UNIT-2

Topics to be covered in UNIT-2

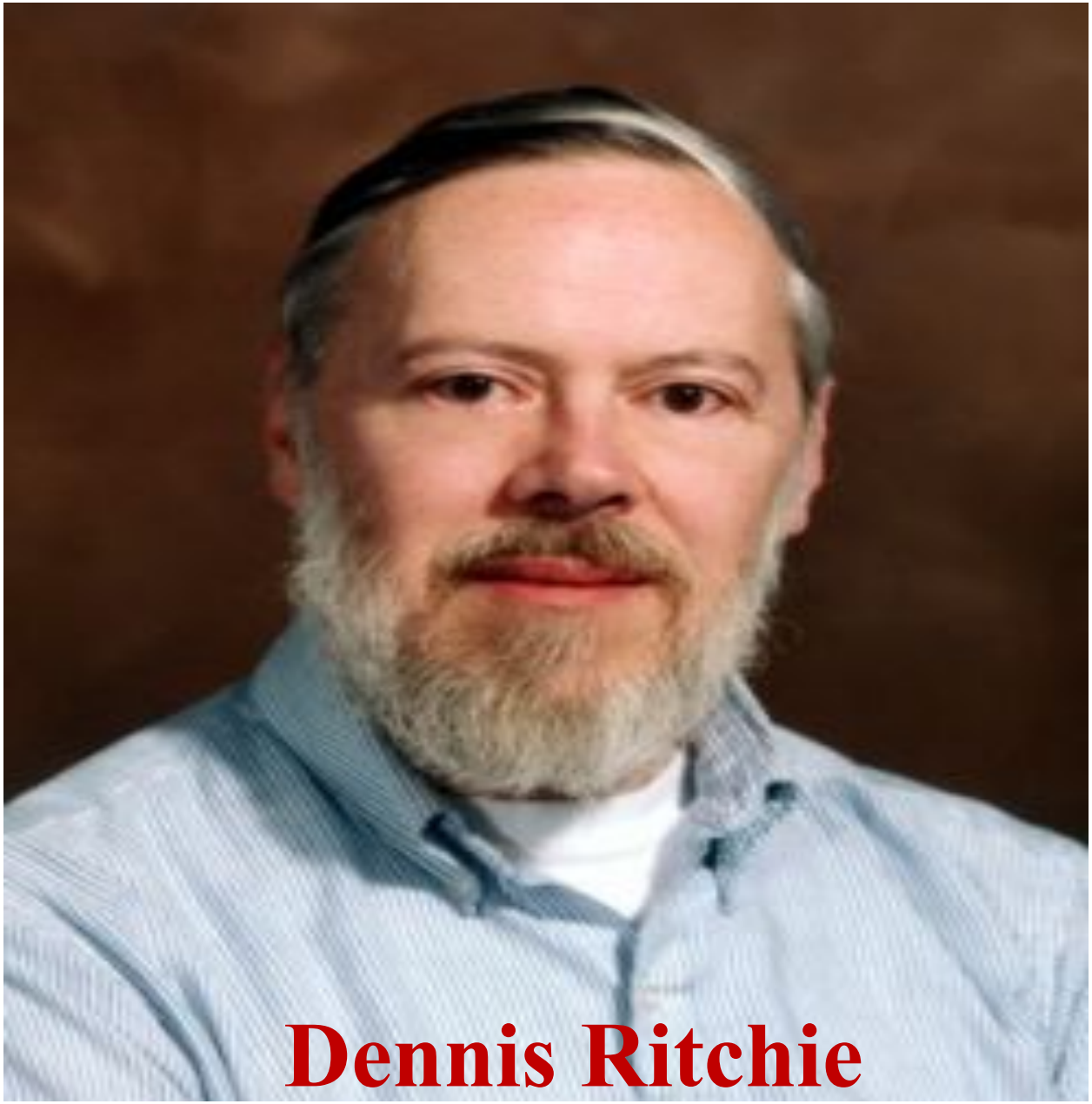
C fundamentals : History of C Language, Features of C, Structure of C, character set, identifiers, constants, variables and keywords.

Simple data types: void, integral, floating-point , memory allocation for these types, type qualifier const.

Operators: Unary, binary and ternary, precedence and association rules among operators.

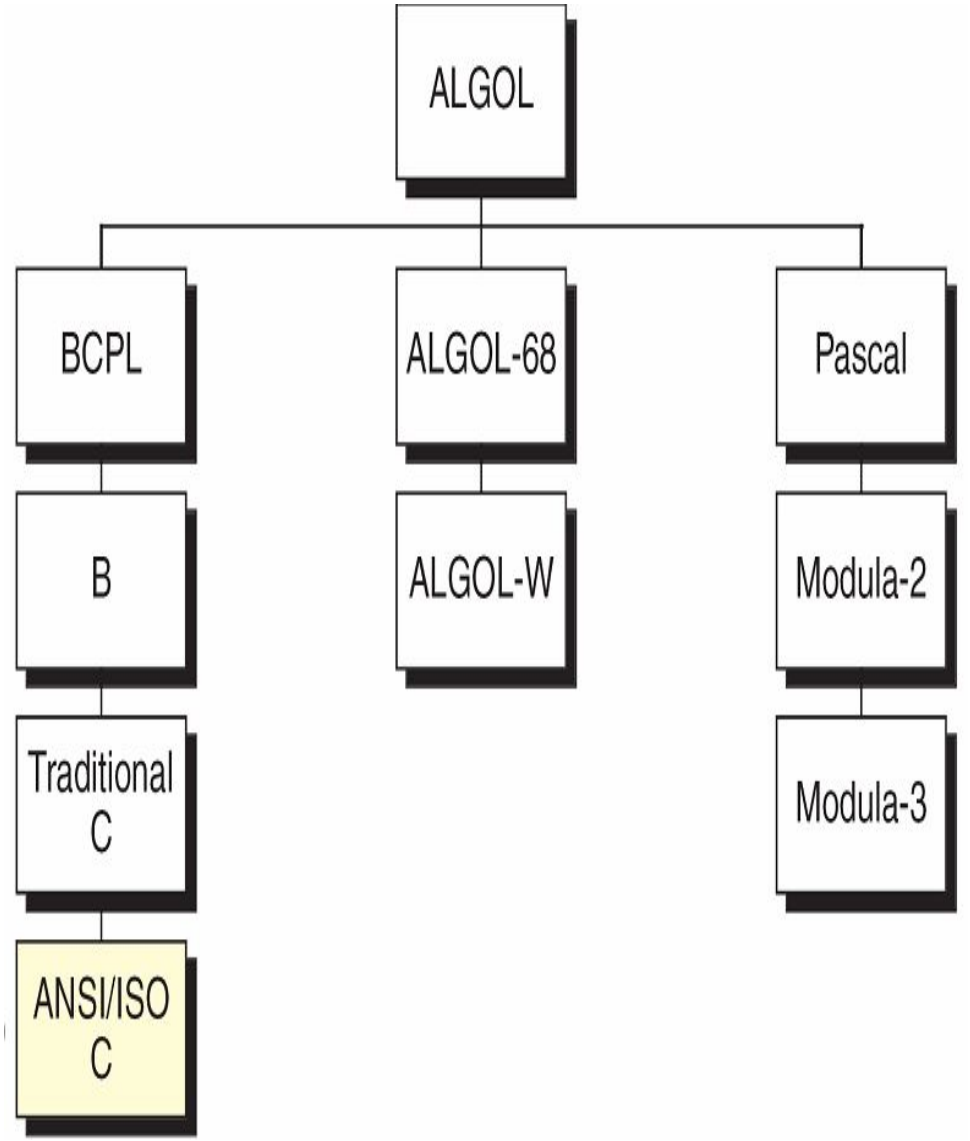
Expressions: Primary expressions, post-fix expressions, pre-fix expressions, unary expressions, binary expressions, evaluating expressions, type conversions, statements.

Decision control structures: if..else, dangling else, switch statement.



Dennis Ritchie

History of C language



Taxonomy of the C Language

History of C language (contd…)

- ALGOL was the first computer language.
- In 1967, **Martin Richards** developed a language called BCPL (Basic Combined Programming Language) at University of Cambridge primarily, for writing system software.
- In 1969, language B was developed by Ken Thompson.
- 'B' was used to create early versions of UNIX operating system at Bell Laboratories.
- In 1972, C was developed by Dennis M. Ritchie at Bell Labs(AT&T) in USA.
- In 1988 C language was standardized by ANSI as ANSI C (ANSI-American National Standards Institute).
- UNIX operating system was coded almost entirely in C.

Features of C language

The increasing popularity of C is due to its various features:

- **Robust:** **C** is a robust language with rich set of built-in functions and operators to write any complex programs.

- **C compilers combines the capabilities of low level languages with features of high level language**. Therefore it is suitable for writing the system software, application software and most of the compilers of other languages also developed with **C** language.

- **Efficient and Fast:** Programs written in **C** are efficient and fast. This is due to its variety of data types.

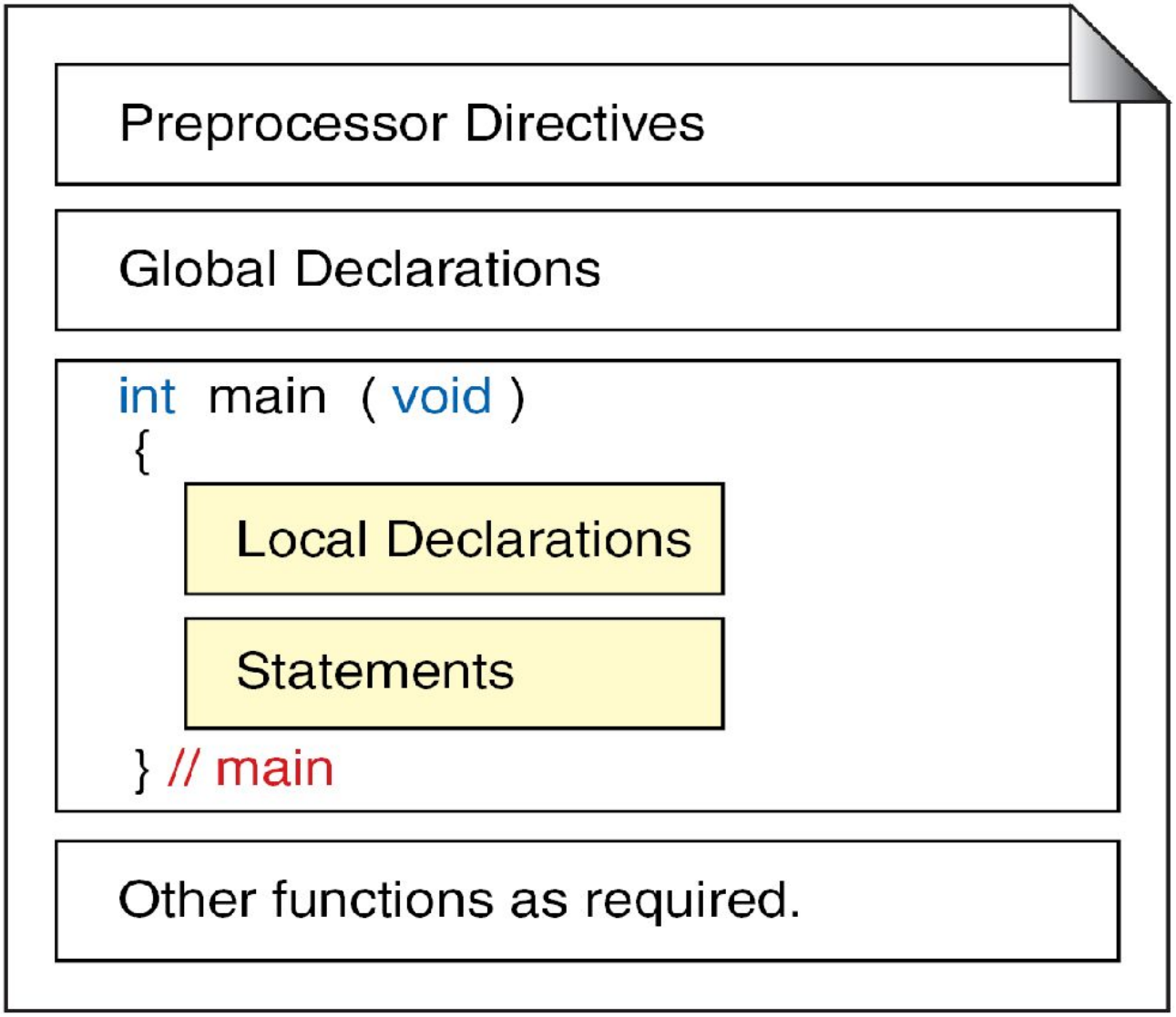
- **Portable:** **C** program written on one computer can also run on the other computer with small or no modification.
  Example: **C** program written in windows can also run on the Linux operating system.

Features of C language (contd…)

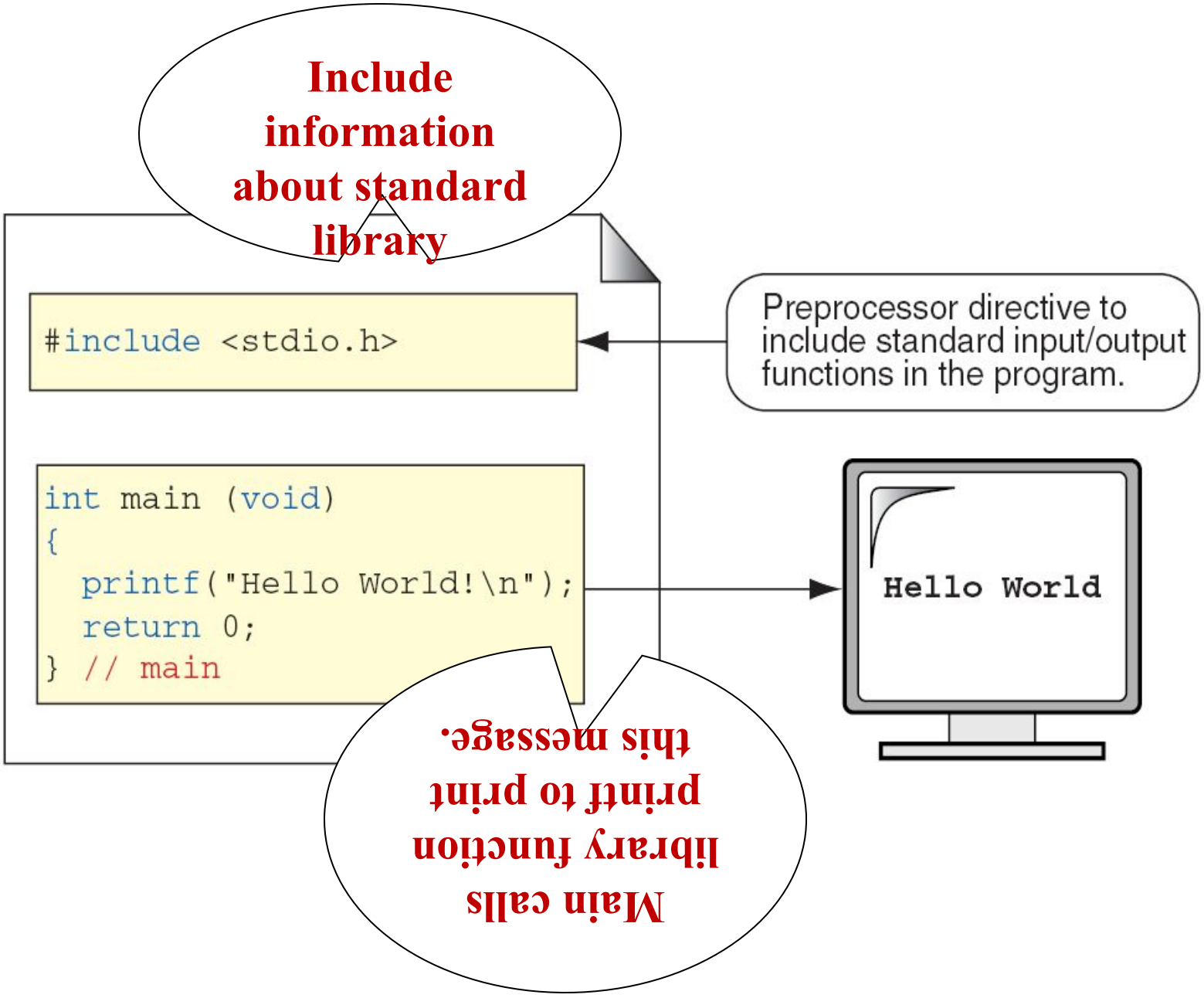
- **Structured Programming:** Every program in C language is divided into small modules or functions so that it makes the program much simple, debugging, and also maintenance of the program is easy.

- **Ability to extend itself:** A C program is basically a collection of various functions supported by C library (also known as header files). We can also add our **own functions** to the **C library**. These functions can be reused in other applications or programs.

Structure of C Program



General structure of a C program


Include information about standard library
#include <stdio.h>
Preprocessor directive to include standard input/output functions in the program.
int main (void)
{
printf("Hello World!\n");
return 0;
} // main
Hello World
Main calls library function printf to print this message.

Preprocessor Directives

- The **preprocessor directives** provide instructions to the preprocessor, to include functions from the system library, to define the symbolic constants and macro.
- The preprocessor command always starts with symbol **#.**
- Example: #include<stdio.h>
- **Header file** contains a collection of library files.
- #include<stdio.h> includes the information about the standard input/output library.

- The variables that are used in common by more than one function are called **Global Variables** and are declared in global declaration section.
- Every C program must have one **main()** function. All the statements of main are enclosed in braces.
- The program execution begins at main() function and ends at closing brace of the main function.
- C program can have any number of **user-defined functions** and they are generally placed immediately after the main () function, although they may appear in any order.

- All sections except the main () function may be absent when they are not required.
- In the previous program, main() function **returns** an integer value to the **operating system**.
- Each statement in C program must end with **;** specifies that the instruction is ended.
- A function can be called by **it's name**, followed by a parenthesized list of arguments and ended with semicolon**.**
- In previous program main() function calls printf() function.
  Example: **printf**("Hello World!\n")**;**

Comments

- To make the program more readable use the comments.
- They may used to make the program easier to understand.
- Two types of comments
    1. Block comment
    2. Line comment.

