!	01011010	.	00100100
SOH	00000001	“	01111111	/	01100001
STX	00000010	$	01011011	:	01111010
ETX	00000011	%	01101100	;	01011110
ENQ	00101101	&	01010000	<	01001100
FF	00001100	’	01111101	=	01111011
SYN	00110010	(	01001101	>	01101110
ETB	00100110	)	01011101	?	01101111
EM	00011001	+	01001110		

0.1.6 Character Representation in the Computer (continued)

ASCII is an acronym for *American Standard Code for Information Interchange.* The is a seven-bit coding system (ASCII-7), as well as an eight-bit coding system (ASCII-8), which simply adds a parity bit to ASCII-7.

- Numeric code starts from 0000 to 1111 in each zone.
- Each zone has 16 characters.

ASCII-7 zone bits are shown in figure 0-4

Figure 0-4: ASCII Coding System

ASCII Codes								
	B_6 - B_4 (High Order)							
B_3 – B_0	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	“	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	’	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Examples			
A	1000001	a	1100001
B	1000010	b	1100010
O	1001111	o	1101111
P	1010000	p	1110000
Q	1010001	q	1110001
R	1010010	r	1110010
S	1010011	s	1110011
Z	1011010	z	1111010
0	0110000		
1	0110001		
9	0111001		

0.1.6 Character Representation in the Computer (continued)

The main problem with the EBCDIC and ASCII systems has to do with the treatment of special characters for instance those used in oriental languages. Such characters are represented (in EBCDIC and ASCII) by combining more than one byte of code. The Unicode system addresses this problem by expanding the basic ASCII-8 code to a sixteen-bit code. In so doing, all original ASCII codes are taken care of, and there is additional bandwidth to represent oriental (and other special) characters. Java — the programming language you use in this course — uses the Unicode system. We will therefore have more to say about this coding system as the course progresses.

In all three systems, the following conventions hold for bits, bytes, kilobytes (KB), megabytes (MB), gigabytes (GB), terabytes (TB), peta-bytes (PB), and exa-bytes (XB):

- 8 bits make 1 byte; 2 bytes make 1 word
- 2^{10} bytes = 1KB
- 2^{10} KB = 2^{20} bytes = 1MB
- 2^{10} MB = 2^{30} bytes = 1GB
- 2^{10} GB = 2^{40} bytes = 1TB
- 2^{10} TB = 2^{50} bytes = 1PB
- 2^{10} PB = 2^{60} bytes = 1 XB

Modern computer systems tend to use some variation of the ASCII coding system and/or Unicode system; the EBCDIC system has been relegated to IBM systems primarily.

0.1.7 Representing Negative Numbers

The binary system, of itself, does not effectively represent negative numbers and very large or very small numbers. Further modification is therefore required: Numbers are represented via *signed magnitude*, *1's complement* or *2's complement*. Very large or very small numbers are represented as floating point numbers.

Signed Magnitude: The Signed Magnitude convention is to use the leftmost bit as the sign bit. Thus:

A number that begins with a 1 is a negative number.
A number that begins with a 0 is a positive number

In a N-bit word, the right most N-1 bits hold the magnitude and the leftmost bit holds the sign.

Example 0-12:

0010010_2 = +18 {Using 8-bit word}
10010010_2 = -18

In an 8-bit word, the smallest number that can be represented is 11111111_2 i.e. is –127. The largest number that can be represented is 0111111_2 i.e. +127. The range of numbers that can be represented is –127 … 127. Generally, for **n** bits, the range is $-(2^{n-1} - 1) \ldots (2^{n-1} - 1)$.

0.1.7 Representing Negative Numbers (continued)

There are two drawbacks to the Signed Magnitude approach:

- There are two representations of zero (10000000 and 00000000). This is undesirable as it makes it difficult to test for zero.
- Addition and subtraction require consideration of the sign bit and the relative magnitudes of the numbers in order to effect the operation.

1's Compliment: The 1's Complement operation on a set of binary digits is obtained by simply flipping the bits: replace each 0 by 1 and each 1 by 0.

Example 0-13:

Let	X = 01010001
Then	1's Complement of X = 10101110
Let	Y = 10101110
Then	1's Complement of Y = 01010001

The convention for the 1's Complement representation is as follows:

- Positive numbers are represented as signed magnitude (no change required).
- Negative numbers are represented by 1's complement of the positive integer with the same magnitude.

Example 0-14:

18 = 00010010_2
-18 = 1's complement of 18 = 11101101

Four observations about the 1's Compliment are worth noting here:

1. Leftmost bit still operates as sign bit.
2. N = 1's Complement of –N, where N is any binary number.
3. For an 8-bit word, number representation is in the range 01111111_2 to 10000000_2 (.i.e. is 127 to –127). Generally for an N-bit word, the number representation is in the same range as it would be for Signed Magnitude: $-(2^{n-1} - 1) \ldots (2^{n-1} - 1)$.
4. Integer arithmetic is easily facilitated.

The approach has one serious drawback: there are two representations of zero (11111111 and 00000000) – as in Signed Magnitude.

2's Compliment: The 2's Complement operation on a set of binary digits is obtained obtaining the 1's Compliment (flipping the bits), and adding 1.

The convention for the 2's Complement representation is as follows:

- Positive numbers are represented as signed magnitude (no change required).
- Negative numbers are represented by 2's complement of the positive integer with the same magnitude.

0.1.7 Representing Negative Numbers (continued)

Example 0-15:

```
18  =  00010010
-18  = 00010010
       11101101     1's complement of the positive integer
             +1

       11101110         2's complement of 18
```

Five observations about the 2's Compliment are worth noting here:

1. N = 2's Complement of –N, where N is any binary number.
2. The leftmost bit continues to function as the sign bit.
3. For an 8-bit word, number representation is in the range 10000000_2 to 01111111_2 (i.e. -128 to 127). [Observe: 10000000 is 1's complement 01111111 + 1 i.e. 10000000] Generally for an N-bit word, the range is $–(2^{n-1})$ to $(2^{n-1} – 1)$ and is therefore wider than that of 1's Complement or Signed Magnitude representation.
4. There is only one representation of zero.
5. Integer arithmetic is easily facilitated.

These observations make 2's Compliment more desirable and widely used than Signed Magnitude or 1' Compliment. Figure 0-5 shows a comparison among the three approaches, based on a 4-bit word.

Figure 0-5: Comparison – Signed Magnitude, 1's Compliment & 2's Compliment

Decimal	Signed Magnitude	1's Complement	2's Complement
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	0000
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	------	------	1000

0.1.8 Representing Very Small and Very Large Numbers

As you are aware, in science, we sometimes want to represent minute or huge numbers for which our conventional system of number representation is unsuited. For these numbers, we resort to the Scientific Notation.

Example 0-16:

18,000,000,000 = $1.8 * 10^{10}$
In CS, we represent this as 1.8E10.

In Computer Science (CS), we use the term *floating point* to mean a convenient representation of the Scientific Notation. There are typically two types of floating point numbers:

- *Single precision floating point* numbers require no additional digits (bits) to be represented. They are frequently loosely referred to as *floating point* numbers.
- *Double precision floating point* numbers are extremely small or large, and therefore require additional digits (bits) to be represented.

A full discussion of floating point numbers is beyond the scope of this course. However, a cursory introduction is warranted, and is therefore provided here:

Generally, a number can be represented as $M * R^E$ where M is equal to the *mantissa* (or significand), R is the *radix* (i.e. base), and E is the *exponent*. M and E may be positive or negative. In floating point representation, R = 2. Any number can therefore be stored in a binary word with three fields namely Sign, M, and E. The base (radix) is implied and is not usually specified.

Given the above, a 32-bit floating point format may be represented as follows:

Figure 0-6: 32-bit Floating Point Representation

B_{31} B_{30} … B_{23} B_{22} … B_0
where

B_{31}	represents the sign bit
B_{30} … B_{23}	represents the *biased exponent*
B_{22} … B_0	represents the *normalized mantissa*

Biased Exponent: A fixed value must be subtracted from this field to get the true exponent value. Put another way, a fixed value is added to the exponent before it is stored. Generally, if **n** bits represent the exponent, the bias is $\mathbf{2^{n-1}}$. In our 32-bit representation, the bias field contains 8 bits; therefore the bias is 2^7 i.e. 128.

0.1.8 Representing Very Small and Very Large Numbers (continued)

Mantissa: The mantissa must be normalized i.e., of the form 0.1bbb… where each b represents a binary digit. Thus, a normalized binary number is of the form ± 0.1bb … * 2 ^{E} where E can be positive or negative. This implies that the leftmost bit of the mantissa is always 1 and can therefore be implied rather than stored. The 23-bit field is therefore used to store a 24-bit mantissa with a value between 0.5 and 1.0.

Example 0-17: Figure 0-7 provides some examples of floating point representations using the 32-bit convention of figure 0-6.