Block Comment

```
/* Write a program to add two integer numbers */
#include<stdio.h>
main()
{
    int a=10,b=20,c;
    c=a+b;
    printf("Sum of a and b=%d",c);
    return 0;
}
```

- Any characters between /* and */ are ignored by the compiler.
- Comments may appear anywhere in a program.
- /* and */ is used to comment the multiple lines of code which is ignored by the compiler.
- Nested block comments are invalid like /* /* */

Line Comment

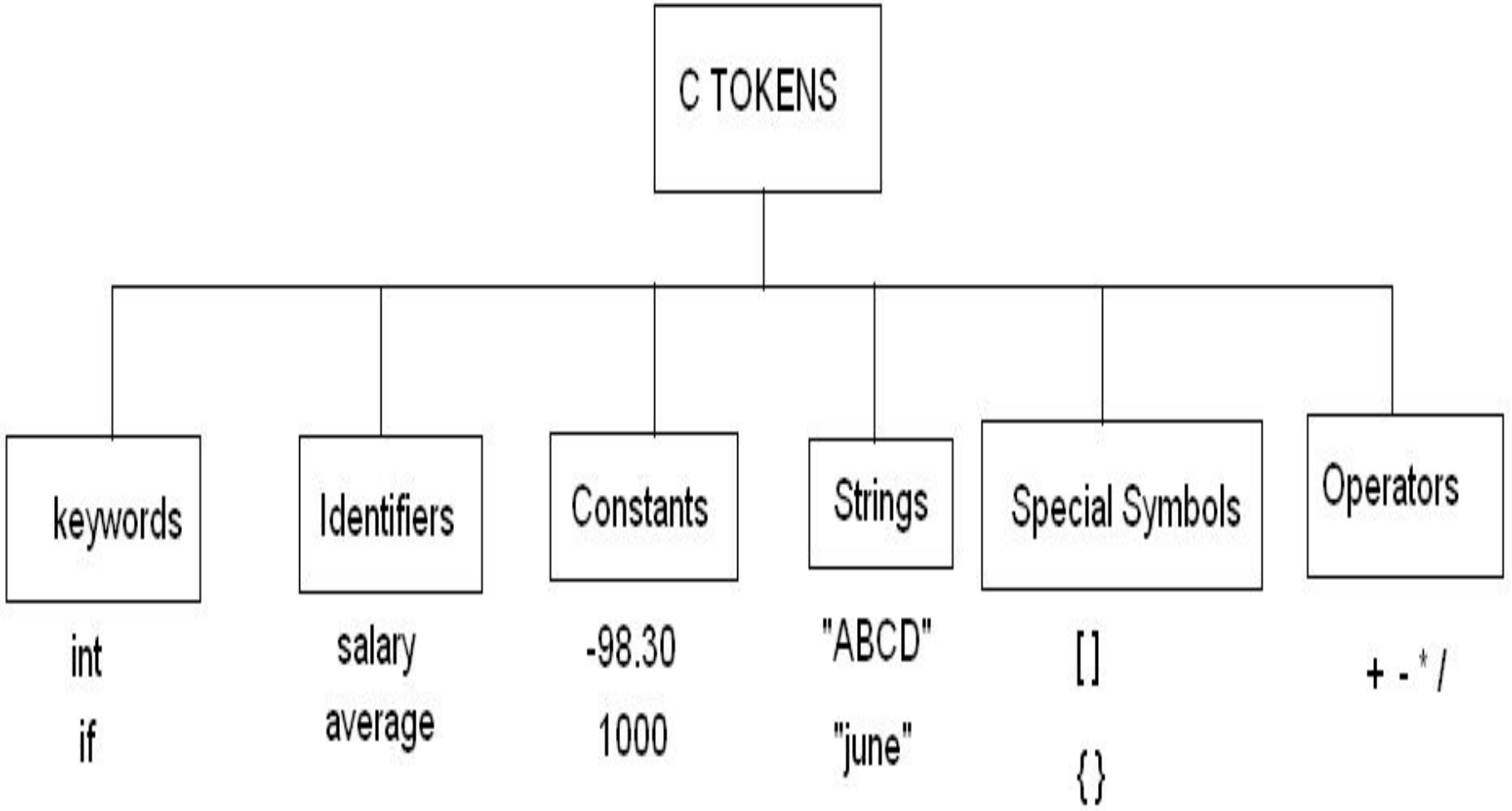
- To comment a single line use two slashes //

```
/* Write a program to add two integer numbers */
#include<stdio.h>        // It includes input, output header file
main()
{
    int a=10,b=20,c;     // Variables declaration & initialization
    c=a+b;               // Adding two numbers
    printf("Sum of a and b=%d",c);
    return 0;
}
```

C Tokens:

In a passage of text, individual words and punctuation marks are called as tokens.

The compiler splits the program into individual units, are known as C tokens. C has six types of tokens.



Character Set

- Characters are used to form words, numbers and expressions.
- Characters are categorized as
  - Letters
  - Digits
  - Special characters
  - White spaces.

Letters: (Upper Case and Lower Case)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

Digits: 0 1 2 3 4 5 6 7 8 9

Special Characters:

’ " ( ) * + - / : = ! & $; < > % ? , . ^ # @ ~ ‘ { } [ ] \ |

White Spaces: Blank Space, Horizontal Space, Carriage Return, New Line.

Identifiers

- Identifiers are **names** given to various programming elements such as variables, constants, and functions.
- It should start with an alphabet or underscore, followed by the combinations of alphabets and digits.
- No special character is allowed except underscore.
- An Identifier can be of arbitrarily long. Some implementation of C recognizes only the first 8 characters and some other recognize first 32 Characters.

The following are the rules for writing identifiers in C:

- First character must be alphabetic character or underscore.
- Must consist only of alphabetic characters, digits, or underscore.
- Should not contain any special character, or white spaces.
- Should not be C keywords.
- Case matters (that is, upper and lowercase letters). Thus, the names **count** and **Count** refer to two different identifiers.

Identifier	Legality
Percent	Legal
y2x5__fg7h	Legal
annual profit	Illegal: Contains White space
_1990_tax	Legal but not advised
savings#account	Illegal: Contains the illegal character #
double	Illegal: It is s a C keyword
9winter	Illegal: First character is a digit

Examples of legal and illegal C identifiers

Variables

- Variable is a valid identifier which is used to store the value in the memory location, that value varies during the program execution.

Types of variables:

  - Global Variables
  - Local Variables

- **Global Variable:** The variables which are declared at the starting of the program are called as global variable. They are visible to all the parts of the program.

- **Local Variable:** The variables which are declared in a function are called local variables to that function. These variables visible only within the function.

Variables (contd…)

Variable Declaration & Definition:

- Each variable in your program must be declared and defined.
- In C, a declaration is used to name an object, such as a variable.
- Definitions are used to create the object.
- When you create variables, the declaration gives them a symbolic name and the definition reserves memory for them.
- A variable's type can be any of the data types, such as character, integer or real except void.
- C allows multiple variables of the same type to be defined in one statement.

  Example: int a, b;

Variable Initialization:

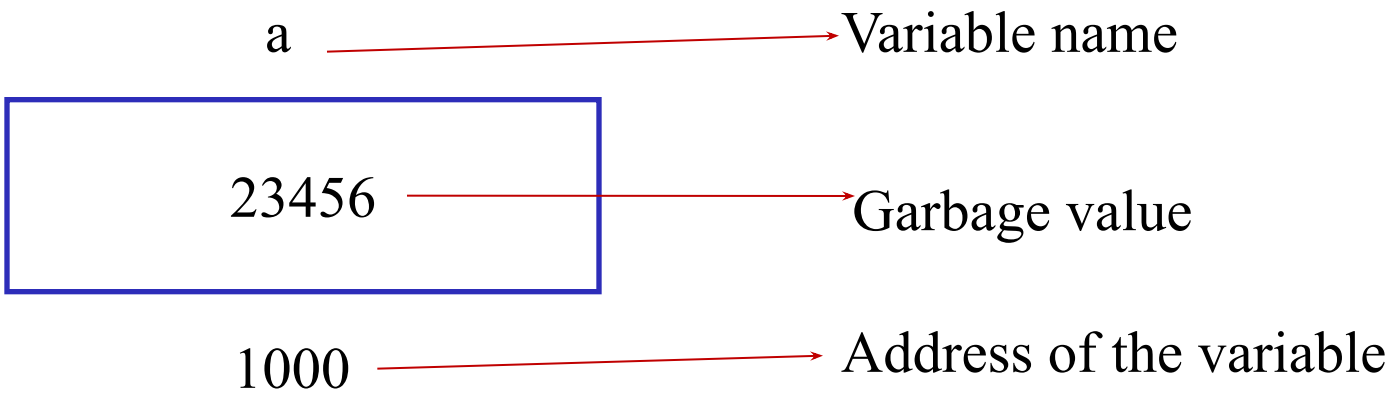
- You can initialize a variable at the same time that you declare it by including an initializer.
- To initialize a variable when it is defined, the identifier is followed by the assignment operator and then the initializer.

  Example: int count = 0;

Variables (contd…)

Variable declaration and definition:

Example: int a;



Variable initialization:

datatype identifier = initial value;

Examples:

```
int a=10;
float b=2.1;
float pi=3.14;
char ch='A';
```

Variables (contd…)

- When you want to process some information, you can save the values temporarily in variables.

```
#include<stdio.h>
main()
{
      int a=10,b=20,c;
      c=a+b;
      printf("sum of a and b=%d\n",c);
      return 0;
}
```

- **Note** : Variables must be declared before they are used, usually at the beginning of the function.

Variables (contd…)

There are some restrictions on the variable names (same as identifiers):

- First character must be alphabetic character or underscore.
- Must consist only of alphabetic characters, digits, or underscore.
- Should not contain any special character, or white spaces.
- Should not be C keywords.
- Case matters (that is, upper and lowercase letters). Thus, the names **count** and **Count** refer to two different identifiers.

- The following list contains some examples of legal and illegal C variable names:

Variable Name	Legality
Percent	Legal
y2x5__fg7h	Legal
annual_profit	Legal
_1990_tax	Legal but not advised
savings#account	Illegal: Contains the illegal character #
double	Illegal: It is a C keyword
9winter	Illegal: First character is a digit

Constants

- Constants are data values that cannot be changed during the program execution.
- Like variables, constants have a type.

Types of constants:

- Boolean constants: A Boolean data type can take only two values **true** and **false**.
- Character constants
  - Single character constants
  - string constants.
- Numeric constants.
  - integer constant
  - real constants.

Type qualifier const

- One way to use the constant is with memory constants. Memory constants use a C type qualifier; **const**.
- This indicates that the data cannot be changed.

const type identifier= value;

const float pi=3.14;

```
#include<stdio.h>
void main()
{
    float area,radius=3.0;
    const float pi=3.14;
    area=pi*radius*radius;
    printf("area of a circle= %f",area);
}
```

Single character constants

- A **single character constants** are enclosed in **single quotes**.

  Example: ‘1’ ‘X’ ‘%’ ‘ ‘

- Character constants have integer values called **ASCII** values.

```
char ch=‘A’;
printf(“%d”,ch);                    Output:  65
similarly printf(“%c”,65)           Output:  A
```

string Constants

- **String** is a collection of characters or sequence of characters enclosed in **double quotes**.
- The characters may be letters, numbers, special characters and blank space.

  Example: “snist” “2011” “A”.

Backslash \escape characters

- **Backslash characters** are used in **output** functions.
- These backslash characters are preceded with the \ symbol.

constant	meaning
'\a'	Alert(bell)
'\b'	Back space
'\f'	Form feed
'\n'	New line
'\r'	Carriage return
'\v'	Vertical tab
'\t'	Horizontal tab
'\''	Single quote
'\"'	Double quotes
'\?'	Question mark
'\\'	Backslash
'\0'	null

Numeric Constants

integer constant: It is a sequence of digits that consists numbers from 0to 9.

Example: 23 -678 0 +78

Rules:

1. integer constant have at least one digit.
2. No decimal points.
3. No commas or blanks are allowed.
4. The allowable range for integer constant is **-32768** to **32767.**

To store the larger integer constants on 16 bit machine use the qualifiers such as U,L,UL.

Type	Representation	Value
int	+245	245
int	-678	-678
unsigned int	65342u / 65342U	65342
unsigned long int	99999UL	99999
long int	999999L	999999

Real constants: The numbers containing fractional parts like 3.14
Example: 1.9099 -0.89 +3.14

Real constants are also expressed in exponential notation.

Mantissa e exponent

- A number is written as the combination of the mantissa, which is followed by the prefix **e** or **E**, and the exponent.

  Example:

87000000	=	8.7e7
- 550	=	-5.5e2
0.00000000031	=	3.1e-10.

Examples of real constants

Type	Representation	Value
double	0.	0.0
double	0.0	.0
float	-2.0f	-2.0
long double	3.14159276544L	3.14159276544

```
#include<stdio.h>
void main()
{
    const int a=10;
    int b=20;
    a=a+1;
    b=b+1;
    printf("a=%d  b=%d",a,b);
}
```

Coding Constants: Different ways to create constants.

Literal constants:

A literal is an unnamed constant used to specify data.

Example: **a = b + 5;**

Defined constants:

By using the preprocessor command you can create a constant.

Example: **#define pi 3.14**

Memory constants:

Memory constants use a C type qualifier, const, to indicate that the data can not be changed.

Its format is: const type identifier = value;

Example: const float PI = 3.14159;

Key Words

- C word is classified as either **keywords** or **identifiers**.
- Keywords have fixed meanings, these meanings cannot be changed.
- Keywords must be in **lowercase**.

Key Words in C

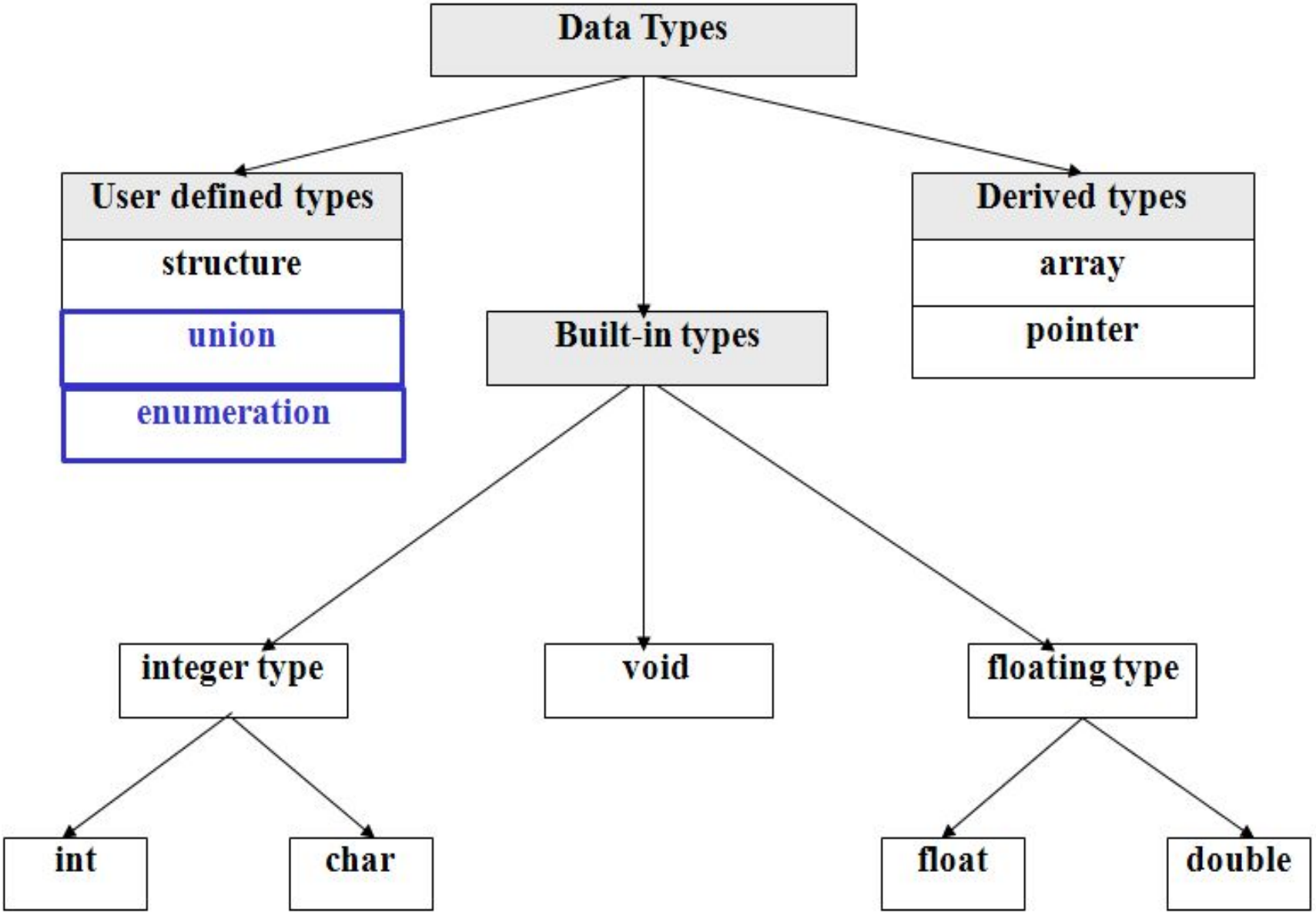
auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

Data types in C

- Data types are used to indicate the **type** of value represented or stored in a variable, the **number of bytes** to be reserved in memory, the **range of values** that can be represented in memory, and the **type of operation** that can be performed on a particular data value.

- ANSI C supports 3 categories of data types:
  - Built-in data types
  - Derived data types
  - User Defined data types

Data types in C (contd…)



Built-in data types:

Built-in data types are also known as primitive data types. C uses the following primitive data types.

int	integer quantity
char	character (stores a single character)
float	floating point number
double	floating point number

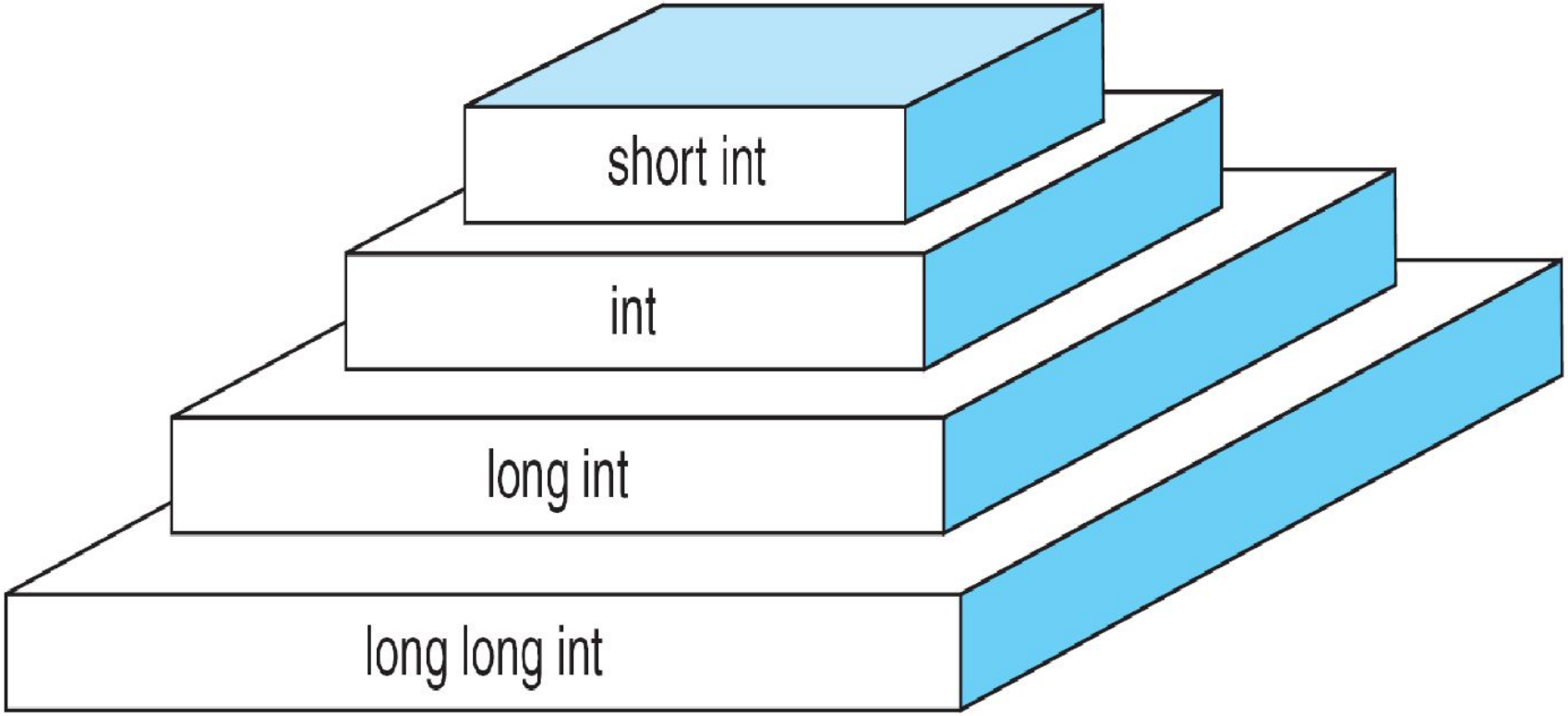
Built-in data types (contd…)

Integer data type :

- An **integer number** (also called whole number) has no fractional part or decimal point.
- The keyword **int** is used to specify an integer variable.
- It occupies 2 bytes (16 bits) or 4 bytes (32 bits), depending on the machine architecture.
- 16-bit integer can have values in the range of **-32768** to **32767**
- One bit is used for sign.

void data type:

- Defines an empty data type which can then be associated with some data types. It is useful with pointers.