Figure 0-7: Examples of Floating Pont Representations

a. 0.11010001 x 2^{10100} :

Sign = 0
Exponent + Bias = 10100 + 10000000 = 10010100
Mantissa stored is 10100010000000000000000
Thus 0.11010001 x 2^{10100} is stored as 010010100 10100010000000000000000

b. – 0.11010001 x 2^{10100} :

Sign = 1
Exponent + bias = 10010100
Mantissa stored is as above.
Thus, the number is stored as 110010100 10100010000000000000000

c. 0.11010001 x 2^{-10100} :

Sign bit is 0

0 1 1 1 1
Exponent + bias = ~~1 0 0 0 0~~^{1}0 0 0
- 1 0 1 0 0
0 1 1 0 1 1 0 0

Mantissa is stores as 10100010000000000000000
Thus, 0.11010001 x 2^{-10100} is stored as 001101100 10100010000000000000000

d. – 0.11010001 x 2^{-10100} is stores as 101101100 10100010000000000000000

e. 1010110 = 0.1010110 x 2^{7}

Sign bit is 0
Exponent + bias = 10000000 + 111 = 10000111
Mantissa is stored as 01011000000000000000000
Thus 1010110 is stored as 01000011101011000000000000000000

0.1.8 Representing Very Small and Very Large Numbers (continued)

By analyzing the floating point notation, a number of general observations can be made, and are worth noting:

1. Floating point may be implemented using 1's Compliment or 2's Compliment, but 2'Compliment is preferred.

2. Floating point representation allows for a wide range of number to be represented as is required in the computer.

Exponent Range is [-2^{n-1} to $2^{n-1} – 1$] when **n** = the binary width of the exponent. As an example, for an 8-bit exponent, the range is -128 to 127.

Mantissa Range: If **n** = binary width of the mantissa, then the mantissa range is:
$-(1-2^{-(n + 1)})$ to – 0.5 for negative numbers and
0.5 to $(1-2^{-(n+1)})$ for positive numbers.

Example: when n = 23 as in the 32-bit representation we have been looking at a range of: [- $(1-2^{-24})$ to 0.5] for negative mantissas and
[0.5 to $(1 – 2^{-24})$] for positive mantissas.

Range is therefore: – $(1-2^{-24}) \times 2^{127}$ to – 0.5×2^{-128} for negative numbers and 0.5×2^{-128} to $(1-2^{-24})$ $*2^{127}$ for positive numbers.

3. Regions on the number line not included (assuming 32-bit as above) are:

- Negative numbers less than – $(1-2^{-24}) \times 2^{127}$ called negative overflow.
- Negative numbers greater than $–0.5 \times 2^{-128}$ called negative underflow.
- Zero
- Positive numbers less than 0.5×2^{-128} called positive underflow.
- Positive numbers greater than $(1 – 2^{-24}) \times 2^{127}$ called positive overflow.

4. Overflow and underflow can be eliminated by increasing the precision of the floating point system, hence the term *double precision*. Increasing the precision simply means adding more bits so that smaller and larger number can be accurately represented.

0.2 Computer Software

In this course, you will be learning to write computer software using the C++ programming language. As you progress to more advanced courses, you will learn to write software at more advanced levels. However, you must start here. This section of the course takes you through the salient points that you need to understand about computer software in general. The discussion will include:

- Basic Software Concepts
- Categories of Software
- Software Development Life Cycle
- Software Quality
- Computer Aided Software Engineering

0.2.1 Basic Software Concepts

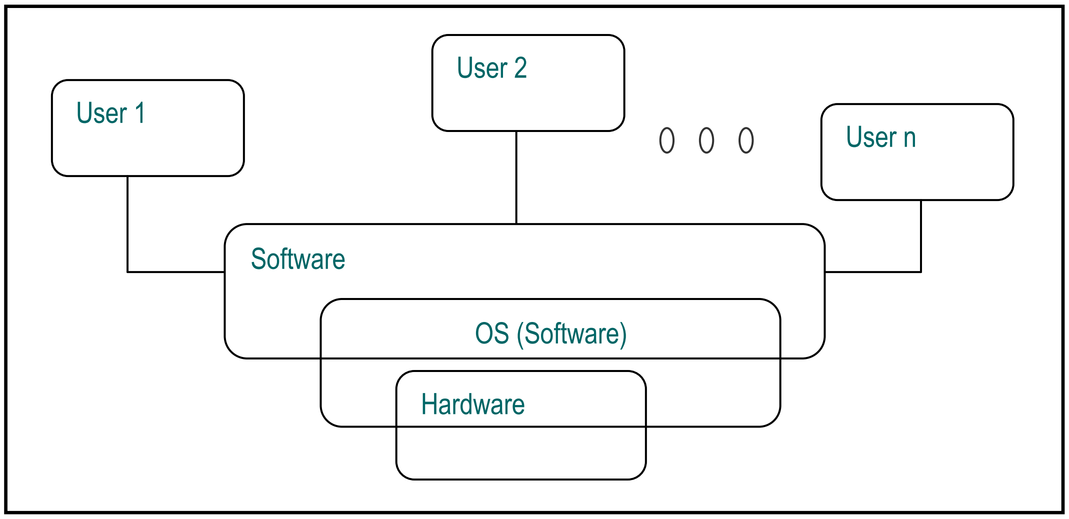
In order to make the computer system useful to human beings, we must create a user interface. Critical objectives of user interface are:

- To shield the user from the gory details of the underlying hardware.
- To present information in a manner that is readily understandable, and does not require in depth knowledge of the internal structure of the system.
- To allow easy user access to the resources of the computer system.
- Prevention of accidental or intentional damage to the hardware, programs and data.
- To facilitate communication between user and hardware. We refer to this communication as human-computer interaction (HCI).

Computer software may be defined as instructions to the (hardware) components of a computer system, so that HCI is facilitated or other specific problems are solved. Figure 0-8 illustrates the role that software plays in the HCI dilemma. Two critical points to be noted about software are:

- It is through software that we create *virtual environments* for end users.
- End users communicate to computer software via its user interface.

Figure 0-8: Role of Computer Software in Facilitating HCI



0.2.1 Basic Software Concepts (continued)

Computer software has been through four (arguably five) generations:

- First generation — machine code
- Second generation — assembly languages
- Third generation — high level languages (HLL)
- Fourth generation — languages which are more powerful and easier to use than HLLs
- Arguably, the fifth generation is one of more intelligent software.

0.2.2 Categories of Software

Software engineering addresses the problem of software planning, development, and management. It is a very wide, variegated field, constrained only by one's own imagination. There are, however, some observable categories of software. Figure 0-9 provides a list of prevalent software categories.

Figure 0-9: Common Software Categories

Operating System: A set of programs that provide certain desirable and necessary features for users of a computer system.
Compiler: A program that allows users (programmers) to code instructions to a computer system in a high level language (HLL). The compiler converts the instructions from source code to object (machine) code.
Interpreter: An interpreter is similar to a compiler. However, it operates in an interactive mode, whereas the compiler operates in batch mode.
Assembler: A special compiler that works on lower level (assembly language) programs, converting them to object code.
Database Management System (DBMS): A set of programs that facilitate the creation and management of a database. A database is a collection of related records. A database consists of several related files containing data.
Network Protocol: A software system that facilitates electronic communication on a computer network, according to a prescribed set of rules and standards.
Desktop Applications: Describe all generic computer software applications that run on microcomputers and notebook computers. They include subcategories such as word games, multimedia applications, and web browsers.
Information System: A software system that facilitates the management of information. There are different kinds of information systems; these include batch processing systems, transaction processing systems, management information systems (MIS), decision support system (DSS), execute information systems (EIS), strategic information systems (SIS), expert systems (ES), hypermedia (documentation) systems, Web information system (WIS).
Data Warehouse: An integrated, subject-oriented, time-variant, nonvolatile, consistent database, constructed from multiple source databases, and made available (in the form of read-only access) to support decision making in a business context.
Business Applications: Describe software applications that solve specific problems in a business. They include, but are not confined to desktop applications and some information systems. Business applications therefore include accounting packages, library management systems, manufacturing systems, desktop applications, college/University administration systems, inventory management systems, point of sale systems, airline reservation systems.
Artificial Intelligence (AI) Systems: A system that causes the computer to exhibit humanlike intelligence. Popular branches include neural networks, natural language processing and expert systems. **Expert System (ES):** A special case AI system that emulates a human expert in a particular problem domain, e.g. medical diagnosis and robotics.
Hypermedia System: A special desktop application that facilitates the creation and maintenance of multi-media-based systems. This includes geographic information systems (GIS), documentaries, documentation systems, etc.
Computer Aided Design (CAD) System: Special business/desktop application used in manufacturing and architecture to design blueprints.
Computer Aided Manufacturing (CAM) System: Used in manufacturing environments.
Computer Integrated Manufacturing (CIM) System: A combination of CAD and CAM.
Computer Aided Software Engineering (CASE) Tool: A sophisticated software product that is used to automate design and construction of other software products.
Rapid Application Development (RAD) Tool: A brand of CASE tool that facilitates the rapid design and construction of other software applications.
Software Development Kit (SDK): A conglomeration of software products bundled together for the purpose of software development.

0.2.3 Software Development Life Cycle

Software passes through a number of phases during their useful life. The software development life cycle (SDLC) describes these phases:

- Investigation
- Requirements Analysis
- Design/Modeling
- Development/Construction
- Implementation
- Maintenance and Management

Each phase involves a number of stages of activities that will be further discussed in more advanced courses. Suffice to emphasize that:

- Before you write software, planning is of paramount importance.
- Ease of development is a function of software planning (design). Good design leads to easy development; poor design leads to difficult, time consuming development.
- After implementation, software maintenance ensures that the relevance of the product is protected.

0.2.4 Software Quality

Software quality is a function of software design and software construction. Zero error tolerance is the ultimate achievement, to be attained. Software quality will be further discussed in more advanced courses. However, it must be stressed here that when a programmer writes a program, he/she must thoroughly test it to ensure that it performs to requirement.

Programming languages and other software development tools provide facilities such as trace and debug to assist the programmer in writing error-free code. Additionally, a structured walk-through must be conducted.

Here are three tips in writing quality software:

- Never be satisfied with a program until it performs according to requirement.
- Adopt a block-by-block (module-by-module) approach to programming.
- Always check your work.

0.2.5 Computer Aided Software Engineering

Although the software engineering industry is relatively new, we can be justly proud of the achievements. *Computer aided software engineering* (CASE) is one of the great breakthroughs for the industry. It is using software to generate software (and more recently, hardware). By using CASE, the SDLC is significantly reduced, so that very sophisticated, powerful software can be developed in an astonishingly short period of time. It has been shown that CASE has the potential to reduce development time of a system by as much as 80%.

CASE tools make the software engineer more productive by generating code which can then be accessed and modified. Examples of CASE tools are Gupta Team Developer, Oracle, Rational Rose, Delphi, and Live model. You will learn more about CASE tools in your more advanced courses.

0.3 Rudiments of Algorithm Development

In this course, you will be refining your ability to develop algorithms and then write computer programs to implement them. In order to be good at this, you need to learn how to write algorithms. The rationale for this is simple: A computer program is the implementation of an algorithm in a particular programming language. It is therefore imperative that you know about algorithms.

In this section, we will cover the following:

- Definition of an Algorithm
- Components of an Algorithm
- Sequential Structures
- Selection Structures
- Iterative Structures
- Illustrations
- Flowcharting
- Stepwise Refinement

0.3.1 Definition of an Algorithm

An algorithm is a procedure for solving a specific problem in a finite number of steps. More formally, an algorithm is a well-ordered collection of unambiguous operations that when executed, produces a result and terminates in a finite amount of time. The algorithm is typically written in a stepwise manner, which facilitates easy implementation in a programming language.

Example 0-18: Below is an algorithm for accepting student records and writing them to a file.

```
While (User wishes to continue) do the following:
        Accept student information;
        Validate student information;
        If      (information is valid),
                Write to student file;
        End-If;
        Else
                Display error message
        End-Else;
End-While;
```

The above example reveals a number of important control structures of algorithms:

- Sequential Structures: the order of the instructions is important.
- Selection Structures: These control the decision-making aspect(s) of the algorithm.
- Iterating Structures: These control the repetitive aspect(s) of the algorithm.
- Recursion: This feature is not illustrated in the example; it will be revisited later.

Successful software development is always preceded by careful research and planning. Algorithm development is an integral part of this research and planning stage.

0.3.1 Definition of an Algorithm (continued)

In software engineering, you will discover that there are several techniques for representing algorithms. These include (but are not confined to) flowcharts, Warnier-Orr diagrams, pseudo code, and formal methods. In this course, we will concentrate on pseudo-code. By pseudo-code, we mean linguistic language (for instance in English), expressed in a manner that facilitates easy conversion to a HLL program code.

0.3.2 Components of an Algorithm

An algorithm may consist of the following components:

- Variable(s) and data type(s)
- Statement(s) and expression(s)
- Punctuation
- Records, Arrays, and other Abstract Data Types
- Subroutine(s)
- Control structures

The algorithm is written in a logical, step-by-step manner (using pseudo-code) that will ensure a solution to the problem it addresses. Any violation of the logical flow will render the algorithm incorrect. There may be alternate solutions, but if the proposed algorithm does not effectively address the problem, it is considered incorrect.

Variables and Data Types:

A *variable* is a data item whose value varies during algorithm (program) execution. For example, in the expression $3x^2 + 2x + 4$, **x** is a variable. Other examples of variables are **DateOf Birth**, **Surname**, **FirstName**, **MaritalStatus**, **Gender**.

Variables must have unique (descriptive) names. Also, by convention, variable names are not written with gaps or spaces. Finally, we normally specify of what data type the variable is. Examples of basic (*primitive*) data types are:

- **Real Numbers**
- **Integers**
- **Characters**
- **Strings**
- **Boolean**

In a programming language, more complex data types are typically built from the above mentioned basic types. Examples of more complex data types are: records, arrays, linked lists, sets, etc. We will briefly look at records and arrays later.

Example 0-19: Below are examples of variable declarations:

```
Let DateOfBirth be an integer;
Let FirstName, LastName, MiddleInitial, MyName be strings;
Let Gender be a character;
Let F, x be real numbers;
```

0.3.2 Components of an Algorithm (continued)

Statements and Expressions:

A *statement* is essentially a sentence, phrase or expression that makes sense to your algorithm. An *expression* is simply part of a statement. The convention is to terminate all statements with a semicolon. A common type of statement is the *assignment* statement. An assignment statement assigns value to a variable. There are two forms of assignment statements:

- A variable takes on the value of *literal* (absolute value).
- A variable takes on the evaluation of an (arithmetic) expression.

Assignment statements are represented differently in different textbooks. Equally, their implementation varies from one programming language to the other. For this course we will use the notation illustrated in the following example:

Example 0-20: Below are some assignments (assume the declarations of Example 19):

```
/* Here, F takes the evaluation of an expression: */
F := 3x^2 + 3x + 4;  /* is read:   Set  F to  3x^2 + 3x + 4  */
…

/* Here, F & My-Name take the value of a literal, respectively: */
F := 0;
MyName := “Bruce F. Jones”; /* is read, Set MyName to “Bruce F. Jones” */


…
/* Here, MyName takes on the value of concatenation of other string variables: */
FirstName := “Bruce”; MiddleInitl := “F.”; LastName := “Jones”;
MyName := FirstName + MiddleInitl + LastName;
```

Other kinds of statements include *subroutine calls, iterative statements*, and *selection statements.* These will be discussed shortly.

As alluded earlier, an expression is part of a statement that evaluates to some value. An expression is made of operators and variables or literals. In some languages, statements are also regarded as expressions. However, in the interest of clarity, the distinction is made here. In Example 20, the right hand part of each assignment statement is an expression.

Two kinds of expressions are prevalent in algorithm development and programming: *arithmetic expressions* and *Boolean expressions*. An arithmetic expression evaluates to alphanumeric data (as in Example 20). A Boolean expression (also referred to as a *condition*) evaluates to **true** or **false**. The following are examples of Boolean expressions.

0.3.2 Components of an Algorithm (continued)

Example 0-21: Boolean expressions:

```
(Today = "Friday")
(Season = "Winter")
(x > 10)
(x > 10) AND (Season = "Winter")
(x > 10) OR (Season = "Winter")
```

Boolean expressions do not occur on their own, but are usually stated as part of selection or iterative statements. These will be discussed later (there is actually an example of a selection statement in Example 23 below).

Punctuation:

As mentioned earlier, all statements are terminated by a semicolon. Most programming languages adopt this convention. Also, it is good practice to give the algorithm a brief, descriptive name. Algorithms translate to programs and in all programming languages, programs are given unique names. Variables and subroutines (to be discussed shortly) are also given unique names. Like variables, the name for a subroutine or algorithm should not include gaps or spaces. Finally, you can write comments that clarify your algorithm by inserting them between the symbols **/*** and ***/**, or after the symbol **//**.

Records and Arrays:

Record: A record is a compound data type, consisting of (at least two) members (fields) possibly (but not necessarily) of different data types. In your algorithm, you must first define the record, and then declare variables of that record.

Example 0-22: Below is an example of how to define a record, and then declare variables of it.

```
Let StudRecord be a record consisting of:
StudNumber, an integer;
StudSurname, a string of length 15;
StudFname, a string of length 15;
StudDOB, an integer;
StudMajor, a string of length 25;
….

Let ThisStud, ThatStud be defined on StudRecord;
```

You can then refer to the *fields* of the record by using the notation:

VariableName.Fieldname

0.3.2 Components of an Algorithm (continued)

Example 0-23: Assuming the declarations of Example 22, we may have the following statements:

```
ThisStud.StudSurname := "Harris"; ThisStud.FirstName := "Terrence";
          . . .

 If (ThisStud.StudSurname = ThatStud.StudSurname)
          Print ("You must be related!");
 End-if;
```

Array: An array is a finite list of items of a specific base type. In defining an array, you must specify the name of the array, the number of items, and the base type (the base type can be any valid type, including an advanced type).

Example 0-24: Below are examples of array declarations:

```
Let Sale be an array of 30 real numbers;
Let Student be an array of 15 StudRecord items; // Assume StudRecord as defined in Example 22
Let Counter be an array of 20 integers;
```

Having defined the array, you can access elements of the array by using array subscripts. The convention for specifying an array subscript is ArrayName [Subscript]. The square bracket is required here to indicate array subscripting.

Example 0-25: Below are examples of array subscripting:

```
/* Assume x is defined as integer and Total is defined as real number*/
Let x be an integer; // Array subscript
Let Total be a real number;
Sale[1] := 120,000.00; // Go Josiahs! Go!
Sale[2] := 200,000.00;  /* Five star!!! */
Sale[3] := 100,000.00;

…
Sale[x] := 0;
 . . .
Total :=  Total + Sale[x];
```

Abstract Data Types:

Abstract data types (ADTs) are advanced data types that typically contain data items and operations defined to manipulate those data items. You will learn more about ADTs later in the course. For now, just note (rather remember) that in contemporary programming languages, ADTs are typically implemented as classes.

0.3.2 Components of an Algorithm (continued)

Subroutines:

A *subroutine* (also called *subprogram*) is a portion of an algorithm that carries out a specific task or set of related tasks. An algorithm may consist of several sub-routines. The following are features of a subroutine that you should be familiar with:

1. Ideally, the subroutine should be written in a manner that makes it coherent and therefore applicable to other algorithms (programs).
2. The subroutine may have parameters (arguments) — variables with which it is invoked (called).
3. Other internal variables may also exist within the subroutine.
4. The subroutine always returns control to the statement following the calling statement. A value may also be returned.
5. The subroutine may return a value to the calling statement. This is initiated by a **Return-Statement** within the body of the subroutine, just prior to exit (typically near the end).