Note

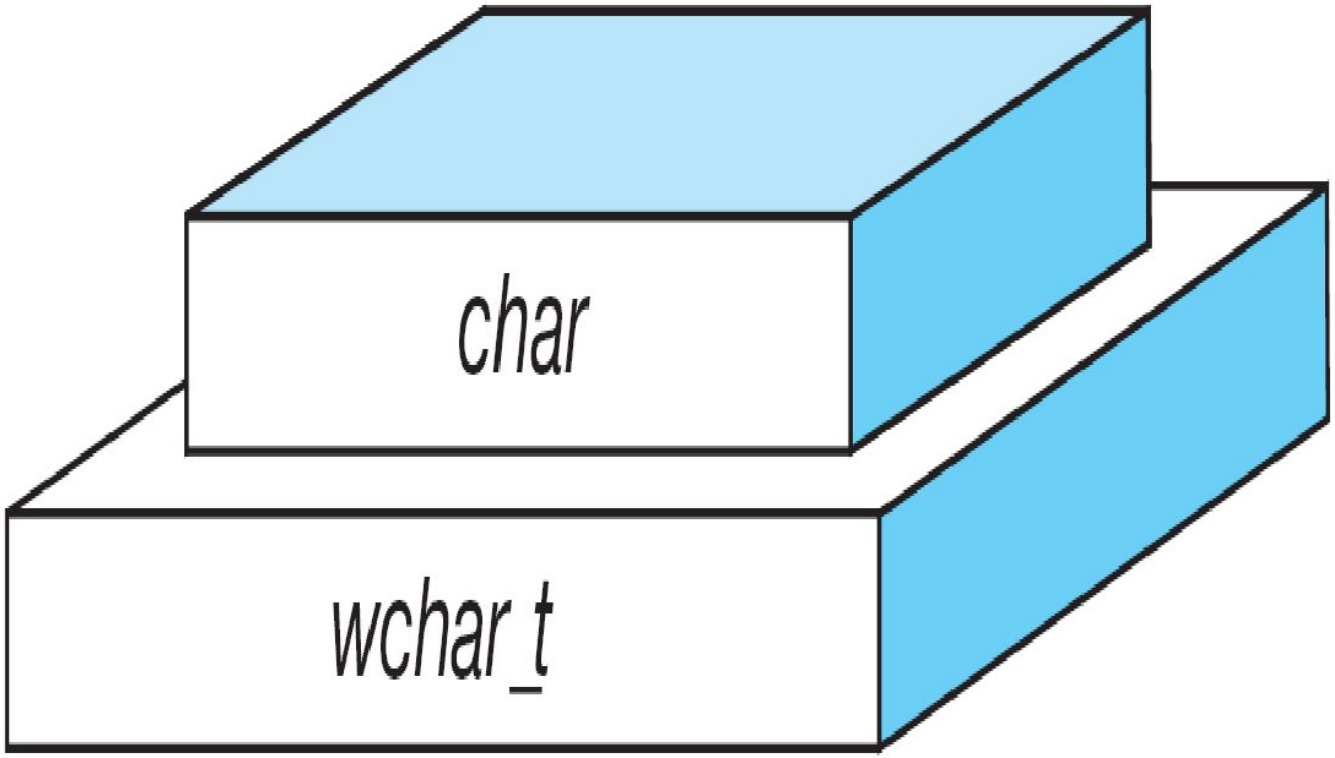
sizeof (short) ≤ sizeof (int) ≤ sizeof (long) ≤ sizeof (long long)

Integer Types

Built-in data types (contd…)

Character Data Type :

- The shortest data type is character.
- The keyword char is used to declare a variable of a character type.
- It is stored in 1 byte in memory.
- Corresponding integer values for all characters are defined in ASCII (American Standard Code for Information Interchange).
- **Example:** character constant ‘a’ has an int value 97,
  ‘b’ has 98, ‘A’ has 65 etc.
- Character can have values in the range of **-128** to **127**.



Character Types

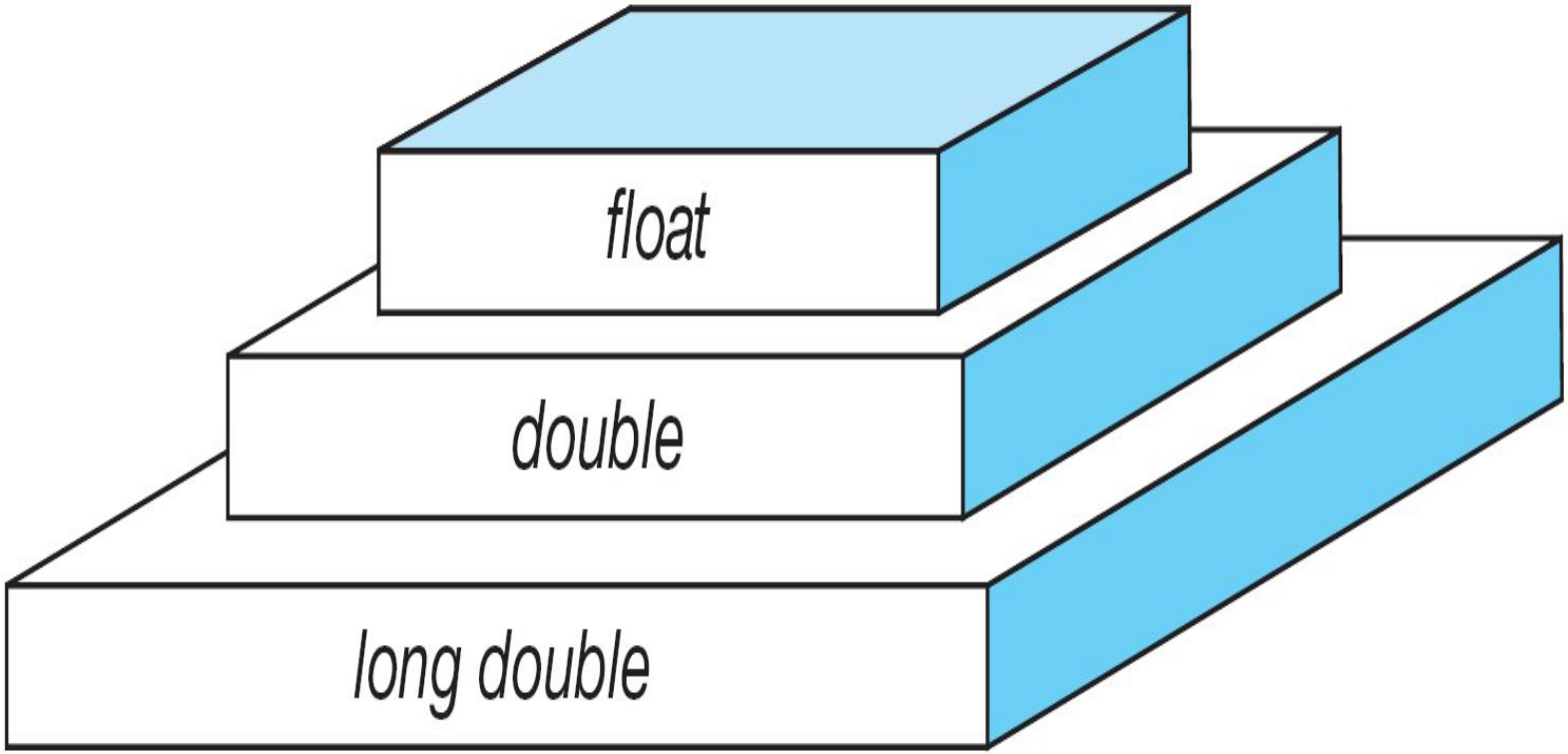
Built-in data types (contd…)

The floating point data type:

- The keyword **float** is used to declare a variable of the type float.
- The float type variable is usually stored in 32 bits, with 6 digits of precision.
- A float variable can have values in the range of **3.4E-38** to **3.4 E+38**.

The double data type:

- A floating point number can also be represented by the double data type.
- The data type double is stored on most machines in 64 bits which is about 15 decimal places of accuracy.
- To declare a variable of the type double, use the keyword **double**.
- A double variable can have values in the range of **1.7E-308** to **+1.7E+308**.



Note

sizeof (float) ≤ sizeof (double) ≤ sizeof (long double)

Floating-point Types

Type Modifiers:

- The basic data types may have various **modifiers** (or **qualifiers**) preceding them, except type ‘void’.
- A **modifier** is used to alter the meaning of the base data type to fit the needs of various situations more precisely.
- The modifiers **signed**, **unsigned**, **long**, **short** may be applied to **integer** base types.
- The modifiers **unsigned and signed** may be applied to **characters**.
- The modifier **long** may also be applied to **double**.
- The difference between signed and unsigned integers is in the way high-order bit (sign bit) of the integer is interpreted.
- If sign bit is 0, then the number is positive; if it is 1, then the number is negative.

Derived data types and User defined data types are the combination of primitive data types. They are used to represent a collection of data.

They are:

- Arrays
- Pointers
- Structures
- Unions
- Enumeration

Note: Number of bytes and range given to each data type is platform dependent.

Data types in C

type	size (Byte)	smallest number	largest number	precision	constant	format specifier
long double	10	3.4E-4932	1.1E+4932	1.08E-19	1.2345L, 1.234E-5L	%Lf, %Le, %Lg
double	8	1.7E-308	1.7E+308	2.22E-16	1.2345, 1.234E-5	%lf, %le, %lg
float	4	1.7E-38	3.4E+38	1.19E-7	1.2345F, 1.234E-5F	%f, %e, %g
unsigned long	4	0	4294967295		123UL	%lu
long	4	-2147483648	2147483647		123L	%ld, %li
unsigned	4	0	4294967295		123U	%u
int	4	-2147483648	2147483647		123	%d, %i
unsigned short	2	0	65535		123U	%hu
short	2	-32768	32767		123	%hd, %hi
unsigned char	1	0	255		'a', 123, '\n'	%c
char	1	-128 or 0	127 or 255		'a', 123, '\n'	%c

A Program That Prints "Nothing!"

```
1  /* Prints the message "Nothing!".
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9  // Statements
10    printf("This program prints\n\n\t\"Nothing!\"");
11    return 0;
12 }  // main
```

```
Results:
This program prints

    "Nothing!"
```

Demonstrate Printing Boolean Constants

```c
/* Demonstrate printing Boolean constants.
      Written by:
      Date:
*/
#include <stdio.h>
#include <stdbool.h>

int main (void)
{
// Local Declarations
   bool x = true;
   bool y = false;

// Statements
   printf ("The Boolean values are: %d %d\n", x, y);
   return 0;
}  // main
```

```
Results:
The Boolean values are: 1 0
```

Print Value of Selected Characters

```
 1  /* Display the decimal value of selected characters,
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     char A          = 'A';
11     char a          = 'a';
12     char B          = 'B';
13     char b          = 'b';
14     char Zed        = 'Z';
15     char zed        = 'z';
```

Print Value of Selected Characters (continued)

```
16      char zero      = '0';
17      char eight     = '8';
18      char NL        = '\n';       // newline
19      char HT        = '\t';       // horizontal tab
20      char VT        = '\v';       // vertical tab
21      char SP        = ' ';        // blank or space
22      char BEL       = '\a';       // alert (bell)
23      char dblQuote  = '"';        // double quote
24      char backSlash = '\\';       // backslash itself
25      char oneQuote  = '\'';       // single quote itself
26
27  // Statements
28      printf("ASCII for char 'A'  is: %d\n",  A);
29      printf("ASCII for char 'a'  is: %d\n",  a);
30      printf("ASCII for char 'B'  is: %d\n",  B);
```

Print Value of Selected Characters (continued)

```
31      printf("ASCII for char 'b'  is: %d\n",  b);
32      printf("ASCII for char 'Z'  is: %d\n",  Zed);
33      printf("ASCII for char 'z'  is: %d\n",  zed);
34      printf("ASCII for char '0'  is: %d\n",  zero);
35      printf("ASCII for char '8'  is: %d\n",  eight);
36      printf("ASCII for char '\\n' is: %d\n", NL);
37      printf("ASCII for char '\\t' is: %d\n", HT);
38      printf("ASCII for char '\\v' is: %d\n", VT);
39      printf("ASCII for char ' '  is: %d\n",  SP);
40      printf("ASCII for char '\\a' is: %d\n", BEL);
41      printf("ASCII for char '\"'  is: %d\n", dblQuote);
42      printf("ASCII for char '\\'  is: %d\n", backSlash);
43      printf("ASCII for char '\''  is: %d\n", oneQuote);
44
45      return 0;
46  }   // main
```

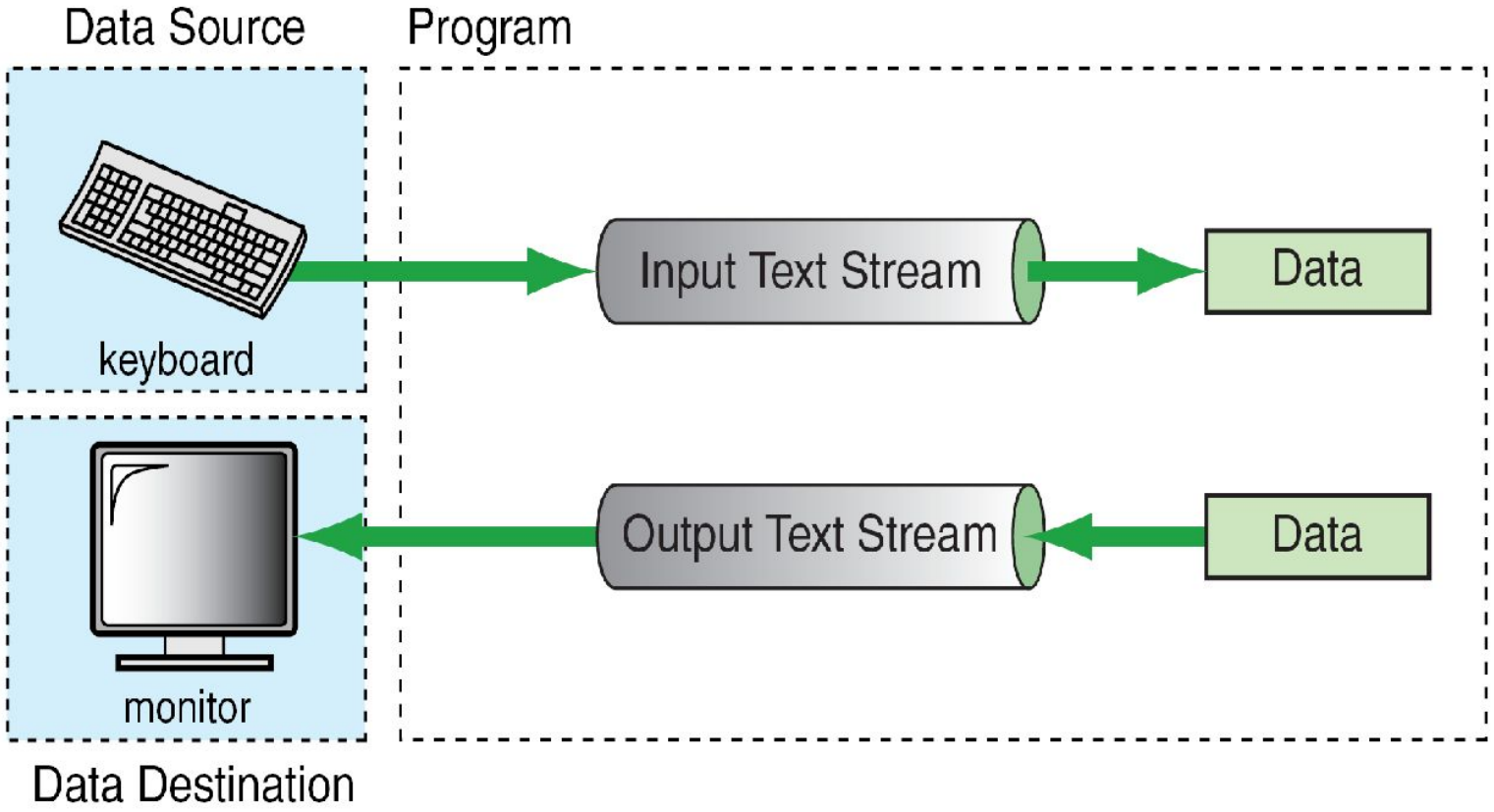
Print Value of Selected Characters (continued)

```
Results:
ASCII for character 'A'  is: 65
ASCII for character 'a'  is: 97
ASCII for character 'B'  is: 66
ASCII for character 'b'  is: 98
ASCII for character 'Z'  is: 90
ASCII for character 'z'  is: 122
ASCII for character '0'  is: 48
ASCII for character '8'  is: 56
ASCII for character '\n' is: 10
ASCII for character '\t' is: 9
ASCII for character '\v' is: 11
ASCII for character ' '  is: 32
ASCII for character '\a' is: 7
ASCII for character '"'  is: 34
ASCII for character '\'  is: 92
ASCII for character '''  is: 39
```

Formatted input/output

- The data is to be arranged in a particular format.
- The data is input to and output from a stream.
- A stream is a source or destination of the data, it is associated with a physical device such as terminals (keyboard, monitor).
- C has two forms of streams: **Text Stream** and **Binary Stream**.
- **Text Stream** consists of sequence of characters.
- **Binary Stream** consists of a sequence of data in 0's and 1's.

- A terminal **keyboard** and **monitor** can be associated only with a **text stream**.
- A keyboard is a **source** for a text stream; a monitor is a **destination** for a text stream.