Here are some additional consequential principles to remember:

1. If the subroutine has parameters, then it must be called with arguments. On the call, the arguments are copied into the corresponding parameters. Corresponding argument-parameter pairs must be therefore be of the same data type. Arguments are specified within parenthesis and separated by the comma.
2. If the subroutine returns a value, it is called by including it in an expression or as he subject of an assignment statement.
3. If the subroutine does not return a value, then it is called by simply specifying its name along with any required argument.

Examples of subroutines are provided ahead (section 0.3.6). In contemporary programming languages, subroutines are implemented as methods (as in Java), functions (as in C++), or procedures (as in Pascal).

Control Structures:

By *control structures*, we mean how the logic of the algorithm is specified. This is extremely important. Four mechanisms are used, and they will be discussed in the upcoming subsections. They are:

- Sequential structures
- Selection structures
- Iteration structures
- Recursion

0.3.3 Sequential Structures

By sequential structures, we mean statements or instructions in sequence. The order in which these instructions are carried out is important. Sequential statements are executed consecutively.

0.3.4 Selection Structures

Selection structures facilitate decision based or certain pre-conditions. Two selection structures are common: the **If-Structure** and the **Case-Structure**. How these structures are implemented will vary from one programming language to another. However, the format that you will use for your algorithms are provided in figure 0-10 below:

Figure 0-10: Selection Structures

The **If-Structure** has the form:

```
If (<Condition>)
   <Statement(s)>
End-If
[Else
   <Statement(s)>
End-Else]
```

The **Case-Structure** has the form:

```
Case <Variable> | <Expression> is
<Value_1>: <Statement(s)>
<Value_2>: <Statement(s)>
      …..
<Value_N>: <Statement(s)>
Otherwise: <Statement(s)>
End-Case
```

Convention: The angular brace (<…>) is used to mean, the programmer supplies the pertinent detail(s). The square brackets ([…]) are used to indicate that whatever is enclosed therein is optional.

An **If-Structure** is applicable in situations where different circumstances warrant different actions, or an action (or set of related actions) is contingent on the occurrence of a particular condition. The **Case-Structure** is applicable in situations where a variable (or arithmetic expression) could have one of several distinct values and for each value, a specific task (or set of related tasks) is required.

Example 0-26: Suppose that you wanted accept a number from the user, and check to see whether it is an even number. To do that, you simply need to divide the number by 2, and if there is no remainder, it is even. In CS, we often call the remainder of an integer division the *modulus* (abbreviated mod). Thus, **N** mod 5 is the remainder of **N** divided by 5. The required algorithm is shown below:

Algorithm: FindEven

```
START
  Let AnyNumber be an integer;
  Prompt the user for AnyNumber;
  If   (AnyNumber mod 2 = 0)
     Print(AnyNumber + “is an even number”);
  End-If;
  Else
     Print (AnyNumber + “ is an odd number.”);
  End-Else;
STOP
```

0.3.4 Selection Structures (continued)

Example 0-27: The algorithm below displays a menu to the user, prompts the user to select a menu item, and calls a subroutine for each possible option taken.

Algorithm: GenericMenu

```
START

  Let Option be an integer;
  Display the following menu options:
         1.      Take Option 1
         2.      Take Option 2
         3.      Take Option 3
         4.      Take Option 4
         5.      Take Option 5
         6.      Take Option 6

 Prompt the user for Option;
  Case Option is
  1:     Option1; // Invoke subroutine Option1
  2:     Option2; // Invoke subroutine Option2
  3:     Option3;  // Invoke subroutine Option3
  4:     Option4;  // Invoke subroutine Option4
  5:     Option5;  // Invoke subroutine Option5
  6:     Option6;  // Invoke subroutine Option6
  Otherwise: Print ("Invalid option taken.");
  End-Case;

STOP.

/* The subroutines would then follow */

Subroutine:  Option1
START
 … /* Instructions for Option1 goes here */

STOP

…

Subroutine:  Option6
START
 … /* Instructions for Option6 goes here */

STOP
```

0.3.5 Iteration Structures

An iteration structure is a structure which forces repetitions of the instructions specified within the structure. Four iteration structures are common: the **While-Structure** the **Repeat-Until-Structure**, the **For-Structure**, and **Recursion**. Recursion will be revisited shortly. The typical format of each of the other three control structures is shown in figure 0-11.

Figure 0-11: Generic Representation of Iteration Structures

The **While- Structure** has the following form:

```
While (<Condition>) Do the following:
          <Statement(s)>
   …
 End-while;
```

The **Repeat-Until-Structure** has the following form:

```
Do the following:
     <Statement(s)>
     ….
Until (<Condition>)
```

The **For- Structure** has the following form:

```
For <Variable>|<Expression>  : = <Value1> To <Value2> With increments of <Value3>,  Do the following:
<Statement(s)>
…
End-For
```

Referring to the **While-Structure**, the condition specified evaluates to either true or false. If it is true, the statements enclosed in the loop are executed consecutively. If the condition evaluates to false, then control goes to the first statement beyond the **End-While** tag. Additionally, please note:

1. If the maximum possible number of iterations is N, the **While-Structure** ensures a minimum of zero iterations.
2. Some languages will allow a premature exit from a while loop. This, we will denote by the word **Exit**.
3. Some languages will allow a premature iteration of a while loop. This we will denote by the word **Iterate**.

0.3.5 Iteration Structures (continued)

In the **Repeat-Until-Structure** structure, the statements of the loop are executed until the condition specified evaluates to true; when this happens, control then goes to the first statement beyond the **Until** tag. Additionally, please note:

1. If the maximum possible number of iterations is N, the **Until-Structure** ensures a minimum of 1 iteration.
2. Some languages will allow a premature exit from a while loop. This, we will denote by the word **Exit**.
3. Some languages will allow a premature iteration of a while loop. This we will denote by the word **Iterate**.

The **While-Structure** is applicable in situations where an action (or set of related actions) is to be repeatedly carried out as long as a particular circumstance (condition) prevails. The **Repeat-Until-Structure** is applicable in situations where an action (or set of related actions) is to be repeatedly carried out until a particular circumstance (condition) forbids its execution.

Turning to the **For-Structure**, the statements of the loop are executed until the condition (Variable = Value2) is true; control then goes to the first statement beyond the **End-For** tag. For the first iteration, **Variable** is assigned the value of **Value1**; for each subsequent iteration, **Variable** is incremented by **Value3** and tested at the **End-For** tag. If **Value3** is not specified, its default is 1. Additionally, please note:

1. If the maximum possible number of iterations is N, the **For-Structure** ensures a minimum of 1 iteration.
2. Some languages will allow a premature exit from a while loop. This, we will denote by the word **Exit**.
3. Some languages will allow a premature iteration of a while loop. This we will denote by the word **Iterate**.

The **For-Structure** is applicable in situations a variable is to be varied (typically in equal increments) from an initial value to a final value, and at each value (increment), an action (or set of related actions) is to be carried out.

Recursion is the act of an algorithm calling itself. Typically, what we mean is that at least one subroutine in the algorithm calls itself. Recursion occurs in many aspects of programming as will become clear later in the course. Every recursive algorithm can be replaced by a non-recursive one, but developing the latter for certain problems is sometimes difficult. While recursion is supported in contemporary programming languages, many traditional languages (for example COBOL and RPG-400) did not support the principle in their earlier years. Since recursion is integral to a course in data structures and algorithms, we will be revisiting the topic at various points throughout the course.

0.3.6 Illustrations

Let us cement the principles we have covered so far by considering a mathematical problem, and its CS solution. The problem we will consider is finding the factorial of a number. The factorial of a positive integer N, (denoted N!) is given by the notation

$$\boxed{N! = N(N-1)(N-2)\ldots(N-I+1)\ldots(1)}$$

For instance, 5! = 5*4*3*2*1 = 120. We want to develop an algorithm for finding N! What this means is, given any positive integer input (denoted by N), we would like to calculate and return the factorial of N.

Example 0-28: Figure 0-12 provides four alternate solutions to the factorial problem.

Example 0-29: Let us now develop the algorithm for a program that will allow the user to indefinitely enter numbers for which the factorial will be produced. This will continue until the user quits. The algorithm is shown in figure 0-13.