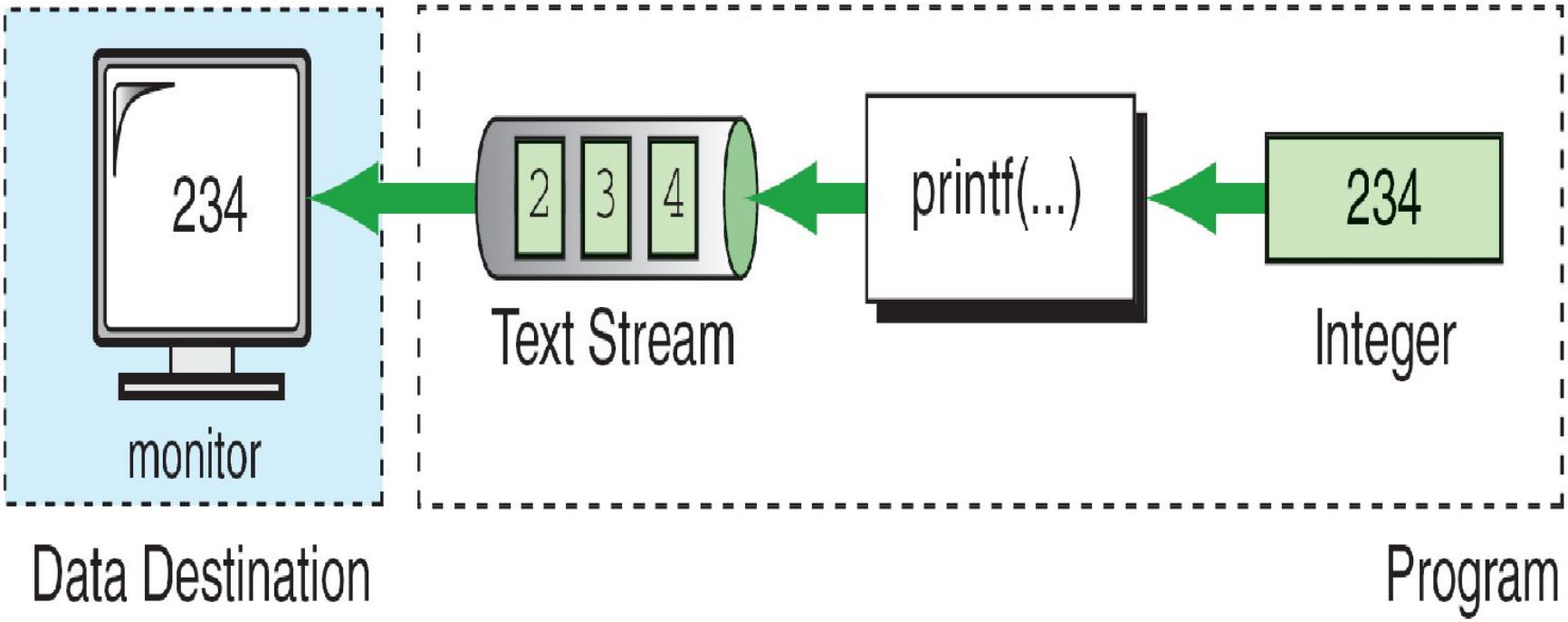


Stream Physical Devices

- The data is formatted using the printf and scanf functions.
- scanf() converts the text stream coming from the keyboard to data values (char, int, etc.,) and stores them in the program variables.
- printf() converts the data stored in variables into the text stream for output the keyboard.

- **printf()** takes the set of data values and converts them to text stream using formatting instructions contained in a **format control string**.

- **Format control specifiers** are used to format the text stream.



Output Formatting Concept

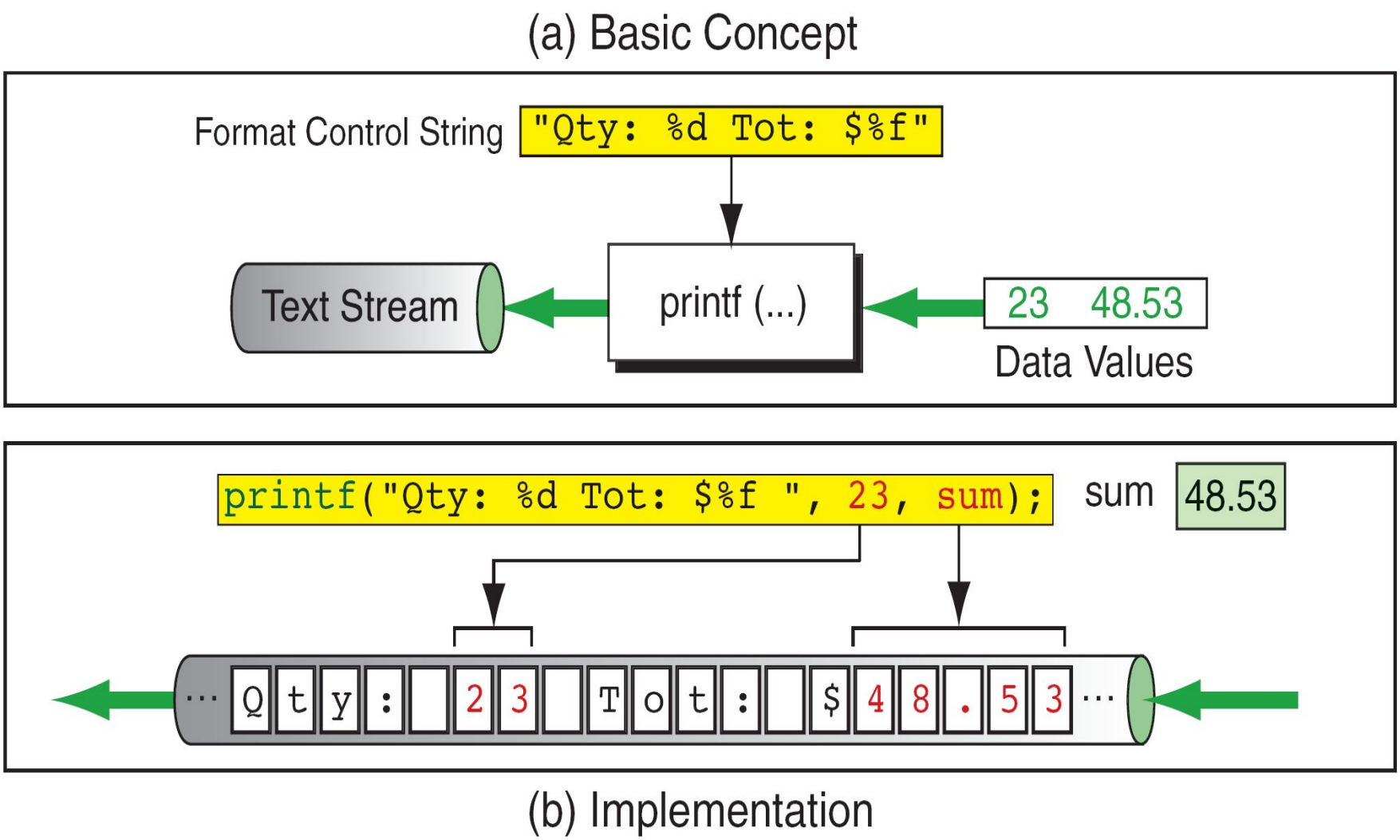
- Format specifier specifies the data values type, size and display position.

- Printf statement takes two arguments
  1. Control String and
  2. Data Values(Variables)

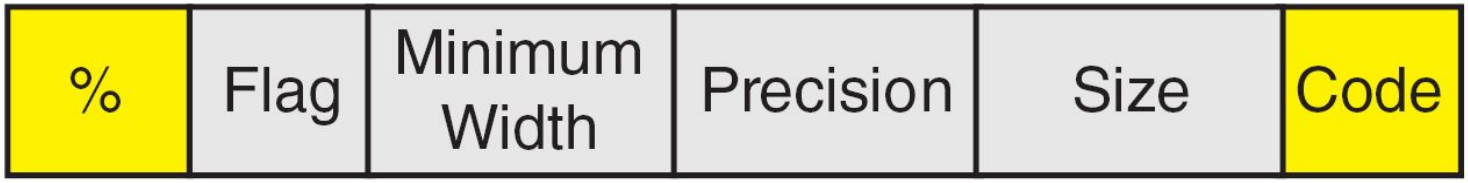
- Control string contains the format specifiers and some text.
  Syntax: printf("control string",var1,var2,…,varn);

  Example:

```
int a=10, b=20;
printf("a=%d  b=%d", a,b);
```



Output Stream Formatting Example



Conversion Specification

Flag Type	Flag Code	Formatting
Justification	None	right justified
	–	left justified
Padding	None	space padding
	0	zero padding
Sign	None	positive value: no sign negative value: –
	+	positive value: + negative value: –
	Space	positive value: space negative value: –

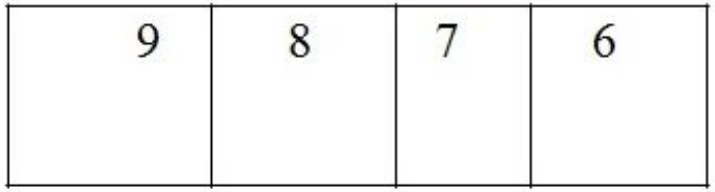
Flag Formatting Options

Type	Size	Code	Example
char	None	c	%c
short int	h	d	%hd
int	None	d	%d
long int	None	d	%ld
long long int	ll	d	%lld
float	None	f	%f
double	None	f	%f
long double	L	f	%Lf

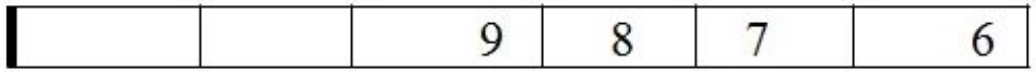
Format Codes for Output

Format

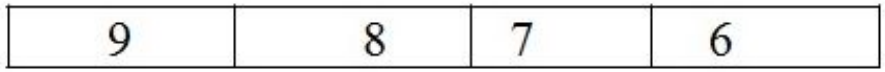
printf(“%d”,9876);



printf(“%6d”,9876);

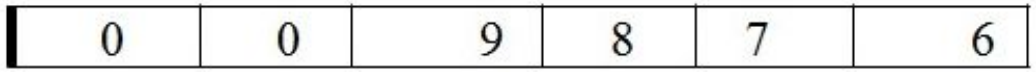


printf(“%2d”,9876);



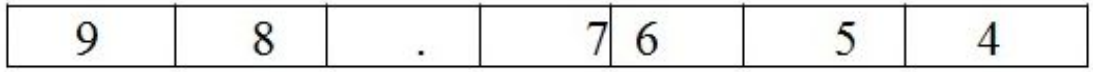
printf(“%-6d”,9876);

printf(“%06d”,9876);

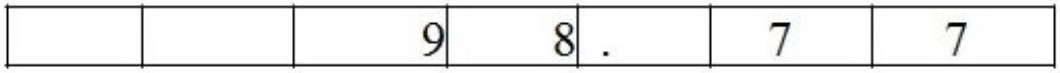


int y=98.7654;

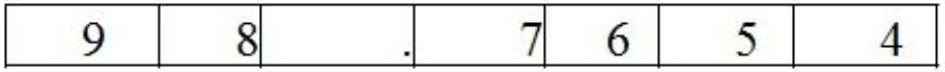
printf(“%7.4f”,y);



printf(“%7.2f”,y);



printf(“%f”,y);



Output Examples

1.
```
printf("%d%c%f", 23, 'z', 4.1);
```
```
23z4.100000
```

2.
```
printf("%d %c %f", 23, 'z', 4.1);
```
```
23 z 4.100000
```

3.
```
int num1 = 23;
char zee = 'z';
float num2 = 4.1;
printf("%d %c %f", num1, zee, num2);
```
```
23 z 4.100000
```

4.
```
printf("%d\t%c\t%5.1f\n", 23, 'z', 14.2);
printf("%d\t%c\t%5.1f\n", 107, 'A', 53.6);
printf("%d\t%c\t%5.1f\n", 1754, 'F', 122.0);
printf("%d\t%c\t%5.1f\n", 3, 'P', 0.1);
```

```
23      z        14.2
107     A        53.6
1754    F       122.0
3       P         0.1
```

5.
```
printf("The number%dis my favorite.", 23);
```

```
The number23is my favorite.
```

6.
```
printf("The number is %6d", 23);
```

```
The number is     23
^^^^^^^^^^^^^^^^^^^^
```

7. `printf("The tax is %6.2f this year.", 233.12);`

```
The tax is 233.12 this year.
```

8. `printf("The tax is %8.2f this year.", 233.12);`

```
The tax is   233.12 this year.
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

9. `printf("The tax is %08.2f this year.", 233.12);`

```
The tax is 00233.12 this year.
```

10. `printf("\"%8c   %d\"", 'h', 23);`

```
"       h   23"
^^^^^^^^^^^^^^^^
```

11.
```
printf("This line disappears.\r...A new
line\n");
printf("This is the bell character \a\n");
printf("A null character\0kills the rest of the
line\n");
printf("\nThis is \'it\' in single quotes\n");
printf("This is \"it\" in double quotes\n");
printf("This is \\ the escape character it
self\n");
```

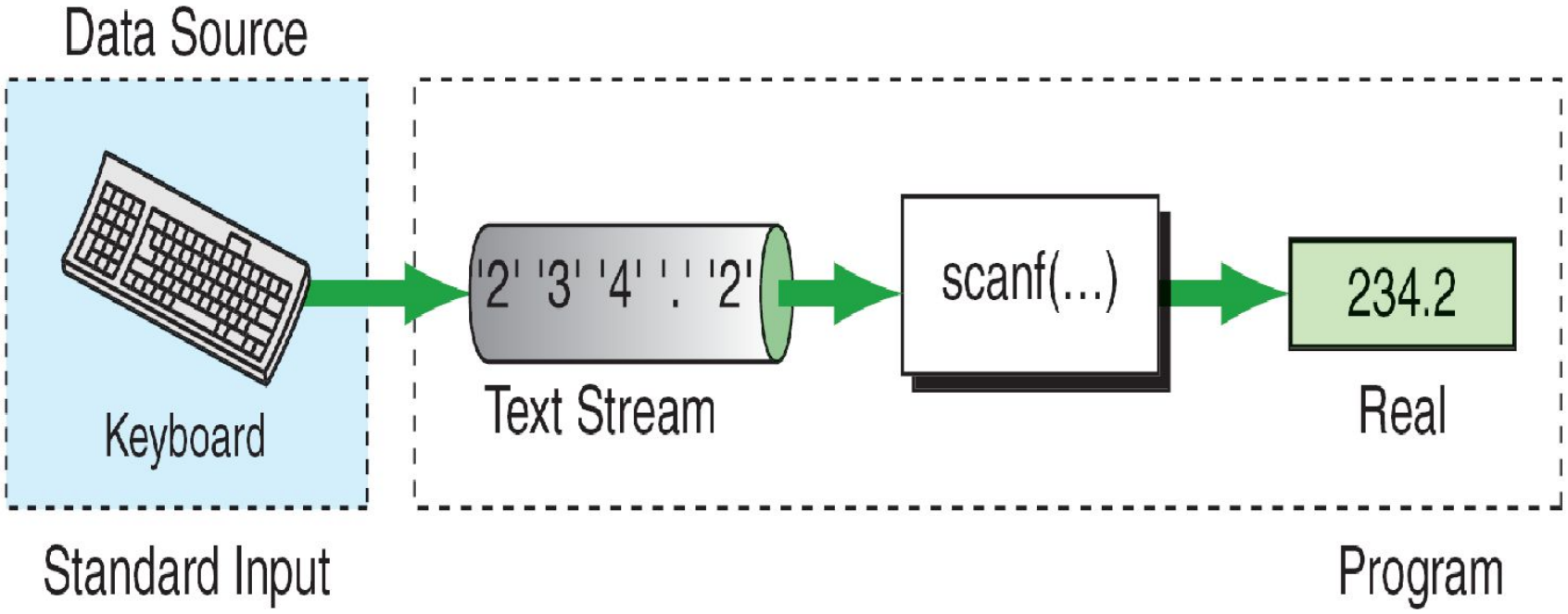
```
...A new line
This is the bell character
A null character
This is 'it' in single quotes
This is "it" in double quotes
This is \ the escape character it self
```

12. `printf("|%-+8.2f| |%0+8.2f| | %-0+8.2f|", 1.2, 2.3, 3.4);`

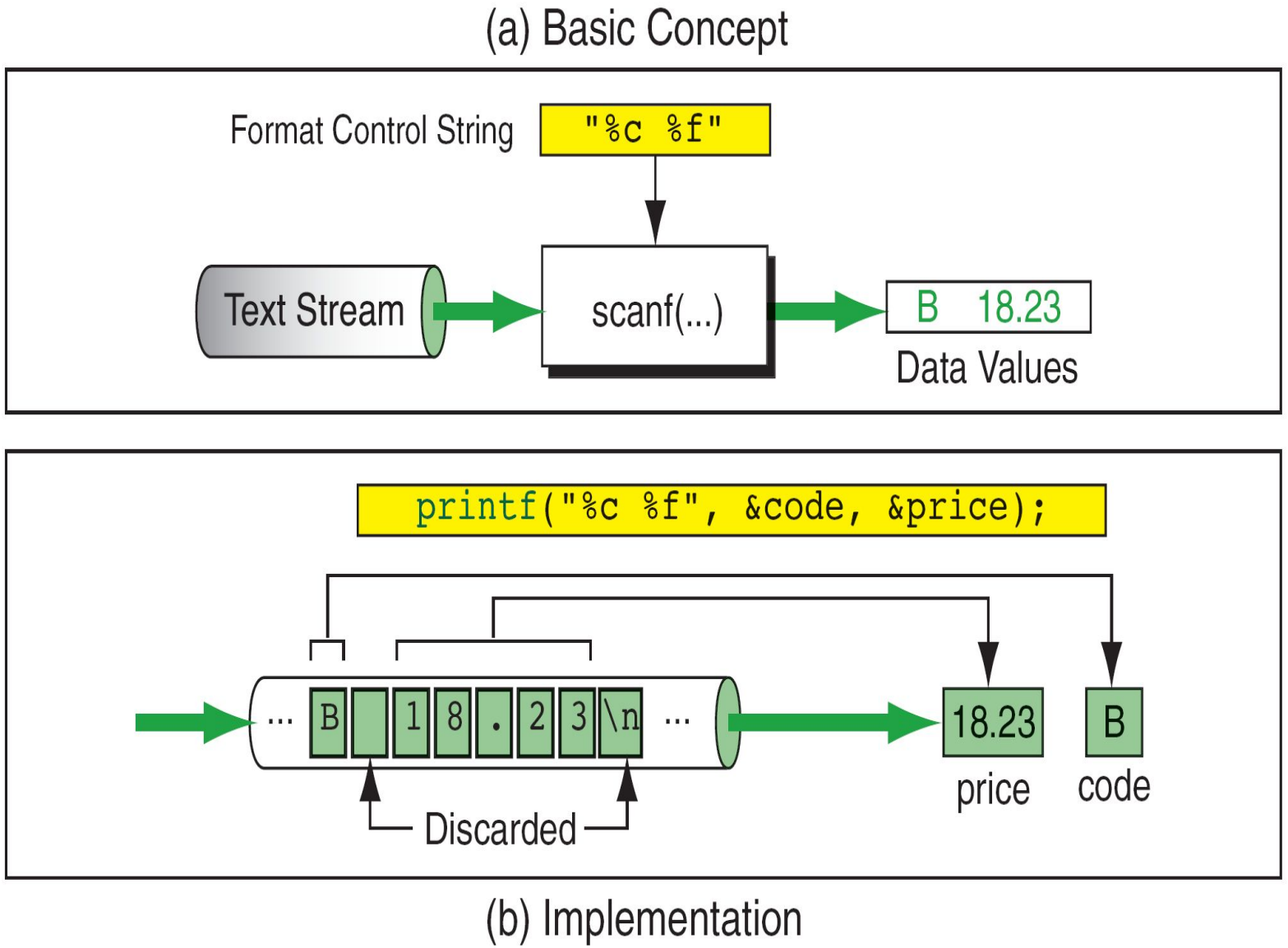
```
|+1.20   | |+0002.30| | +3.40   |
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

scanf("control string",&var1,&var2.. &varn);

- control string includes format specifiers and specifies the field width.
- scanf requires variable addresses in the address list.



Formatting Text from an Input Stream



Input Stream Formatting Example

Conversion Specification

Input Examples

1. 214 156 14z

```
scanf("%d%d%d%c", &a, &b, &c, &d);
```

2. 214 156 14 z

```
scanf("%d%d%d %c", &a, &b, &c, &d);
```

3. 2314 15 2.14

```
scanf("%d %d %f", &a, &b, &c);
```

4. 14/26 25/66

```
scanf("%2d/%2d %2d/%2d", &num1, &den1, &num2,
&den2);
```

5. 11-25-56

```
scanf("%d-%d-%d", &a, &b, &c);
```

Operators

- C supports a rich set of operators.
- An **operator** is a symbol that tells the computer to perform mathematical or logical operations.
- Operators are used in C to **operate on data and variables**.

expression

X=Y+Z

Operators: **=, +** Operands: **x, y, z**

- **Unary operators** are used on a **single operand** (**- -, +, ++, --**)
- **Binary operators** are used to apply in between **two operands** (**+, -, /,*, %**)
- Conditional (or **ternary**) operator can be applied on **three operands**. (**?:**)

Types of Operators

C operators can be classified into a number of categories. They include:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operator
- Increment and Decrement Operators
- Conditional Operators
- Bitwise Operators
- Special Operators

Arithmetic Operators

C Operation	Binary Operator	C Expression
Addition	+	a + b
Subtraction	-	a - b
Multiplication	*	a * b
Division(second operand must be nonzero)	/	a / b
Modulus (Remainder both operands must be integer and second operand must be non zero)	%	a % b

Syntax: **operand1 arithmetic_operator operand2**

Examples:

10 + 10 = 20 **(addition on integer numbers)**
10.0 + 10.0 = 20.0 **(addition on real numbers)**
10 + 10.0 = 20.0 **(mixed mode)**
14 / 3 = 4 **(ignores fractional part)**

Relational Operators

- Relational operators are used to compare the relationship between two operands.

  Syntax: **exp1 relational_operator exp2**

- The value of a relational expression is either one or zero.
- It is **one** if the specified relation is **true** and **zero** if the relation is **false.**
- Relational operators are used by **if , while** and **for** statements.

C operation	Relational Operator	C expression
greater than	>	x > y
less than	<	x < y
greater than or equal to	>=	x >= y
less than or equal to	<=	x <= y
Equality	==	x == y
not equal	!=	x != y

Logical Operators

- Logical operators used to test more than one condition and make decision. Yields a value either one or zero.
- **Syntax:** **operand1 logical_operator operand2** **or** **logical_operator operand**
- **Example:** (x<y) && (x= = 8)

Operator	Meaning
&&	Logical AND (true only if both the operands are true)
||	Logical OR (true if either one operand is true)
!	Logical NOT (negate the operand)

A	B	A && B	A || B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

A	! A
0	1
1	0

Assignment Operators

- Assignment operators are used to assign the result of an expression to a variable.
- Assignment Operator is =

Syntax: **variable = expression;**

- **Types of assignment:**
  - Single Assignment Ex: a = 10;
  - Multiple Assignment Ex: a=b=c=0;
  - Compound Assignment Ex: c = a + b;

Operator	Example	Equivalent Statement
+=	c += 7	c = c + 7
-=	c -= 8	c = c – 8
*=	c *= 10	c = c * 10
/=	c /= 5	c = c / 5
%=	c %= 5	c = c % 5

Increment and Decrement Operators

- We can add or subtract 1 to or from variables by using **increment (++)** and **decrement (--)** operators.
- The operator ++ **adds 1** to the operand and the operator – – **subtracts 1**.
- They can apply in two ways: **postfix** and **prefix.**
- **Syntax:** **increment or decrement_operator operand**
  operand increment or decrement_operator
- **Prefix form:** Variable is changed before expression is evaluated
- **Postfix form:** Variable is changed after expression is evaluated.

Operator	Example	Meaning	Equivalent Statements
++	i++	postfix	i=i+1; i+=1;
++	++i	prefix	i=i+1; i+=1;
--	i--	postfix	i=i-1; i -=1;
--	--i	prefix	i=i-1; i-=1;

Conditional (ternary)Operators (?:)

- C's only conditional (or **ternary**) operator requires three operands.

  Syntax: **conditional_expression? expression1: expression2;**
- The conditional_expression is any expression that results in a true (nonzero) or false (zero).
- If the result is true then expression1 executes, otherwise expression2 executes.

Example:

```
a=1;
b=2;
x = (a<b)?a:b;
```

This is like

```
if(a<b)
    x=a;
else
    x=b;
```

Bitwise Operators

- C has a special operator known as Bitwise operator for manipulation of data at bit level.
- Bitwise operator may not be applied for float and double.
- Manipulates the data which is in binary form.
- **Syntax:** **operand1 bitwise_operator operand2**

Bitwise Operators	Meaning
&	Bitwise AND
|	Bitwise OR
^	Exclusive OR
<<	Shift left
>>	Shift right
~	One's compliment

A	B	A&B	A|B	A^B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- **Examples:**

  & Bitwise AND 0110 & 0011 □ 0010

  | Bitwise OR 0110 | 0011 □ 0111

  ^ Bitwise XOR 0110 ^ 0011 □ 0101

  << Left shift 01101110 << 2 □ 10111000

  >> Right shift 01101110 >> 3 □ 00001101

  ~ One's complement ~0011 □ 1100

- Don't confuse bitwise & | with logical && ||

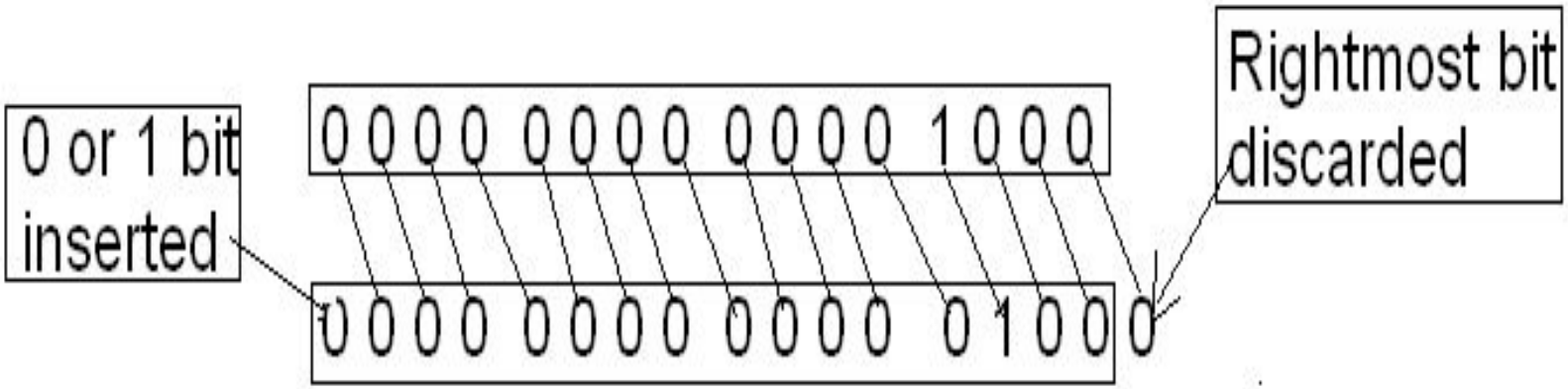
- >> is a binary operator that requires two integral operands. the first one is value to be shifted, the second one specifies number of bits to be shifted.
- The general form is as follows:

variable >> expression;

- When bits are shifted right, the bits at the rightmost end are deleted.
- Shift right operator divides by a power of 2. I.e. a>>n results in $a/2^n$, where **n** is number of bits to be shifted.

Example: a=8;

b=a>>1; // assigns 4 after shift right operation



- << is a binary operator that requires two integral operands. the first one is value to be shifted, the second one specifies number of bits to be shifted.
- The general form is as follows:

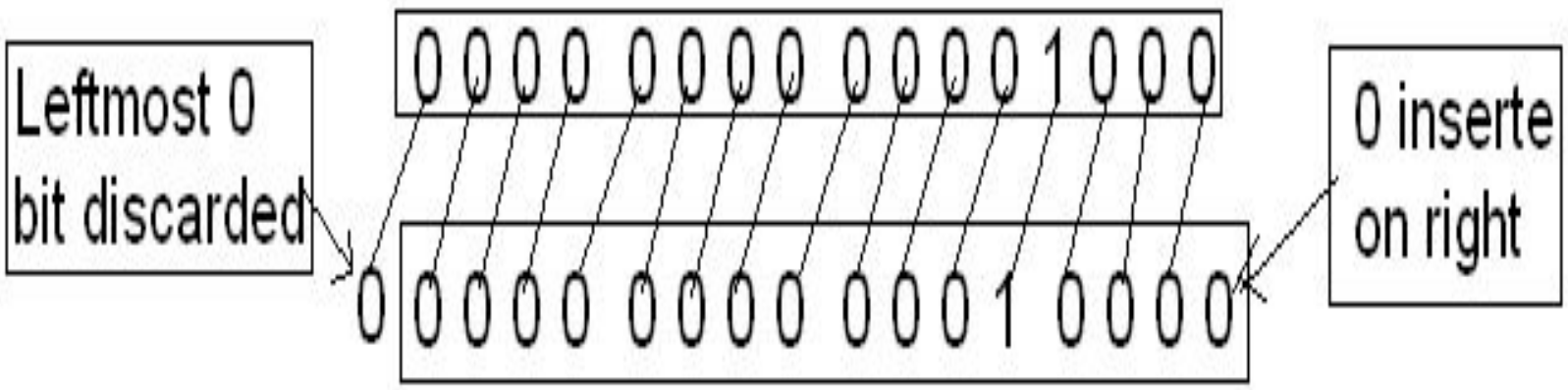
  variable << expression;

- When bits are shifted left, the bits at the leftmost end are deleted.

  Example: a=8;

  b=a<<1; // assigns 16 after left shift operation

- Shift left operator multiply by a power of 2, a<<n results in $a*2^n$, where **n** is number of bits to be shifted.



Special Operators

- C supports the following special category of operators.

&	Address operator
*****	Indirection operator
,	Comma operator
sizeof()	Size of operator
. and	Member selection Operators

comma operator :

- It doesn't operate on data but allows more than one expression to appear on the same line.

  Example: int i = 10, j = 20;
  printf (%d %.2f %c", a,f,c);
  j = (i = 12, i + 8); //i is assigned 12 added to 8 produces 20

sizeof Operator :

- It is a unary operator (operates on a single value).
- Produces a result that represent the size in bytes.

  Syntax: **sizeof(datatype);**

  Example:
  ```
  int a = 5;
  sizeof (a);          //produces 2
  sizeof(char);        // produces 1
  sizeof(int);         //  produces 2
  ```

Precedence and Association rules among operators

- **Precedence** is used to determine the order in which different operators in a complex expression are evaluated.

- **Associativity** is used to determine the order in which operators with the same precedence are valuated in a complex expression.

- Every operator has a precedence.

- The operators which has higher precedence in the expression is evaluated first.

  Example: a=8+4*2;

  a=?

Precedence and Associativity of Operators in C (from higher to lower)

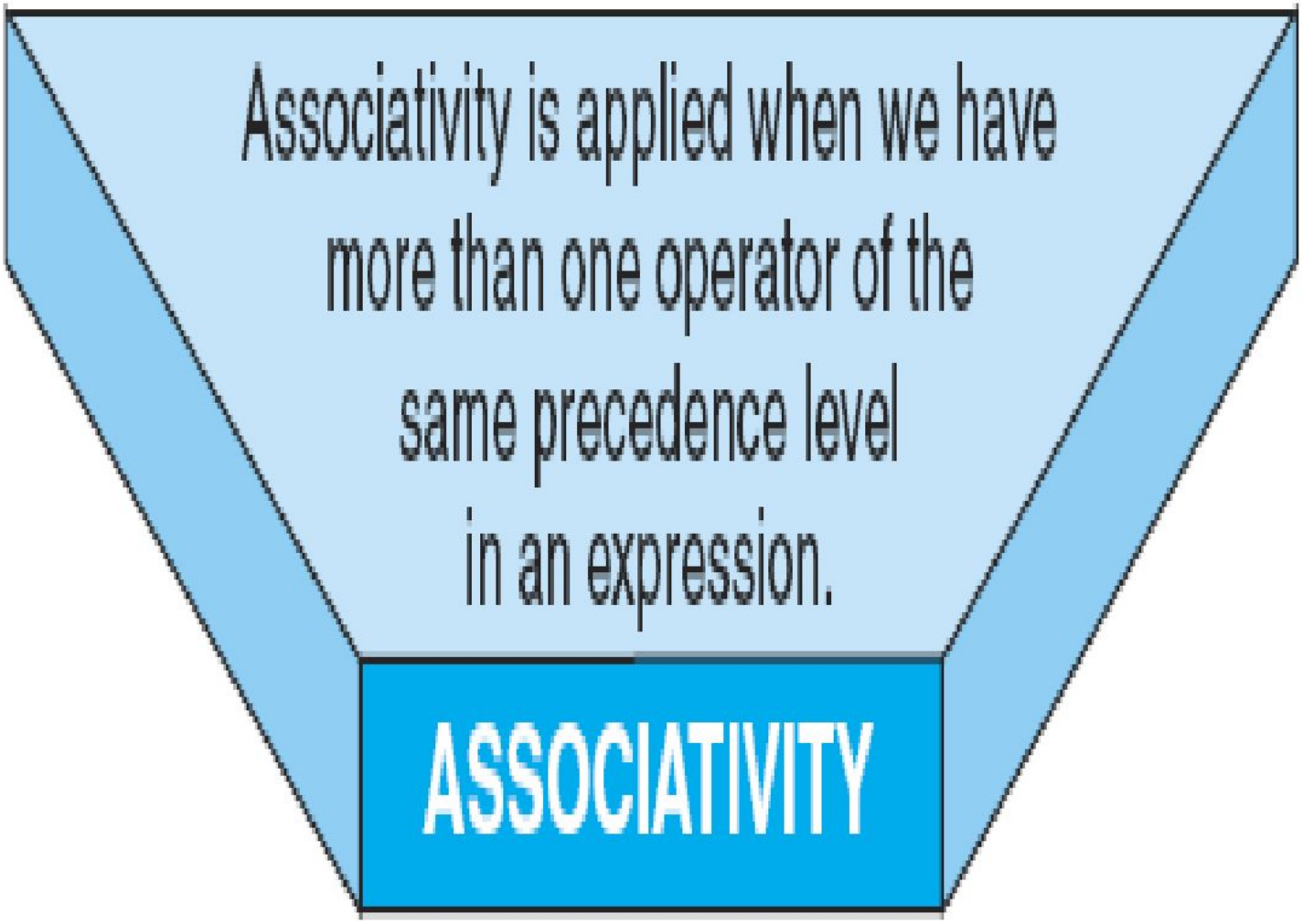
Category	Operator	Associativity
Postfix	`()  []  ->  .  ++  --`	Left to right
Unary,prefix	`+ - ! ~ ++ -- (type) * & sizeof`	Right to left
Multiplicative	`*  /  %`	Left to right
Additive	`+  -`	Left to right
Shift	`<<  >>`	Left to right
Relational	`<  <=  >  >=`	Left to right
Equality	`==  !=`	Left to right
Bitwise AND	`&`	Left to right
Bitwise XOR	`^`	Left to right
Bitwise OR	`|`	Left to right
Logical AND	`&&`	Left to right
Logical OR	`||`	Left to right
Conditional	`?:`	Right to left
Assignment	`=  +=  -=  *=  /=  %=` `>>=  <<=  &=  ^=  |=`	Right to left
Comma	`,`	Left to right

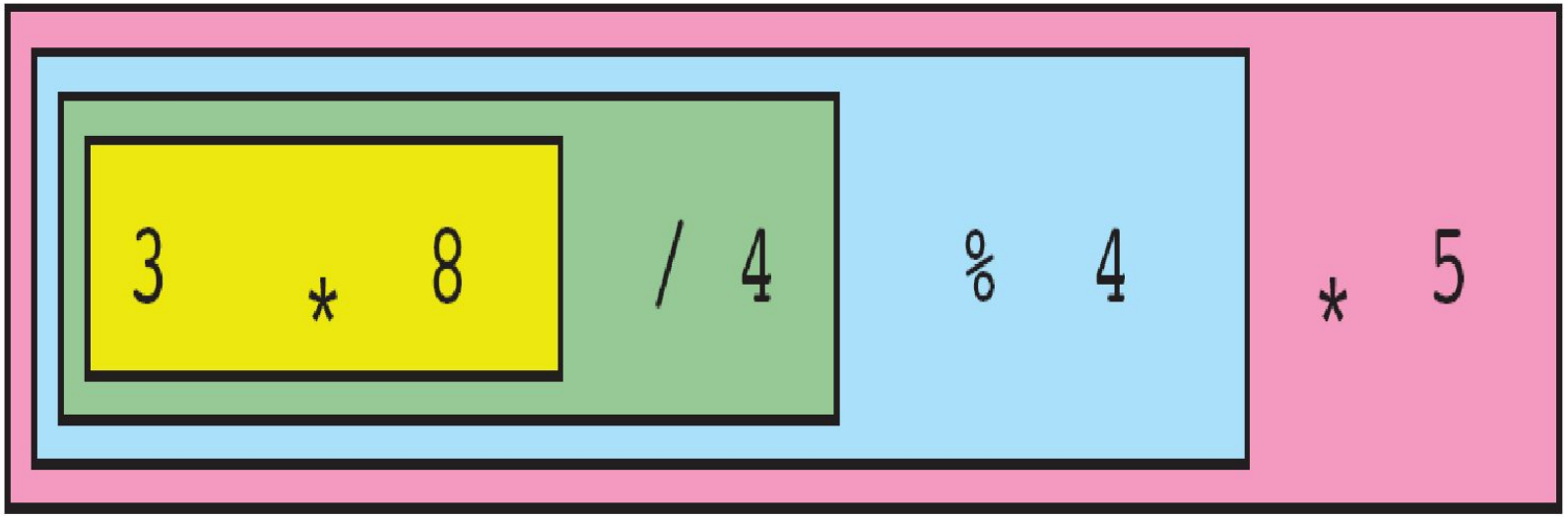
Example program to illustrate operator precedence

```
 1  /* Examine the effect of precedence on an expression.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int a = 10;
11     int b = 20;
12     int c = 30;
13
14  // Statements
15     printf ("a *  b + c  is: %d\n", a *  b + c);
16     printf ("a * (b + c) is: %d\n", a * (b + c));
17     return 0;
18  }  // main
```

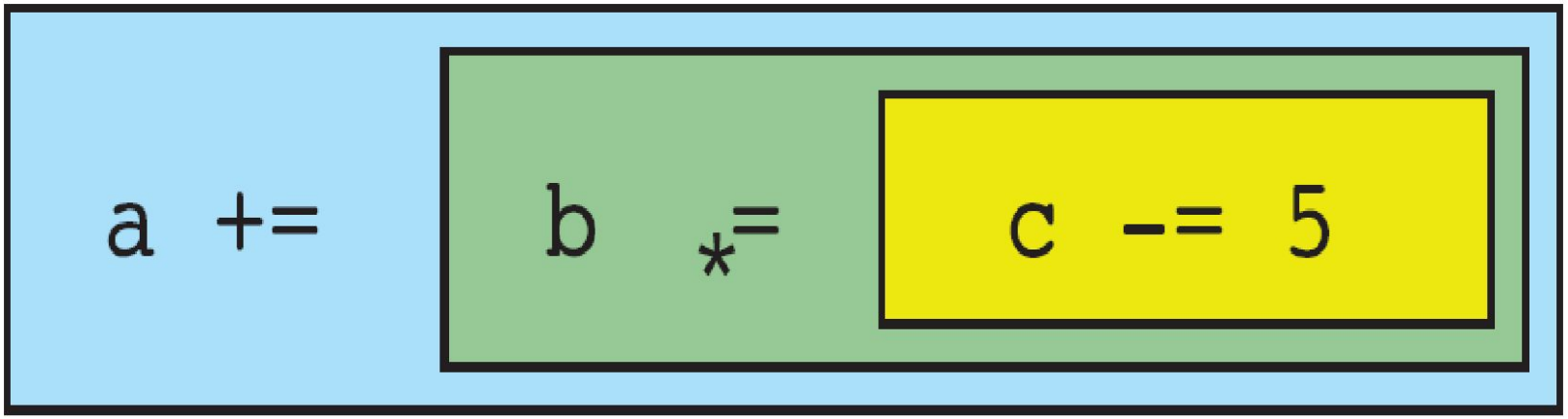
Example program to illustrate operator precedence (contd…)

```
Results:
a *  b + c  is: 230
a * (b + c) is: 500
```