Figure 0-12: Solutions to the Factorial Problem

```
Subroutine: Factorial (Number):_Returns a real number        /* Using a For-Loop */
START
  Let Number, x  be positive integers and Fact be a real number;
  Fact := Number;
  For x := Number -1 to 1, With increment –1, Do
            Fact := Fact * x;
  End-For;
  Return Fact;
STOP
```

```
Subroutine: Factorial (Number):_Returns a real number        /* Using a recursive subroutine */
START
  Let Number be a positive integer and Fact be a real number;
  If ((Number = 1) OR (Number  = 0))
     Fact := 1
 End-If;
 Else
   Fact := Number * Factorial(Number -1);
 End-Else;
 Return Fact;
STOP
```

```
Subroutine: Factorial (Number):_Returns a real number        /* Using a While-Loop */
START
  Let Number, x  be positive integers and Fact be a real number;
  Fact, x := Number;
  While (x > 1) Do the following
           Fact := Fact * (x-1);
           x := x-1;
   End-While;
   Return Fact;
STOP
```

```
Subroutine: Factorial (Number):_Returns a real number        /* Using another While-Loop */
START
  Let Number, x  be positive integers and Fact be a real number;
  Fact:= Number;
  x := Number – 1;
  While ((x – 1) >= 0) Do the following
           Fact := Fact * x;
           x := x-1;
  End-While;
  Return Fact;
STOP
```

Figure 0-13: Algorithm to Produce the Factorial of any Positive Integer

```
Algorithm: AnyFactorial
Input Variable: AnyNumber, an integer;
Output Variable: AnyNumberFact, a real number;
Working Variables: More, a character variable;

Main Routine:
START
More := 'Y'; /* User wishes to continue */
While (More = 'Y') do the following: // While user wishes to continue
    Prompt for and accept AnyNumber;
    AnyNumberFact := Factorial(AnyNumber);
    Display AnyNumberFact;
    Prompt the user to specify Y(es) or N(o) to indicate whether he/she wishes to continue, and
      store the response in More;
End-While;
STOP

// Subroutine Factorial could be any of the versions shown in figure 0-12
Subroutine Factorial(Number): Returns a real number
START
 // Let Number a positive integer
  Let x  be positive integer
  Let Fact be a real number;
  Fact := Number;
  For x := Number -1 to 1, With increment –1, Do
            Fact := Fact * x;
  End-For;
  Return Fact;
STOP
```

0.3.6 Illustrations (continued)

Example 0-30: Develop an algorithm that will accept as input, the year and month, and return as output, the number of days in the month. A crude solution is shown in figure 0-14. Observe that this solution can be further simplified by introducing an array of 12 integers, where each position in the array represents a month of the year. Also, note that the check for a leap year is a bit more sophisticated than shown. The refined pseudo-code is shown in the lower portion of the figure (notice that its length is about half the length of the crude solution).

Figure 0-14: Algorithm to Determine Number of Days in Month

```
// Crude Solution
Subroutine: DaysofMonth  (Year, Month): Returns an integer
START
  Let Year, Month, Days be positive integers;
  Let LeapYear be a Boolean flag;
  LeapYear:= False;
  If (Year Mod 4) = 0
    LeapYear := True
  End-If;
  Case Month is
  1: Days :=31;
  2: If LeapYear
        Days :=29;
      End-If;
      Else
        Days :=28;
     End-Else;
  3,5,7,8,10,12: Days := 31;
  4, 6, 9, 11: Days := 30;
  Otherwise: Days := 0;
  End-Case;
  Return Days;
STOP
```

```
// Refined solution
Subroutine: DaysofMonth  (Year, Month): Returns an integer
START
  Let Year, Month be positive integers;
  Let Days be an array of 12 integers initialized to {31, 28, 31, 30, 31 30, 31, 31, 30, 31, 30, 31};
  // Assume array indexes of 1 .. 12. For C-based languages like C++, Java, etc. either adjust indexes by -1
  // or define the array length of 13, while ignoring index 0.
  If (Year mod 400) is 0) OR ((Year Mod 4) is 0 AND (Year Mod 100) <> 0))
     Days[2] := 29; // This is a leap year, so adjust February
  End-If;
  Return Days[Month];
STOP
```

0.3.7 Flowcharting

An alternate approach to algorithm development is flowcharting. Generally speaking, flowcharts are not as flexible as algorithms. However, one significant advantage of the flowchart is that it provides a graphic illustration of the logic of the program represented. Figure 0-15 illustrates the symbols used in flowcharting; figure 0-16 shows the flowchart constructs for sequential, selection and iteration structures; and figure 0-17 shows the flowchart for the program **AnyFactorial** of Example 29.

Figure 0-15: Symbols Used in Flowcharting

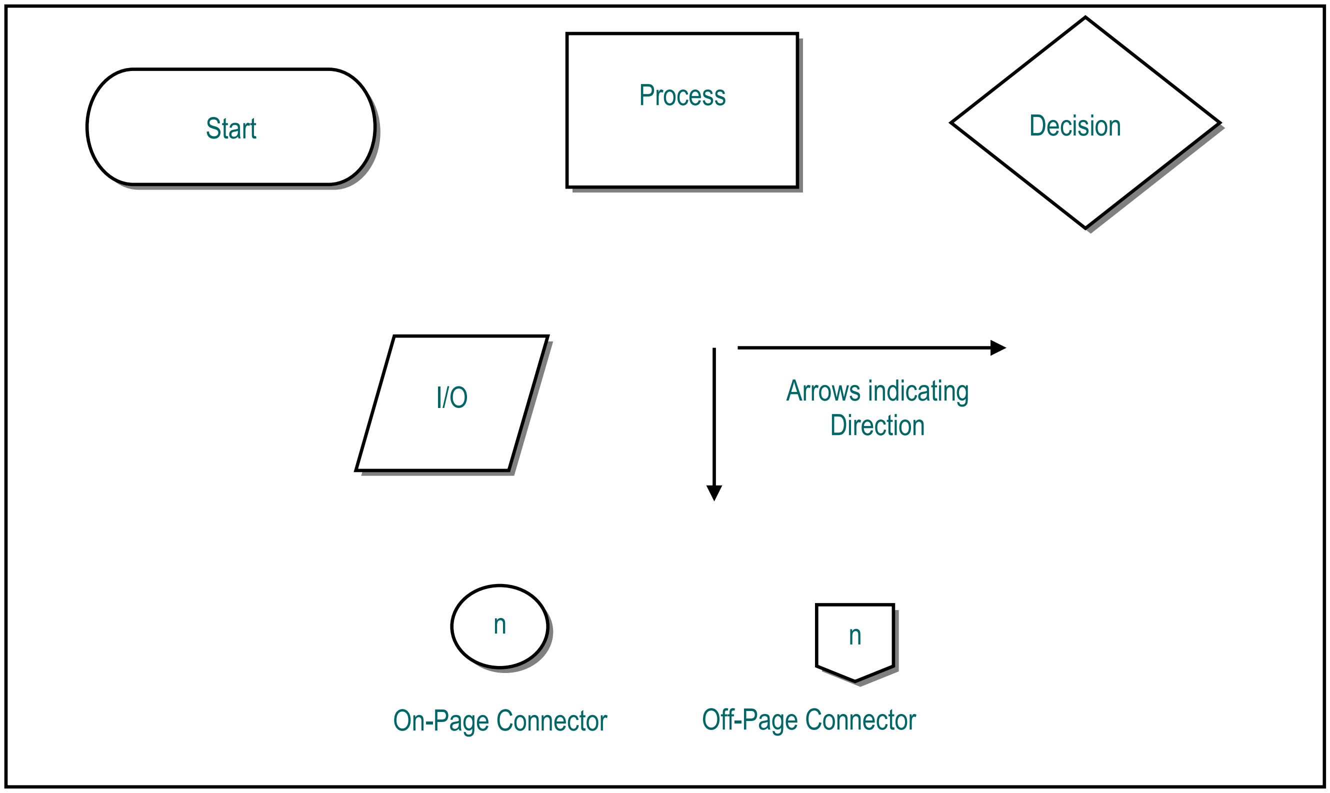


Figure 0-16: Control Structures for Programming Flowcharting

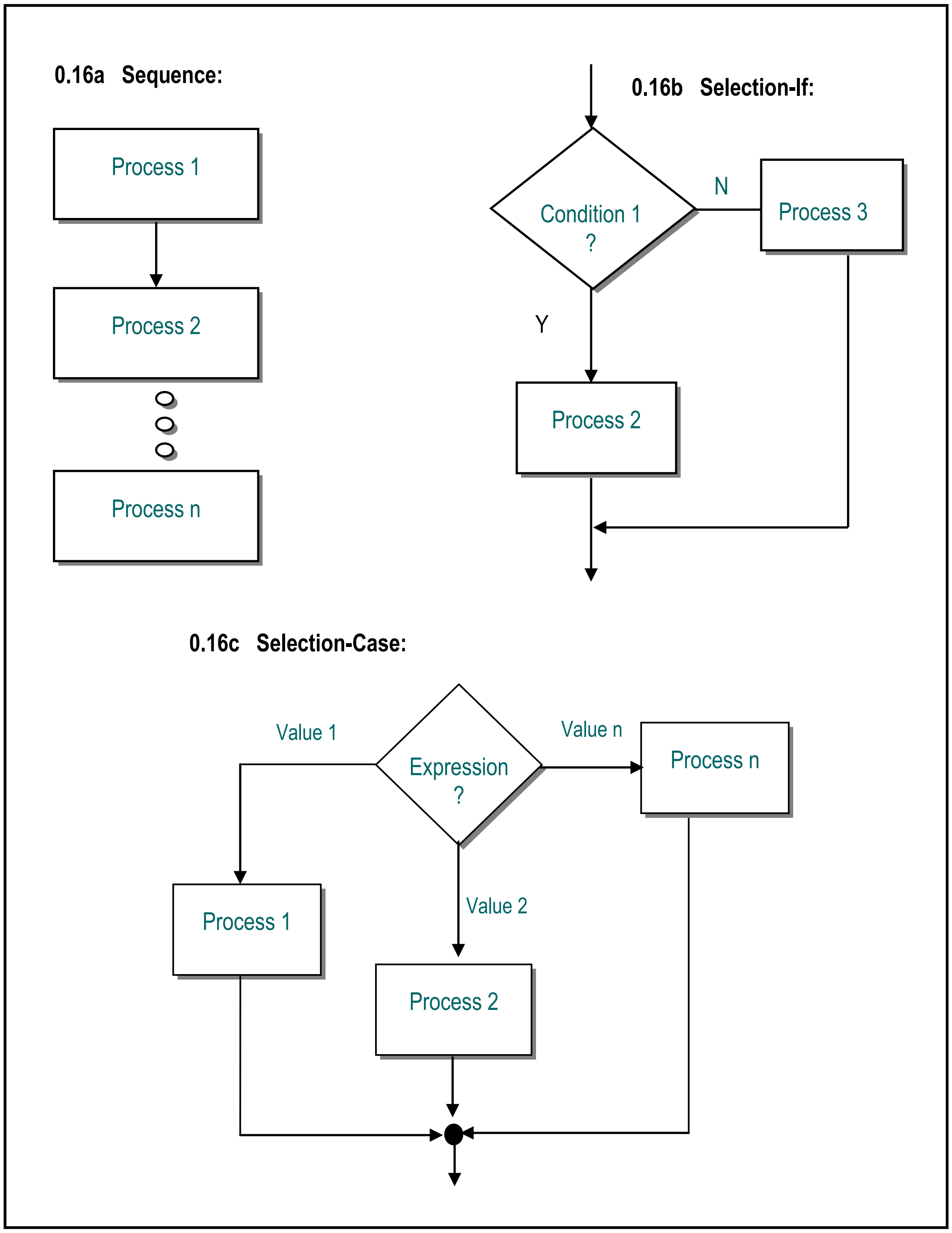


Figure 0-16: Control Structures for Programming Flowcharting (continued)