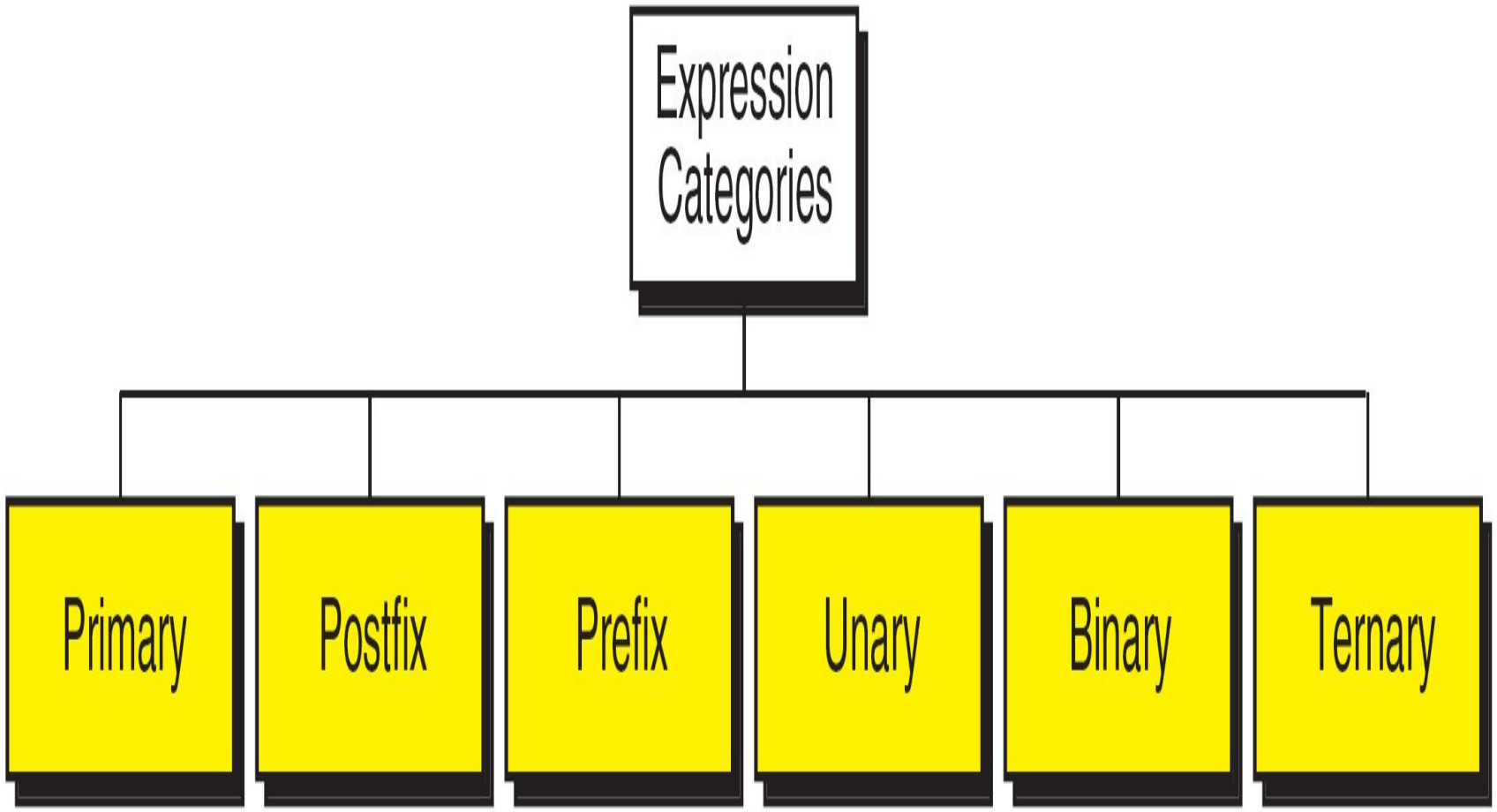

Associativity is applied when we have more than one operator of the same precedence level in an expression.
ASSOCIATIVITY



Left-to-Right Associativity



Right-to-Left Associativity



Expression Categories

- An **expression** is a sequence of operands and operators that reduces to a single value.

- Expressions can be simple or complex.

- An **operator** is a syntactical token that requires an action be taken.

- An **operand** is an object on which an operation is performed; it receives an operator's action.

Primary Expression:

- The most elementary type of expression is a primary expression.
- It consists of **only one operand with no operator.**
- In C, the operand in the primary expression can be a **name**, a **constant**,or a **parenthesized expression**.
- Name is any identifier for a variable, a function, or any other object in the language.
- The following are examples of primary expressions:

  Example: a price sum max

- Literal Constants is a piece of data whose value can't change during the execution of the program.

Primary Expression: (contd…)

- The following are examples of literal constants used in primary expression:
  Example: ‘A’ 56 98 12.34

- Any value enclosed in parentheses must be reduced in a single value is called as primary expression.

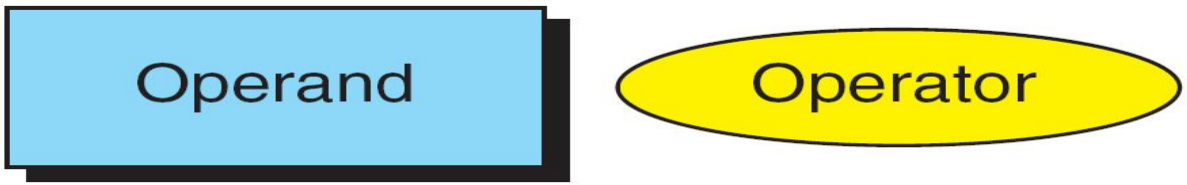
- The following are example of parentheses expression:
  Example: (a*x + b) (a-b*c) (x+90)

Post fix expression:

- It is an expression which contains **operand** followed by one **operator**.

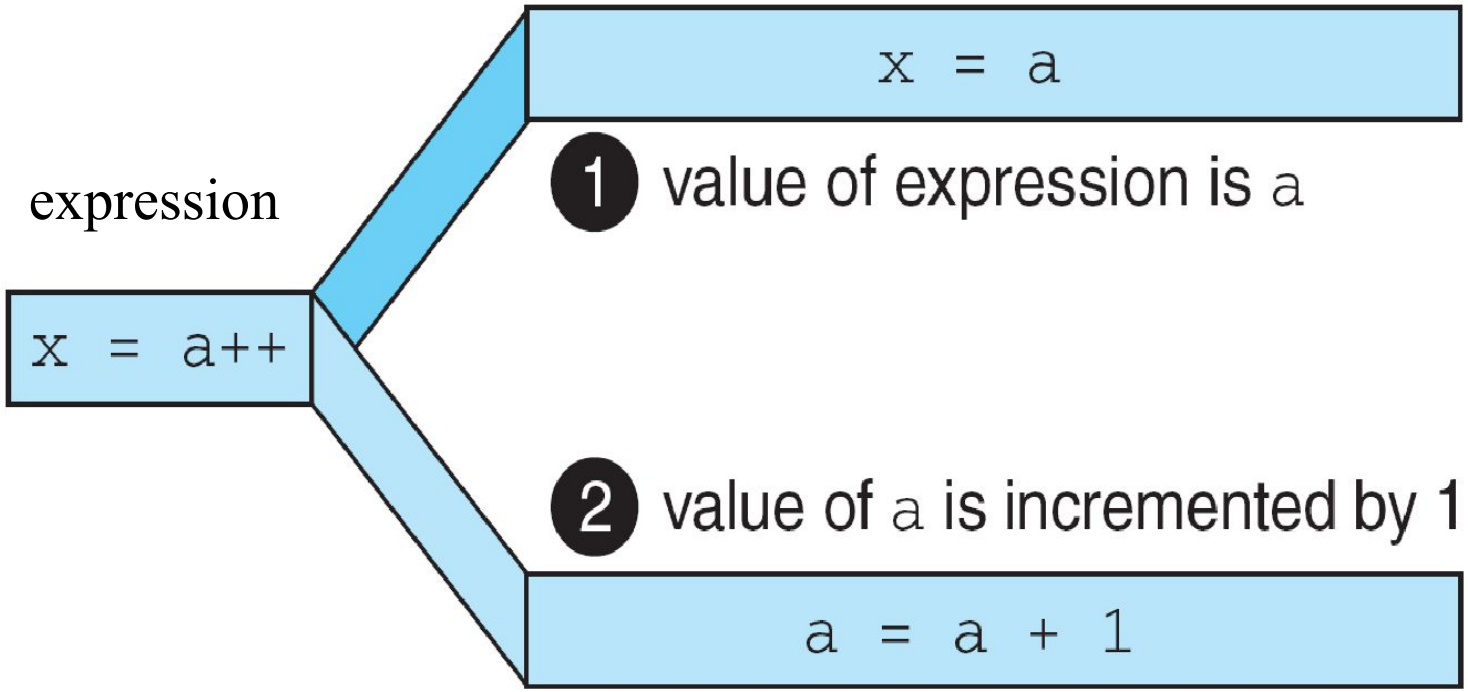
Example: a++; a- -;

- The operand in a postfix expression must be a variable.
- (a++) has the same effect as (a = a + 1)
- If ++ is after the operand, as in a++, the increment takes place **after** the expression is evaluated.



In the following figure:

1. Value of the variable a is assigned to x
2. Value of the a is incremented by 1.



Result of Postfix a++

Demonstrate Postfix Increment

```
 1  /* Example of postfix increment.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  int main (void)
 7  {
 8  // Local Declarations
 9     int a;
10
11  // Statements
12     a = 4;
13     printf("value of a     : %2d\n", a);
14     printf("value of a++   : %2d\n",   a++);
15     printf("new value of a: %2d\n\n", a);
16     return 0;
17  }  // main
```

```
Results:
value of a    :  4
value of a++  :  4
new value of a:  5
```

Example for Post fix expression

```
#include<stdio.h>
  void main()
  {
      a=10;
      x=a++;
      printf("x=%d, a=%d",x,a);
   }
```

Output:

x=10, a=11

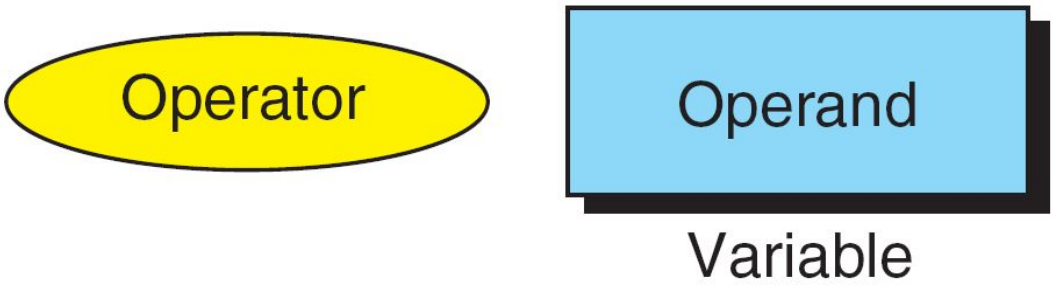
Pre fix expression:

- It is an expression which contains **operator** followed by an **operand**.

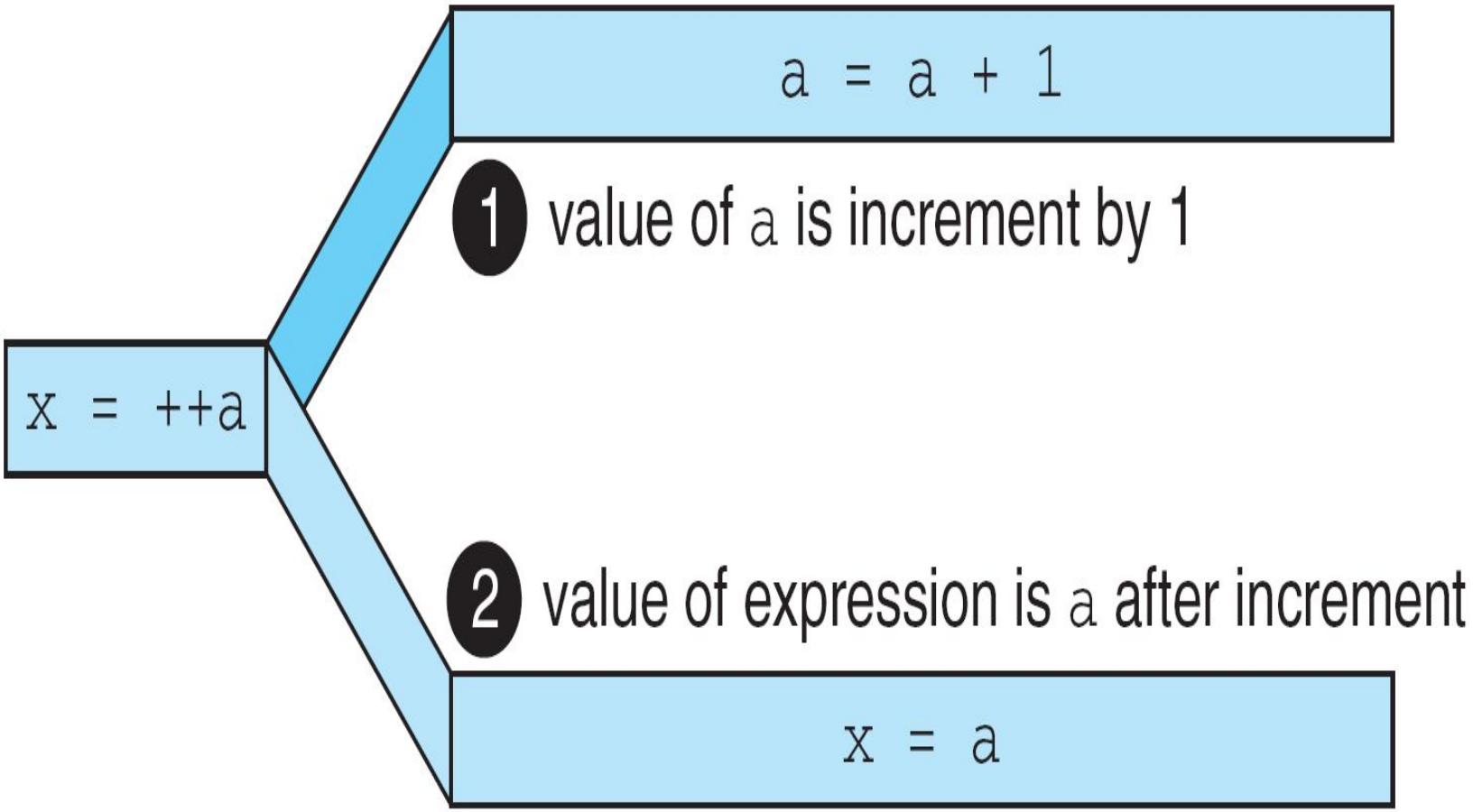
  Example: ++a; - -a;

- The operand of a prefix expression must be a variable.
- (++a) has the same effect as (a = a + 1)

- If ++ is before the operand, as in ++a, the increment takes place **before** the expression is evaluated.



Prefix Expression



Result of prefix ++a

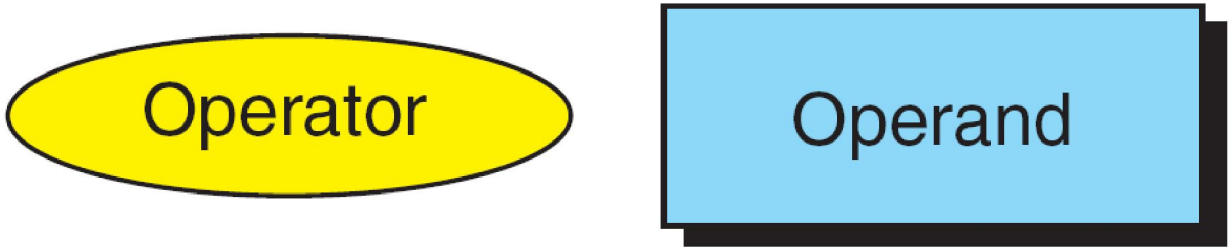
Example on pre fix expression

```
#include<stdio.h>
void main()
{
      a=10;
      x=++a;
      printf("x=%d, a=%d",x,a);
}
```

Output:

x=11, a=11

Unary expression: It is an expression which consists of unary operator followed by the operand



Expression	Contents of a Before *and* After Expression	Expression Value
+a	3	+3
-a	3	-3
+a	-5	-5
-a	-5	+5

Examples of Unary Plus And Minus Expressions

Binary Expressions:

- In binary expression **operator** must be placed in between the **two operands**.

- Both operands of the modulo operator (%) must be integral types.

Example for Binary Expressions

```
 1  /* This program demonstrates binary expressions.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  int main (void)
 7  {
 8  // Local Declarations
 9     int   a = 17;
10     int   b = 5;
11     float x = 17.67;
12     float y = 5.1;
13
14  // Statements
15     printf("Integral calculations\n");
16     printf("%d + %d  = %d\n", a, b, a + b);
```

Example for Binary Expressions (continued)

```
17      printf("%d - %d  = %d\n", a, b, a - b);
18      printf("%d * %d  = %d\n", a, b, a * b);
19      printf("%d / %d  = %d\n", a, b, a / b);
20      printf("%d %% %d  = %d\n", a, b, a % b);
21      printf("\n");
25      printf("%f - %f  = %f\n", x, y, x - y);
26      printf("%f * %f  = %f\n", x, y, x * y);
27      printf("%f / %f  = %f\n", x, y, x / y);
28      return 0;
29  }  // main
```

Example for Binary Expressions (continued)

```
Results:
Integral calculations
17 + 5 = 22
17 - 5 = 12
17 * 5 = 85
17 / 5 = 3
17 % 5 = 2

Floating-point calculations
17.670000 + 5.100000  = 22.770000
17.670000 - 5.100000  = 12.570000
17.670000 * 5.100000  = 90.116997
17.670000 / 5.100000  = 3.464706
```

The left operand in an **assignment expression** must be a single variable.

Compound Expression	Equivalent Simple Expression
`x *= expression`	`x = x * expression`
`x /= expression`	`x = x / expression`
`x %= expression`	`x = x % expression`
`x += expression`	`x = x + expression`
`x -= expression`	`x = x - expression`

Expansion of Compound Expressions

Demonstration of Compound Assignments

```
 1  /* Demonstrate examples of compound assignments.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int x;
11     int y;
12
13  // Statements
14     x = 10;
15     y = 5;
16
```

Demonstration of Compound Assignments (contd…)

```
17      printf("x: %2d  |  y: %2d ",     x, y);
18      printf("  |  x *= y + 2: %2d ", x *= y + 2);
19      printf("  |  x is now: %2d\n",  x);
20
21      x = 10;
22      printf("x: %2d  |  y: %2d ",     x, y);
23      printf("  |  x /= y + 1: %2d ", x /= y + 1);
24      printf("  |  x is now: %2d\n",  x);
25
26      x = 10;
27      printf("x: %2d  |  y: %2d ",      x, y);
28      printf("  |  x %%= y - 3: %2d ", x %= y - 3);
29      printf("  |  x is now: %2d\n", x);
30
31      return 0;
32  }   // main
```

Demonstration of Compound Assignments (contd…)

```
Results:
x: 10  |  y:  5  |  x *= y + 2: 70  |  x is now: 70
x: 10  |  y:  5  |  x /= y + 1:  1  |  x is now:  1
x: 10  |  y:  5  |  x %= y - 3:  0  |  x is now:  0
```

Evaluating Expressions

- A side effect is an action that results from the evaluation of an expression.

- For example, in an assignment, C first evaluates the expression on the right of the assignment operator and then places the value in the left variable.

- Changing the value of the left variable is a side effect.

Evaluating Expressions

```
 1  /* Evaluate two complex expressions.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  int main (void)
 7  {
 8  // Local Declarations
 9     int a = 3;
10     int b = 4;
11     int c = 5;
12     int x;
13     int y;
14
15  // Statements
16     printf("Initial values of the variables: \n");
17     printf("a = %d\tb = %d\tc = %d\n\n", a, b, c);
18
```

Evaluating Expressions (contd…)

```
19      x = a * 4 + b / 2 - c * b;
20      printf
21        ("Value of  a *  4 + b  / 2 - c  * b:  %d\n", x);
22
23      y = --a * (3 + b) / 2 - c++ * b;
24      printf
25        ("Value of --a * (3 + b) / 2 - c++ * b: %d\n", y);
26      printf("\nValues of the variables are now: \n");
27      printf("a = %d\tb = %d\tc = %d\n\n", a, b, c);
28
29      return 0;
30  }   // main
```

```
Results:
Initial values of the variables:
a = 3   b = 4   c = 5

Value of   a *  4 + b  / 2 - c   * b:  -6
Value of --a * (3 + b) / 2 - c++ * b: -13

Values of the variables are now:
a = 2   b = 4   c = 6
```

Type Conversion

- Up to this point, we have assumed that all of our expressions involved data of the same type.
- But, what happens when we write an expression that involves two different data types, such as multiplying an integer and a floating-point number?
- To perform these evaluations, one of the types must be converted.

- **Type Conversion:** Conversion of one data type to another data type.

- Type conversions are classified into:
  - Implicit Type Conversion
  - Explicit Type Conversion (Cast)

Implicit Conversion:

- In implicit type conversion, if the operands of an expression are of different types, the lower data type is automatically converted to the higher data type before the operation evaluation.

- The result of the expression will be of higher data type.
- The final result of an expression is converted to the type of the variable on the LHS of the assignment statement, before assigning the value to it.
- Conversion during assignments:

```
char c = 'a';
int i;
i = c;    /* i  is assigned by the ascii of ‘a’ */
```

- Arithmetic Conversion: If two operands of a binary operator are not the same type, **implicit** conversion occurs:

```
int i = 5 , j = 1;
float x = 1.0, y;
y = x / i;          /* y = 1.0 /  5.0  */
y = j / i;          /* y = 1 / 5  so  y = 0 */
```

Explicit Conversion or Type Casting:

- In explicit type conversion, the user has to enforce the compiler to convert one data type to another data type by using typecasting operator.

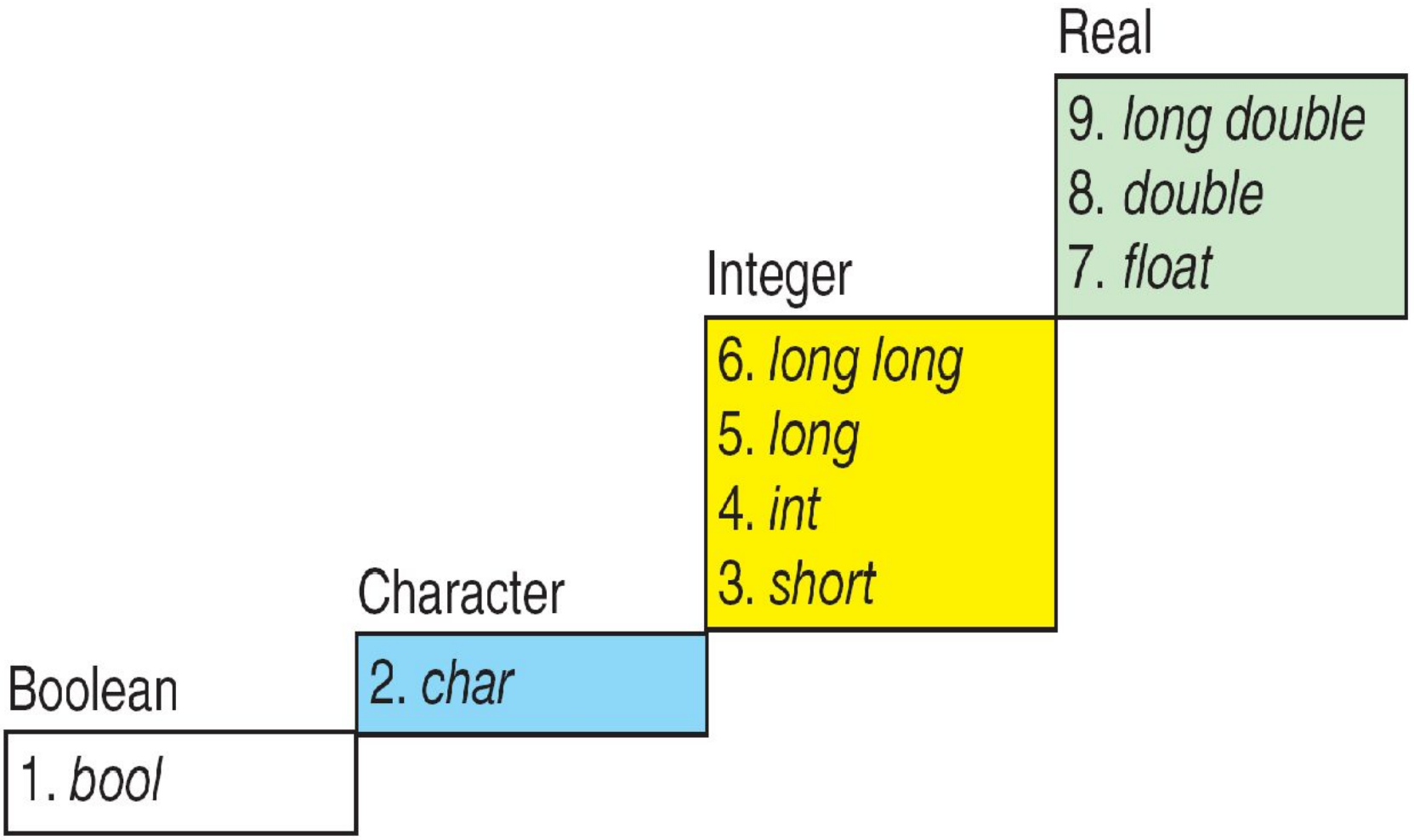
- This method of typecasting is done by prefixing the variable name with the **data type enclosed within parenthesis**.

  (data type) expression

- Where (**data type**) can be any valid C data type and expression is any variable, constant or a combination of both.

Example:

```
int x;
x=(int)7.5;
```



Conversion Rank (C Promotion Rules)

Example program for Implicit Type Conversion

```
 1  /* Demonstrate automatic promotion of numeric types.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <stdbool.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     bool  b  = true;
12     char  c  = 'A';
13     float d  = 245.3;
14     int   i  = 3650;
15     short s  = 78;
16
```

Example program for Implicit Type Conversion (contd…)

```
17  // Statements
18     printf("bool + char is char:   %c\n", b + c);
19     printf("int * short is int:    %d\n", i * s);
20     printf("float * char is float: %f\n", d * c);
21
22     c = c + b;                    // bool promoted to char
23     d = d + c;                    // char promoted to float
24     b = false;
25     b = -d;                       // float demoted to bool
26
27     printf("\nAfter execution...\n");
28     printf("char + true:    %c\n", c);
29     printf("float + char:   %f\n", d);
30     printf("bool = -float:  %f\n", b);
31
32     return 0;
33  }  // main
```

Example program for Implicit Type Conversion (contd…)

```
Results:
bool + char is char:   B
int * short is int:    284700
float * char is float: 15944.500000

After execution...
char + true:    B
float + char:   311.299988
bool = -float:  1
```

Example program for Explicit Casts

```
 1  /* Demonstrate casting of numeric types.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     char   aChar    = '\0';
11     int    intNum1  = 100;
12     int    intNum2  =  45;
13     double fltNum1  = 100.0;
14     double fltNum2  =  45.0;
15     double fltNum3;
16
17  // Statements
18     printf("aChar numeric   :  %3d\n",   aChar);
```

Example program for Explicit Casts (contd…)

```
19      printf("intNum1 contains:  %3d\n",   intNum1);
20      printf("intNum2 contains:  %3d\n",   intNum2);
21      printf("fltNum1 contains:  %6.2f\n", fltNum1);
22      printf("fltNum2 contains:  %6.2f\n", fltNum2);
23
24      fltNum3 = (double)(intNum1 / intNum2);
25      printf
26         ("\n(double)(intNum1 / intNum2): %6.2f\n",
27          fltNum3);
28
29      fltNum3 = (double)intNum1 / intNum2;
30      printf("(double) intNum1 / intNum2 : %6.2f\n",
31             fltNum3);
32
33      aChar = (char)(fltNum1 / fltNum2);
34      printf("  (char)(fltNum1 / fltNum2): %3d\n",   aChar);
35
```

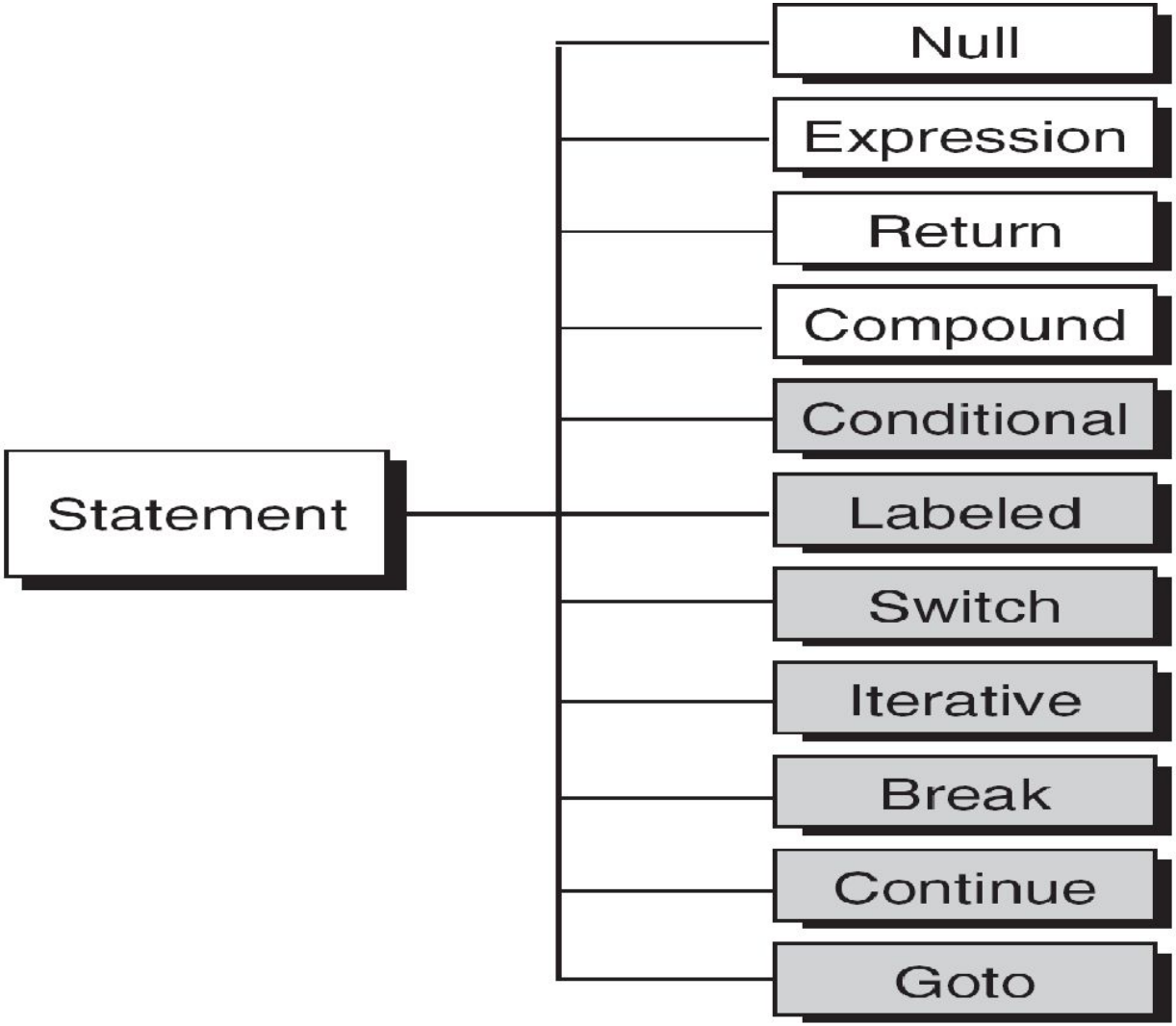
Example program for Explicit Casts (contd…)

```
36      return 0;
37  }   // main
```

```
Results:
aChar numeric    :     0
intNum1 contains:   100
intNum2 contains:    45
fltNum1 contains:   100.00
fltNum2 contains:    45.00

(double)(intNum1 / intNum2):    2.00
(double) intNum1 / intNum2 :    2.22
  (char)(fltNum1 / fltNum2):    2
```

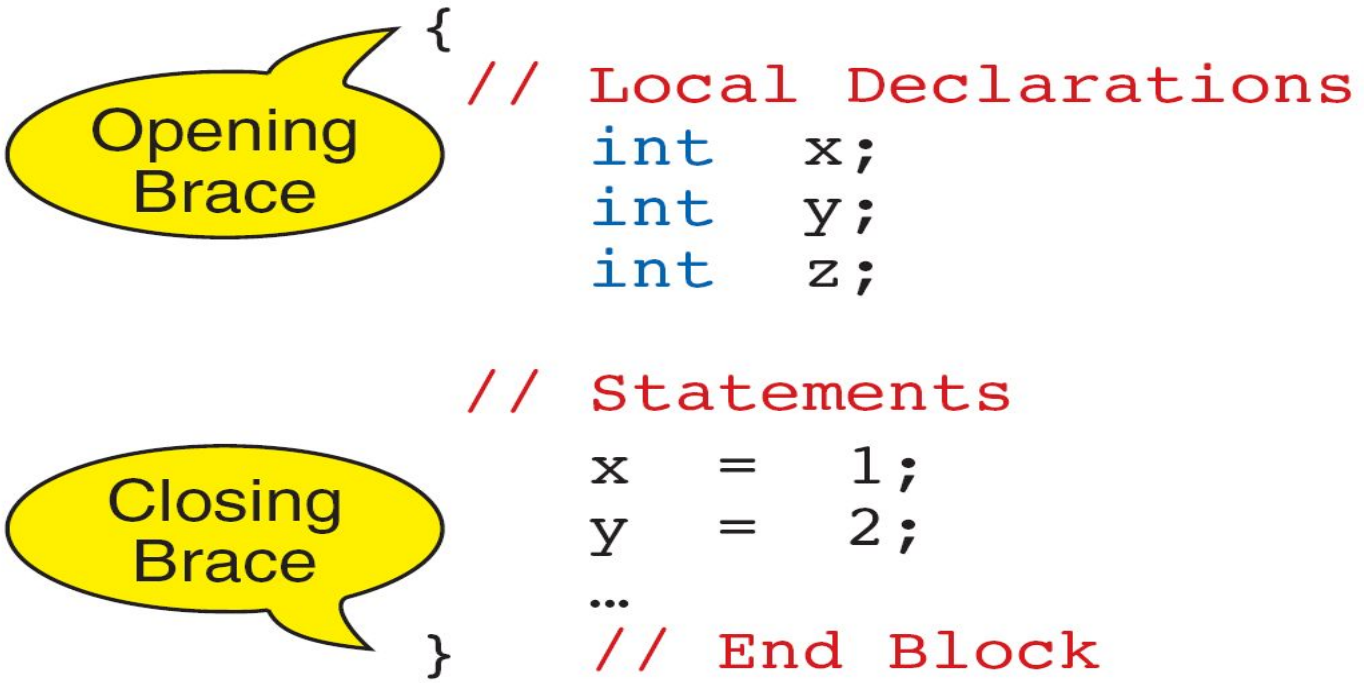
Statements

- A statement causes an action to be performed by the program.
- It translates directly into one or more executable computer instructions.
- Generally statement is ended with semicolon.
- Most statements need a semicolon at the end; some do not.



Types of Statements

- Compound statements are used to group the statements into a single executable unit.
- It consists of one or more individual statements enclosed within the braces { }



Compound Statement

Decision Control Structures

- The decision is described to the computer as a conditional statement that can be answered either **true** or **false**.

- If the answer is true, one or more action statements are executed.

- If the answer is false, then a different action or set of actions is executed.

Types of decision control structures:

- if
- if..else
- nested if…else
- else if ladder
- dangling else
- switch statement

Decision Control Statement: if

The general form of a simple if statement is:

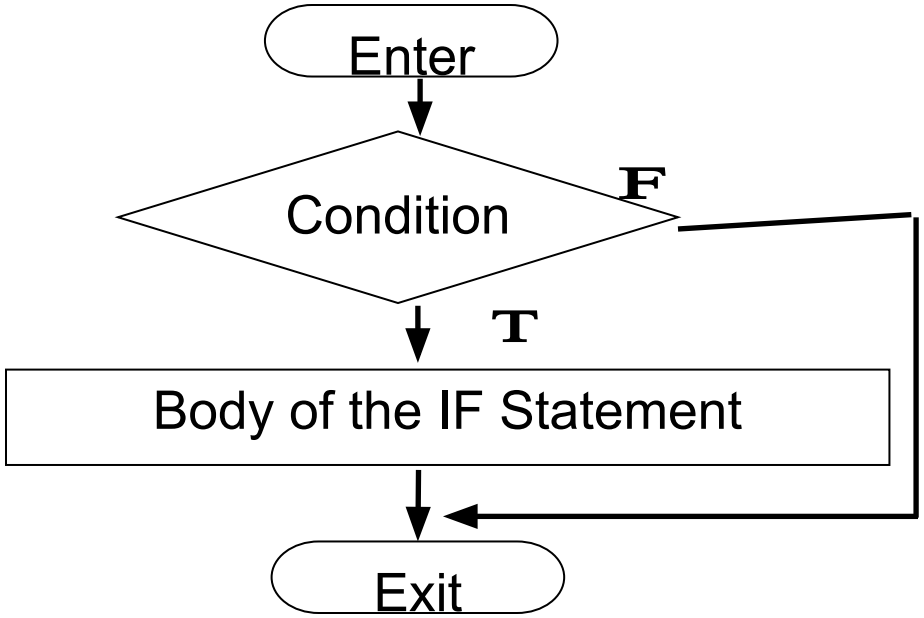
```
if (condition)
{
  statement-block;
}
```

Following are the properties of an if statement:

- If the condition is true then the statement-block will be executed.
- If the condition is false it does not do anything (or the statement is skipped)
- The condition is given in parentheses and must be evaluated as true (nonzero value) or false (zero value).
- If a compound statement is provided, it must be enclosed in opening and closing braces.