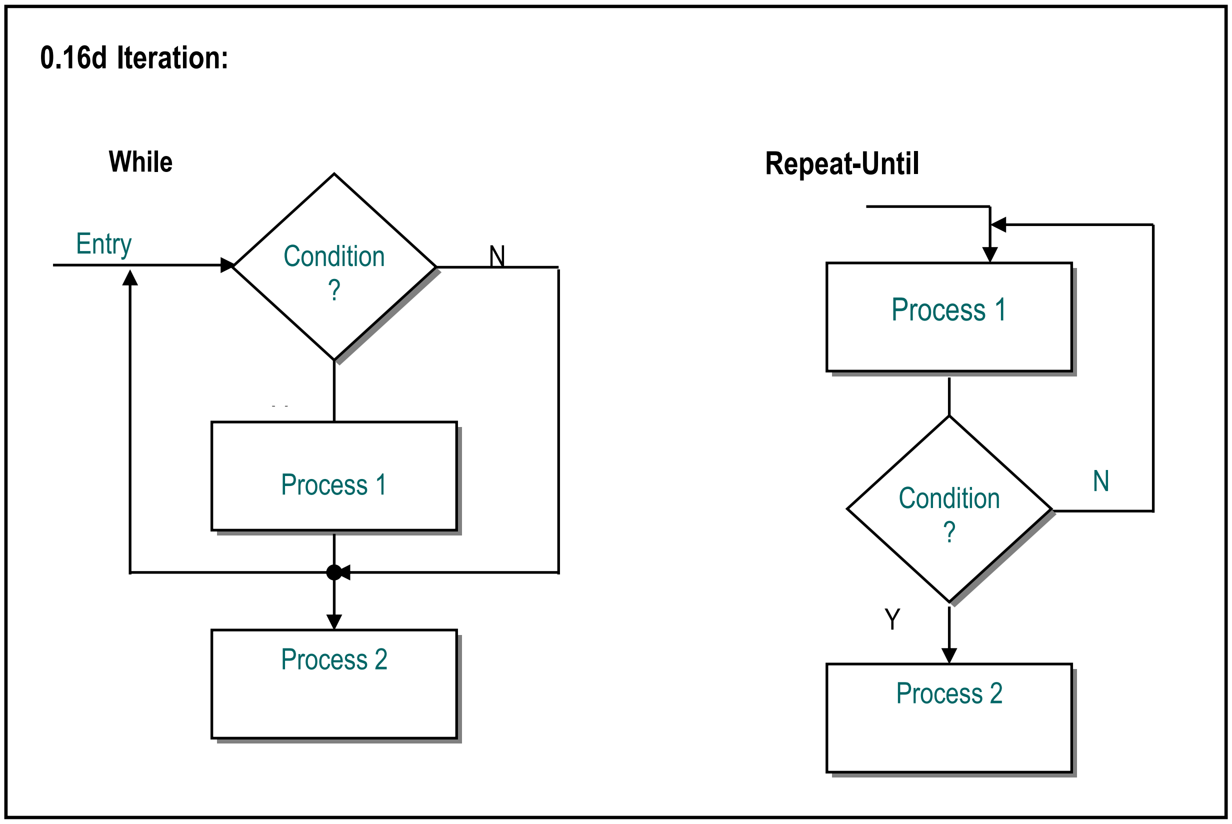
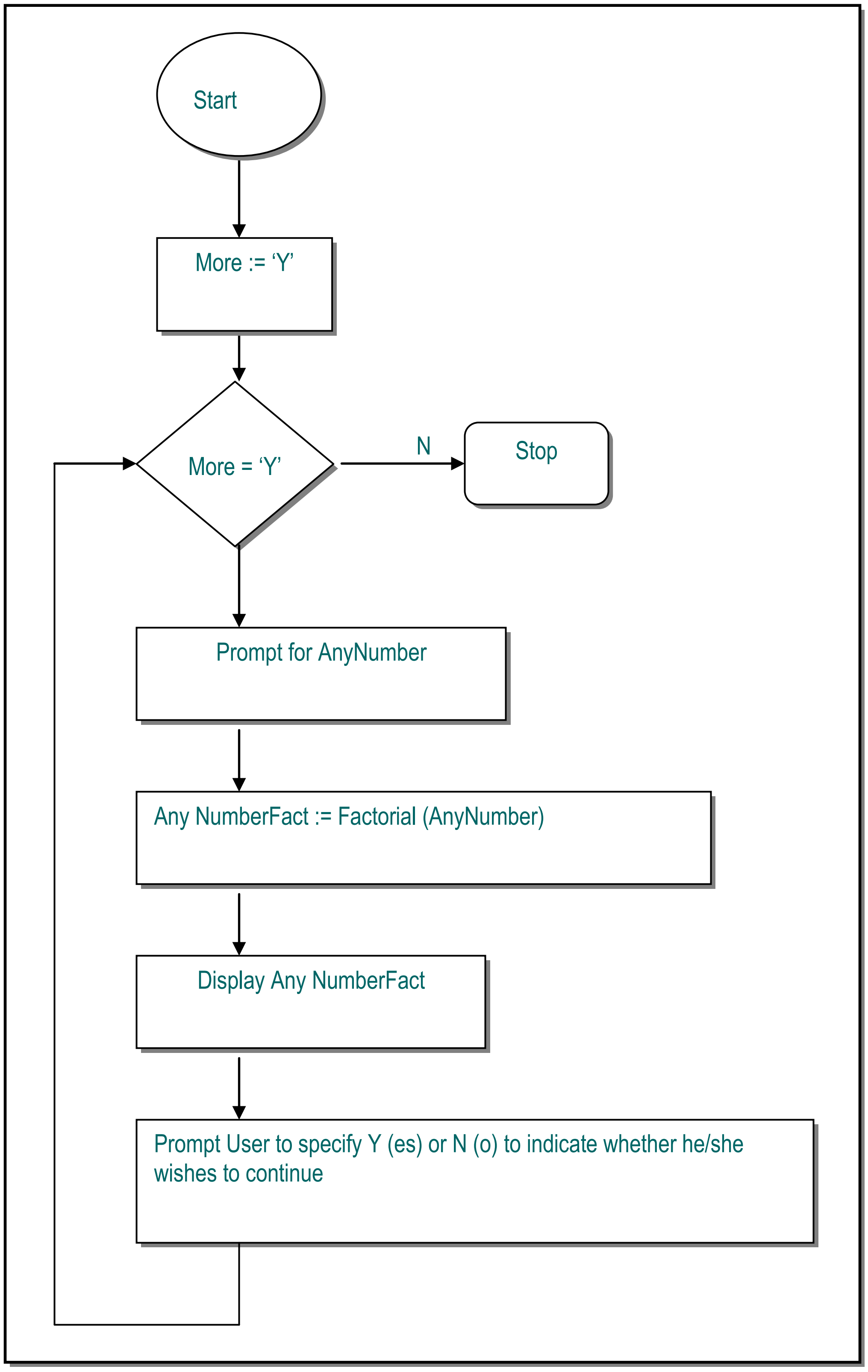


Figure 0-17: Flowchart for Program AnyFactorial


Start
More := 'Y'
More = 'Y'
N
Stop
Prompt for AnyNumber
Any NumberFact := Factorial (AnyNumber)
Display Any NumberFact
Prompt User to specify Y (es) or N (o) to indicate whether he/she wishes to continue

0.3.8 Stepwise Refinement

You may have heard about the principle of "*divide and conquer*." This principle is used in various aspects of life. In programming, what it really means is that you should always break down a (programming) problem into a set of smaller, more manageable problems. You then repeatedly refine the smaller problems. This is what we call *stepwise refinement*. The principle can be applied to simple problems as well as complex problems.

Example 0-31: As an illustration, let us develop an algorithm for preparing food. This algorithm will be developed as a transportable subroutine that accepts as argument, the food to be prepared. The pseudo-code is shown in figure 0-18.

Figure 0-18: Food Preparation Algorithm

```
Algorithm: FoodPrep(Food)
Let Food be a string;
Let FoodCooked, ValidFood be Boolean;

START
  ValidFood := ExamineFood(Food); /* subroutines sets ValidFood flag */
  If (ValidFood)
      PreparePot; /* subroutine call */
      PrepareFire; /* subroutine call */
      PrepareFood(Food); /* subroutine call */
      Put Food in pot;
      Put Pot on fire;
      FoodCooked := CookFood(Food); /* subroutine sets FoodCooked */
  End-if
  Return FoodCooked;
STOP
********************************************************************************
Subroutine: ExamineFood (InFood): Returns boolean
Let InFood be a string;
Let FoodValid be Boolean initialized to False;
START
  /* Let as an exercise for you */
  Return FoodValid;
STOP
********************************************************************************
Subroutine: PreparePot
START
  /* Let as an exercise for you */
STOP
********************************************************************************
Subroutine: PrepareFire
START
  /* Let as an exercise for you */
STOP
********************************************************************************
```

Figure 0-18: Food Preparation Algorithm (continued)

```
Subroutine: PrepareFood (InFood)
Let InFood be a string;
Let FoodValid be Boolean initialized to False;
START
  If (InFood = "Rice")
    …  /* Instructions for Rice preparation */
  End-If
  …  /*   May include instructions for various foods */

  If (InFood = "Chicken")
    …  /* Instructions for Chicken preparation */
  End-If
 STOP
*****************************************************************************
Subroutine: CookFood (InFood) Returns Boolean
Let InFood be a string;
Let isCooked be Boolean, initialized to False;
Let TimeOnFire, CookTime be integers; /* to store time in minutes */

START
  Case InFood is
    "Rice":        CookTime := 45;
      …  /*        May include cook times for various foods */
   "Chicken": CookTime := 180;
  End-Case;

  Put Pot on Fire;
  TimeOnFire := 0;
  While (NOT isCooked) do the following:
      Keep Pot on Fire;
      If (TimeOnFire = CookTime)
          isCooked := True;
      End-If;
      Test Food;
       If (Food is cooked)
           isCooked := True;
       End-If;
        Increment TimeOnFire by 2;
  End-While;

   Return isCooked;
STOP
```

0.3.8 Stepwise Refinement (continued)

Stepwise refinement is usually top-down. There is no set rule regarding when to stop; that is a matter of discretion; however, in many cases, this will be obvious. The rule of thumb is, stop when you are comfortable with the level of detail provided. Another guiding principle is to remember that your algorithm may be used by anyone; therefore clarity is of paramount importance.

Finally, remember that accurate, efficient algorithms lead to accurate, efficient programs. The converse is also true: inaccurate or inefficient algorithms lead to inaccurate or inefficient programs.

0.4 Rudiments of Program Development

Once you have developed an algorithm for problem at hand, the next logical step is to develop the program that implements the algorithm. This section provides you with an overview of programming. We will cover the following:

- Overview of Programming
- Types of Programming Languages
- Program Specification
- Overview if File Processing

0.4.1 Overview of Programming

Programming is the act of implementing an algorithm in a specific high level language (HLL). An HLL program therefore, is the implementation of a particular algorithm, via specific rules of the language.

A programmer is a person who writes programs, according to specifications derived or obtained. The programmer learns and applies various programming techniques which allow for efficient, accurate code. Remember: the compiler or interpreter converts source code, written by the programmer, into object code which the machine understands. Professional programmers are typically familiar with several programming environments. They learn to do their job in different environments, using different tools.

0.4.2 Types of Programming Languages

There are five major types (paradigms) of programming languages:

- Procedural languages
- Object oriented languages
- Hybrid languages
- Functional languages
- Declarative languages

0.4.2 Types of Programming Languages (continued)

Procedural Languages: Procedural languages are used in procedural programming. They force the user (programmer) to specify instructions in a step-by-step manner. The programmer concerns himself with functions and procedures. Procedural languages predate contemporary languages which are predominantly object-oriented. Examples include Pascal, Fortran, C, COBOL, Algol, etc.

Object-Oriented Languages: Object-oriented programming languages (OOPLs) are used in object-oriented programming (OOP). They force the programmer to program with classes. Methods (functions and/or procedures) are encapsulated within the classes. Classes communicate with each other via messages. Examples of OPPLs are Eiffel, SmallTalk, Java, C#, etc.

Hybrid Languages: Hybrid Languages are procedural languages which have been upgraded to OO languages. They facilitate both procedural programming and OOP. For this reason, they are often (incorrectly) referred to as OOPLs. Examples of hybrid languages include C++, Visual Basic, Object COBOL, Object Pascal, Ada, etc.

Functional Languages: Functional languages are special kinds of procedural languages. However, they emphasize knowledge representation and are therefore widely used in AI. Examples of functional languages are LISP, CLOS, and ML.

Declarative Languages: Declarative languages are more sophisticated (and powerful) than other types of programming languages. However, they are also more limited in their scope. Declarative languages are typically used to manage databases (DB) and knowledge bases (KB). They are higher level languages than third generation languages; in fact, most 4GLs are declarative languages. Examples of declarative languages are SQL (Structured Query Language), Ideal, KQL (Knowledge Query Language). .Arguably, we may classify hypermedia markup languages as declarative; or we may classify them as being part of the fifth generation languages. Examples of these include HTML (Hypertext Markup Language), VRML (Virtual Reality Markup Language), and XML (Extensible Markup Language).

0.4.3 Program Specification

A program specification (commonly shortened as program spec) is a blueprint for a program. Typically, the program spec is prepared by a software engineer, system analyst, or some senior person on a software engineering project. It may also be prepared by a knowledgeable programmer.

Once the program specs are prepared, they are kept in a safe place that is easily accessible to programmers on the project (for example, a secured network shared folder/directory). Whenever a programmer is required to write a particular program, he/she would first access ("pull") the spec for that program.

0.4.3 Program Specification (continued)

Typically, the program spec has the following components:

- System Name
- Subsystem/Module Name
- Program Name
- Program Description Brief
- Author, date of preparation, date of last modification
- Input Files
- Output Files
- Validation Rules
- Special Notes
- Program Outline — usually consisting of UML diagram(s) and pseudo-c
- Sample Inputs/Outputs

Please note the following points of clarification:

2. For a considerable portion of this course, the first two components will be substituted by the institution name, department name, and the course title.
3. You are being taught to write small, independent programs. Bear in mind however, that in industry, programs (written by different individuals) comprise larger software systems and applications. Your learning will necessarily be incremental, starting with simple to moderately complex programs in this course, and advancing to more complex software systems in your upper level courses. .
4. Input Files are read by the program, output files are files written to by the program. Note that a file may qualify as both input and output.
5. Validation Rules are conditions which input variable must satisfy before they can be accepted for processing. For example, if your program is to accept date as an input, a validation rule would be that the date must be valid.
6. Special Notes include significant considerations that affect the program logic. Calculation rules would qualify as special notes.
7. Assuming an OOP environment, the program outline would typically be a set of one or more Unified Modeling Language (UML) class diagrams, followed by your pseudo-code.

Your program may consist of various classes. A Java class is simply the holding area for your Java code. For each class, you will present the UML (Unified Modeling Language) diagram for the class and then clarify its internal methodological components via pseudo-coding. Figure 0-19 shows a UML class template.

Figure 0-19: UML Class Diagram Template

```
Class-Name   // The name of the class

// State the data items of the class; for each, state the access keyword, data-type, and variable-name
<Access-Keyword> <Data-type> <Variable-Name1>
<Access-Keyword> <Data-type> <Variable-Name2>
. . .

// State the method signatures for the class. For each method, state the return-type, method-name, and
// parameters
<Return-type> <Method-Name> (<Parameters>)  // Initially, your methods will not have parameters
<Return-type> <Method-Name> (<Parameters>)
```

0.4.3 Program Specification (continued)

Note: In the original UML notation, the data type is specified after the property (data item or method). However, since the Java language specifies data type first in variable and method declaration, in the interest of clarity, this convention is also used for the UML notation. Initially, your methods will not have parameters. However, as you learn more, you will be able to write methods with parameters.

Example 0-32: Let's take a simple example of writing a program to generate a random bar chart based on specifications keyed in by the user. The user will specify the number of random bars to be generated, where each bar has a length in the range of 1 . . 40. The program will print a character that the user also specifies at each position (going from left to right). Figure 0-20 shows a program specification that could be used for this exercise.

Figure 0-20: Sample Program Specification for Random Bar Chart Generator

Program Biography	
Institution:	**Keene State College**
Department & Course:	**Computer Science Department; ISCS140 Programming Foundations I**
Program Name:	**CP-Ass03C_JonesB**
Package Name:	**cp-ass03c_jonesb**
Program Description:	This program accepts a positive integer **n** from the user and then generates **n** bars each of random length between 1 and 40. Processing continues until the user quits.
Program Author:	Bruce Jones
Submission Date:	January 14, 2013
Sources:	Elvis Foster's *Lecture Notes in Java Programming*

UML Class Diagram
CP-Ass03C_JonesB // BarChart
public String HEADING = "Bar Charts of Bruce Farnsworth Jones" public int NumBars; public char BarChartChar
public static void main(String[] args) public static String DrawBar(int Width, char ThisChar)

Figure 0-20: Sample Program Specification for Random Bar Chart Generator (continued)

Program Outline for CP-Ass03C_JonesB

The main(String[] args) Method

```
START
        Let x, BarWidth  be integers;
        Let MAX_WIDTH be an integer, initialized to 40;
        Let ThisBarChart be a string;
        Let More be boolean, initialized to True;

        While (More) do the following // While user wishes to continue

                // Initialize ThisBarChart and obtain user inputs
                ThisBarChart := " ";
                Prompt for and accept NumBars;
                Validate NumBars; // Must be in the range desired
                Prompt for and accept BarChartChar;

                // Construct the random bar chart
                // Assume that the method Random() generates a random number between 0 and 1.
                // Assume that the method Floor(double x) returns the largest integer that is <= x
                For (x := 1 to NumBars with increments of 1) do the following
                BarWidth := 1 + (Floor(Random() * MAX_WIDTH);
                        Append DrawBar(BarWidth, BarChartChar) to ThisBarChart;
                End-For;

                // Display the bar chart, then prompt the user whether to continue processing or quit
                Display (ThisBarChart);
                Prompt for additional processing;
                If no additional processing is required More := False; End If;
        End-While; // End-While user wishes to continue
STOP
```

The DrawBar(int Width, char ThisChar) Method: Returns a string

```
START
        Let BarChartString be a string, initialized to " ";
        Let y be an integer;

        For (y := 1 to Width with increments of 1) do the following
                Append ThisChar to BarChartString;
        End-For;
        Append <NewLine> to BarChartString;
        Return BarChartString;
STOP
```

0.4.3 Program Specification (continued)

Example 0-33: Let's take a slightly more complex problem as another example. Suppose that we desire to write a program that accepts student data, perform appropriate data validation, and then write each record accepted to a file. Figure 0-21 illustrates a program spec that could be used to achieve that objective.

Figure 0-21: Sample Program Specification for Student Data Entry

Program Biography

Institution:	**Keene State College**
Department & Course:	**Computer Science Department; CS28 Data Structures & Algorithms**
Program/Package Name:	**CP_AddStudent / ds_addstudent**
Program Description:	This program allows addition of validated records to the student file.
Author:	Elvis C. Foster
Date Written:	10-09-2013
Input File(s):	CS280_StudentFile
Output File(s):	CS280_StudentFile

Validation Rules:

1. Sex must be 'M' (male) or 'F' (female.
2. Date of birth must be a valid date between 1930 and the current date.
3. GPA must be between 0 and 4.
4. Student's name must be non-blank and non-null.

UML Class Diagram for CP_AddStudent

```
StudentRecord consists of the following fields:
    Integer:      StudentNumber
    String, 15:   Surname, FirstName // Strings of 15 bytes
    String, 4:    MiddleInitial
    Character:    Sex
    Number:       DateOfBirth
    String, 20:   AddressLine1,  AddressLine2, Province, Country
    String, 30:   Major
    Real 4,2:     StudentGPA          // Decimal number
// Note: In languages like Java, this has to be implemented as a class (in which case you would need a separate UML diagram).
// However, in languages like C++ it can be implemented as a structure.

Boolean:      More
String, 75:   ErrorLine
Boolean:      AcceptanceFlag
Boolean:      ErrorExists
File:         ISCS140_StudentFile
```

```
Void MainRoutine ( )
Boolean ValidateFields(StudentRecord Student)  // accepts an instance of StudentRecord and returns a Boolean value
Boolean ValidateFields(StudentRecord Student) // Determines whether InDate is valid and returns a Boolean value
```

Figure 0-21: Program Spec for Student Data Entry (continued)

Program Outline for CP_AddStudent

Void MainRoutine

```
START
   Let Stud be a variable of type Student-Record;
   More := True;
      While (More) do the following: // While there is more required work
            If (user wishes to continue)
                  Accept Stud.StudentNumber;
                  Check ISCS140_StudentFile for record existence;
                  If  (no record with this identification exists)
                        AcceptanceFlag := True;
                        While (AcceptanceFlag = True) do the following:
                              Accept all non-key fields; // Stud.FirstName, etc.
                              ErrorExists := ValidateFields(Stud); // subroutine call to validate fields
                              If  (ErrorExists)
                                    Redisplay the fields;
                                    Display ErrorLine;
                              End-If;
                              Else
                                    Redisplay the full record for confirmation;
                                    If      (user confirms)
                                          Write new Stud record to ISCS140_StudentFile;
                                          AcceptanceFlag := False;
                                    End-If;
                              End-Else;
                        End-While;
                  End-If;
                  Else /* Trying to write duplicate record */
                        ErrorLine := "No duplicates allowed"
                        Display ErrorLine;
                  End-Else;
            End-If;
            Else /* user wishes to quit */
                  More := False;
            End-Else;
      End-While; // End-While there is more required work
STOP
```

Figure 0-21: Program Spec for Student Data Entry (continued)

Program Outline (continued) for CP_AddStudent

```
Boolean ValidateFields(StudentRecord: Student)
// Validates input fields, sets ErrorLine if required and returns a Boolean flag called Error

Let Error be a Boolean flag;
Let Student be record variable of type StudentRecord;

START
Error := False;

If      (Student.Sex <> "M") AND (Stud.Sex <> "F")
        Error := True;
        ErrorLine := "Invalid sex: must be M or F";
        Return Error;
End-If;

If      (Studennt.GPA < 0) OR (Stud.GPA > 4.0)
        Error := True;
        ErrorLine := "Invalid GPA: must be between 0 and 4"
        Return Error;
End-if;

If   (Student.Surname = Blanks) OR (Student.Surname is Null) OR (Student.FirstName = Blanks) OR
     (Student.FirstName is Null)
        Error := True;
        Error-Line := "Name must be non-blank and non-null"
        Return Error;
End-if;

If      Not ValidateDate(Student.DateOfBirth) // subroutine call
        Error := True;
        ErrorLine := "Invalid date of birth"
        Return Error;
End-if;
STOP

Boolean ValidateDate(Number: InDate)
/* Determines whether InDate is valid and returns a Boolean variable called isValid

Let InDate be a number N8,0;
Let isValid be Boolean;

START
  // Left as an exercise
STOP
```

0.4.4 Overview of File Processing

The final issue to be discussed in this overview of program development is the matter of *file processing*. File handling is a critical feature in programming. Invariably, the user wishes to store data in the computer, to access and manipulate this data when required. In order to do this, file processing is necessary.

Some programming languages handle file processing better than others. But generally speaking, except for a few exceptions (e.g. RPG-400 and COBOL), HLLs tend to have poor file handling capabilities.

Database management systems (DBMSs) are far more efficient managers of files than HLLs. For this reason, in commercial programming, what is typically done may be summarized as follows:

- The DBMS facilitates the creation and management of a database.
- The DBMS supports various HLLs, but also supports at least one 4GL (typically SQL).
- When necessary, application programs incorporate the powerful features of the 4GL to take care of file handling.

Despite the fact that HLLs are not great file handlers, it is imperative that you have an appreciation of the matter of file processing via a HLL, so that when you get to more sophisticated tools like DBMSs, you will have a solid foundation to build on. A file is simply a collection of related records. Each record is defined by data elements (fields). Example: A student record may consist of **StudentNumber**, **FirstName**, **Surname**, **DateOfBirth**, **Sex**, **Major**, **GPA**, etc. A student file would consist of several student records.

It is very good practice to determine for each file, the *primary key*. The primary key (sometimes loosely referred to as the key) is the (set of) record element(s) that uniquely defines records in the file. Referring to the student file, the key would be **StudentNumber**.

File organization and management will be thoroughly explored in more advanced courses. However, you should at this point, appreciate the different approaches to file organization:

- Sequential files — records are accessed sequentially in arrival sequence.
- Direct/random access files — records are accessed randomly (directly) via an access key.
- Indexed sequential files — records are accessed both sequentially and randomly.
- Multi-key-access files — records are accessed sequentially or randomly but there may be alternate access paths (keys).

With very few exceptions, the default file organization most HLL is sequential. However, sequential access is undesirable on most occasions that require file handling. This is so because it is too restrictive, and the access time is slow (and gets even slower as the file size increases). This will become abundantly clear to you as you learn more about CS. For now however (in this course), the file processing done will be predominantly sequential.

0.5 Summary and Concluding Remarks

We have covered various concepts under the following captions:

- Overview of Computer Hardware
- Overview of Computer Software
- Rudiments of Algorithm Development
- Rudiments of Program Development

These are fundamental CS concepts. Mastery of them is imperative if you intend to pursue a career in CS. This summary is deliberately brief because you really should know this material.

0.6 Review Questions

The following questions are intended to help you determine if you should continue with your course in Data Structures and Algorithms (DSA) without much worry, or first conduct a serious review of the fundamentals before continuing. If you find that you are struggling on any of the questions, then you should pursue the latter alternative:

1. What are the essential components of a computer system, and what role does each component play?

2. Practice converting numbers among binary, octal, hexadecimal, and decimal systems in any order.

3. How are negative numbers represented in the computer system?

4. How are very large and very small numbers represented in the computer system?

5. What does the acronym SDLC mean?

6. What is an algorithm?

7. What are the main components of an algorithm?

8. What are the main control structures, and how do they work?

9. What is recursion? Give a detailed example of a recursive algorithm. Propose an alternate solution to the problem that uses iteration instead of recursion.

10. Explain what is meant by stepwise refinement. Identify a complex programming problem, and show how through stepwise refinement, you would analyze and/or solve the problem.

11. Identify and briefly describe five programming paradigms. For each paradigm, give two programming languages that fall in that paradigm.

12. Explain how one may use the UML class diagram to assist with algorithm development. Identify a programming problem, and propose an algorithmic solution that involves the use of UML diagram(s).

0.7 Recommended Readings

[Brookshear 2012] Brookshear, J. Glenn, David T. Smith, and Dennis Brylow. 2012. *Computer Science: An Overview* 11th Ed; Boston: Addison-Wesley.

[Liang 2015] Liang, Y. Daniel. 2015. *Introduction to Java Programming –Comprehensive Version* 10h Ed. Boston: Pearson.

[Savitch 2014] Savitch, Walter. 2014. *Java: An Introduction Solving & Programming* 7th Ed. Boston: Addison-Wesley.

[Savitch 2015] Savitch, Walter. 2015.*Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapter 1.