Decision Control Statement: if

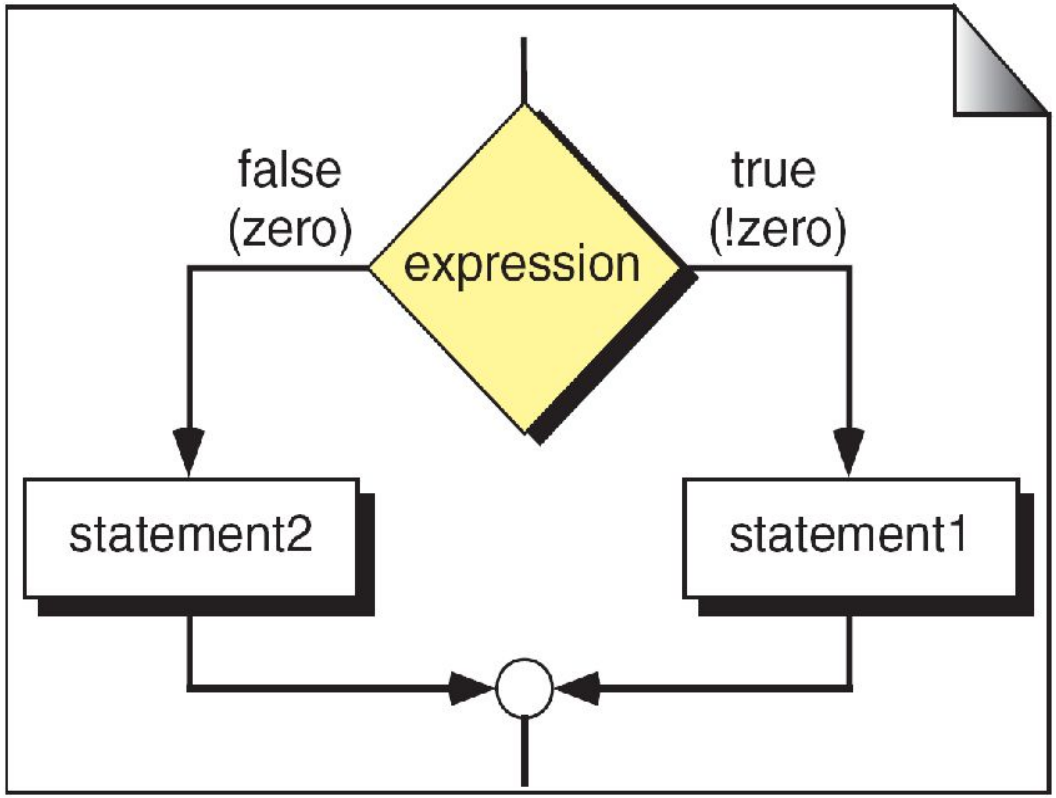
Flow Chart



Example:

```
main()
{
    int a=10,b=20;
    if(a>b)
    {
        printf("%d",a);
    }
    printf("%d",b);
}
```

Decision Control Statement: if..else



(a) Logical Flow

```
if (expression)
    statement1;
else
    statement2;
```

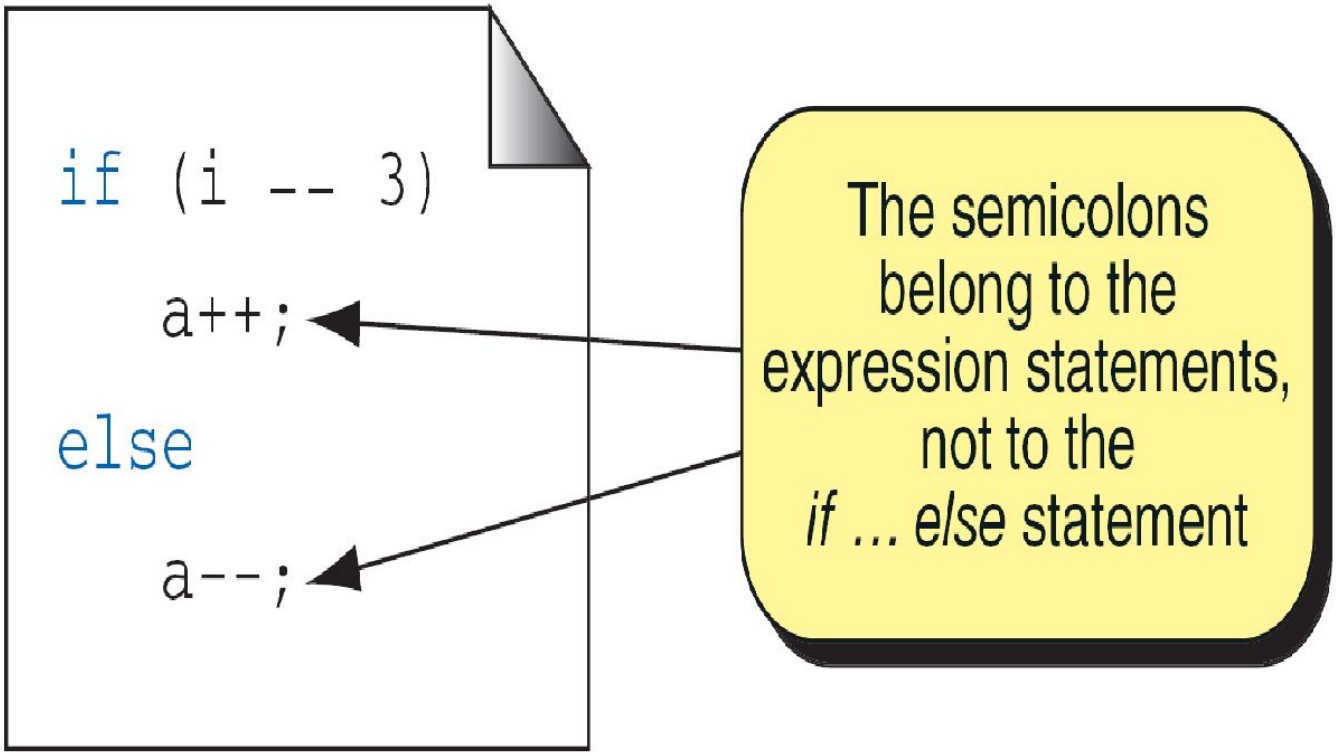
(b) Code

if...else Logic Flow

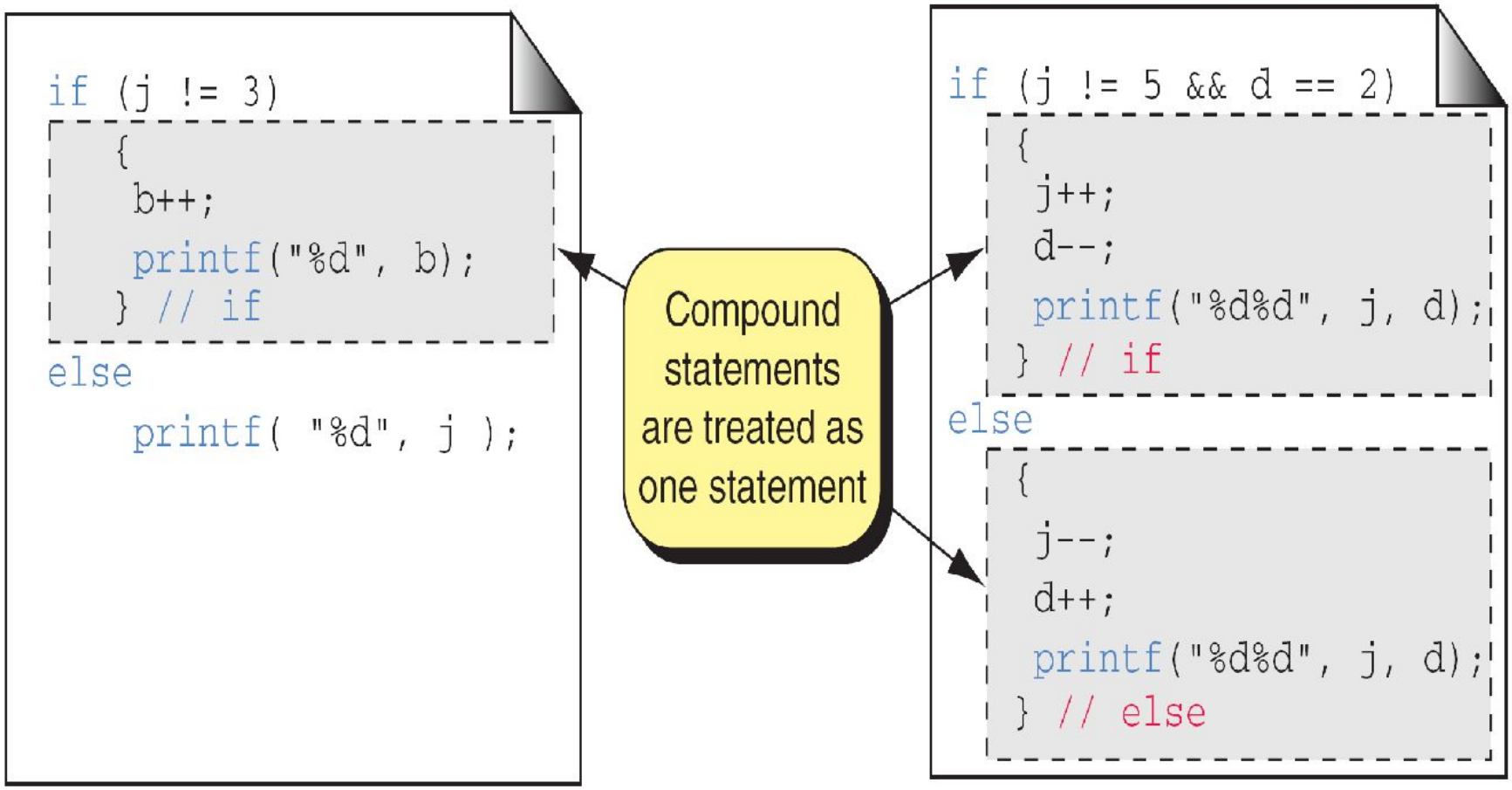
Syntactical Rules for **if…else** Statements

- The expression or condition which is followed by **if** statement must be enclosed in **parenthesis**.
- No **semicolon** is needed for an **if…else** statement.
- Both the true and false statements can be any statement (even another if…else).
- Multiple statements under **if and else** should be enclosed between curly braces.
- No need to enclose a single statement in curly braces.

Decision Control Statement: if..else



A Simple if...else Statement



Compound Statements in an if...else

Example for Decision Control Statement: if..else

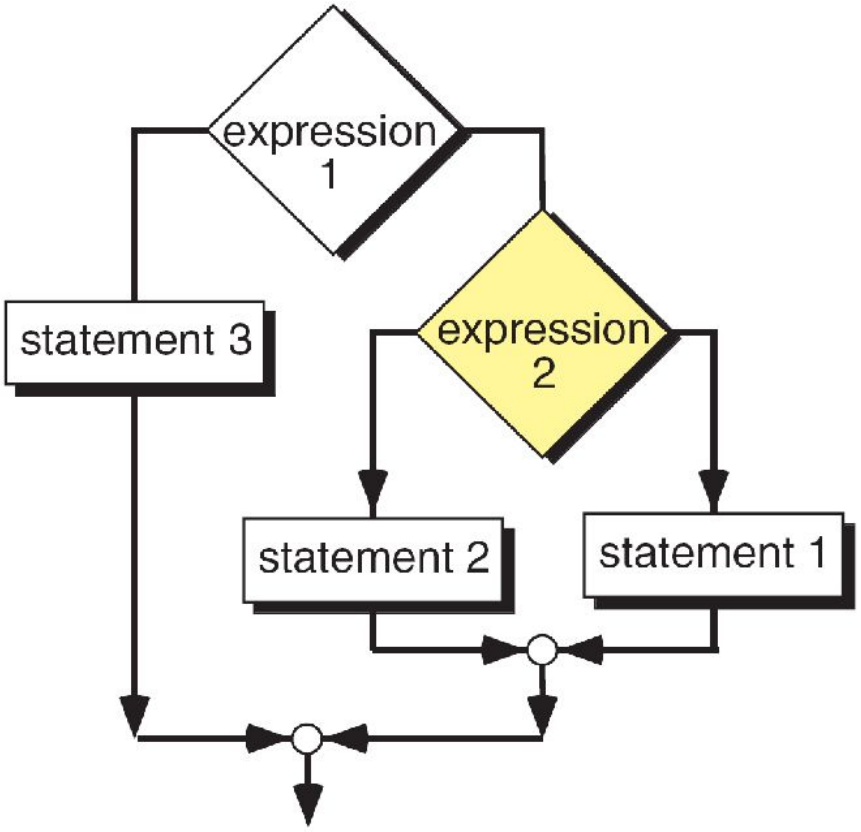
```
1  /* Two-way selection.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9     // Local Declarations
10    int a;
11    int b;
12
13    // Statements
14    printf("Please enter two integers: ");
15    scanf ("%d%d", &a, &b);
16
```

Example for Decision Control Statement: if..else

```
17      if (a <= b)
18          printf("%d <= %d\n", a, b);
19      else
20          printf("%d > %d\n", a, b);
21
22      return 0;
23  }   // main
```

```
Results:
Please enter two integers: 10 15
10 <= 15
```

Decision Control Statement: nested if…else



(a) Logic flow

```
if (expression 1)
    if (expression 2)
        statement 1
    else
        statement 2
else
    statement 3
```

(b) Code

- Nested if…else means within the **if…else** you can include **another if…else** either in **if block or else block**.

Decision Control Statement: nested if…else

```
 1  /* Nested if in two-way selection.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int a;
11     int b;
12
13  // Statements
14     printf("Please enter two integers: ");
15     scanf ("%d%d", &a, &b);
16
```

Decision Control Statement: nested if…else

```
17      if (a <= b)
18          if (a < b)
19              printf("%d < %d\n", a, b);
20          else
21              printf("%d == %d\n", a, b);
22      else
23          printf("%d > %d\n", a, b);
24
25      return 0;
26  }   // main
```

```
Results:
Please enter two integers: 10 10
10 == 10
```

Decision Control Statement: else if

```
if (condition1)
        statements1;
else if (condition2)
                statements2;
        else if (condition3)
                        statements3;
                else if (condition4)
                                statements4;
                        ……
                        else if(conditionn)
                                        statementsn;
                                else
                                                default_statement;
statement x;
```

- The conditions are evaluated from the top to down.
- As soon as a true condition is found the statement associated with it is executed and the control is transferred to the statementx by skipping the rest of the ladder.
- When all **n** conditions become false,final else containing default_statement that will be executed

Example program for nested if…else

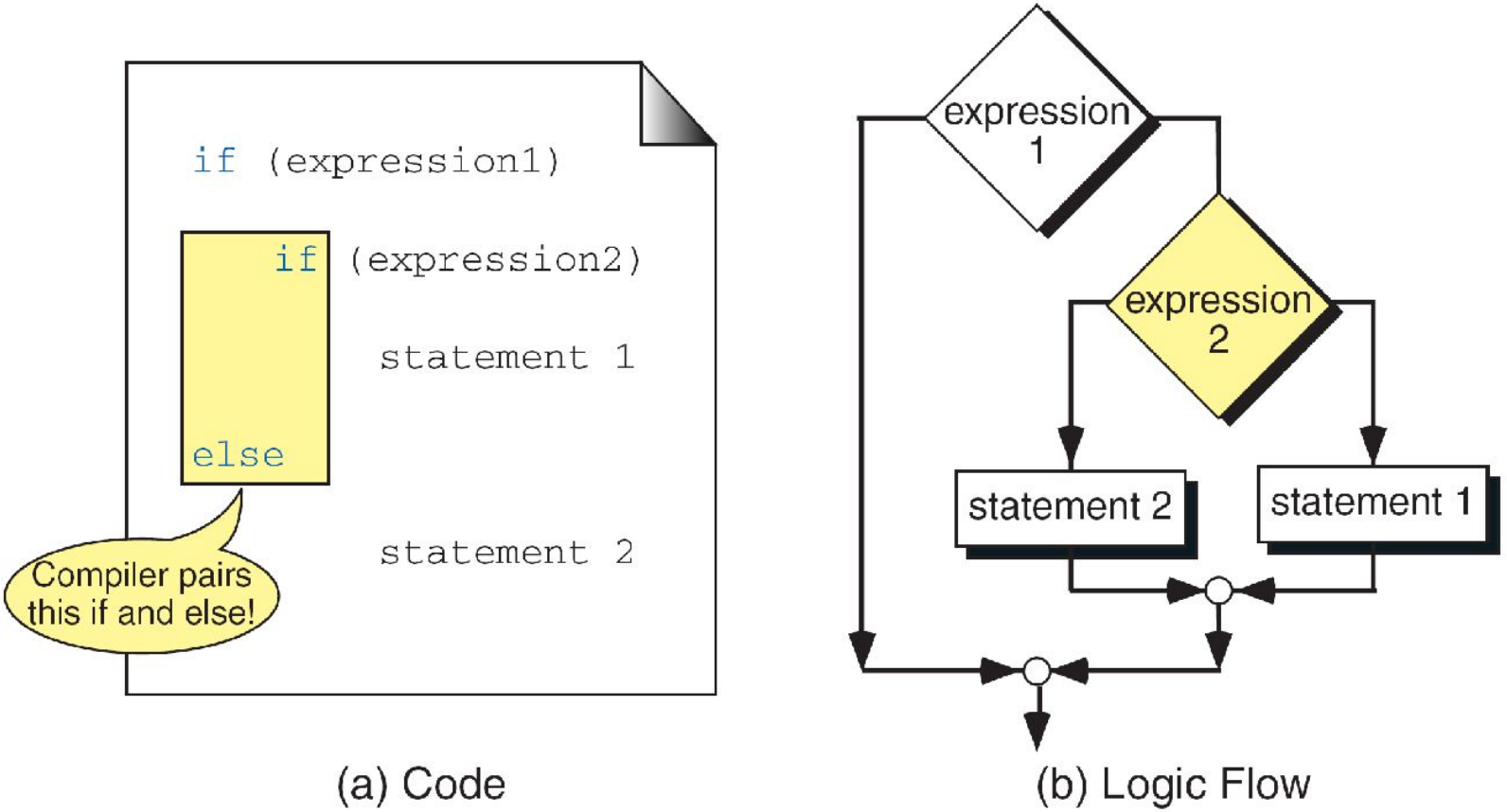
```
main()  {
        float m1,m2,m3,m4;
        float perc;
        printf("Enter marks\n");
        scanf("%f%f%f%f",&m1,&m2,&m3,&m4);
        perc=(m1+m2+m3+m4)/4;
        if(perc>=75)
                printf("\nDistinction");
        else {
                if(per<75 && per>=60)
                        printf("\nFirst Class");
                else {
                        if(per<60 && per>=50)
                                printf("\nSecond Class");
                        else {
                                if(per<50 && per>=40)
                                        printf("\nThird Class");
                                else
                                        printf("\nFail");
                        }//else
                }//else
        }//else
}//main
```

Example program for else if

```
main()
{
        float m1,m2,m3,m4;
        float perc;
        printf("enter marks\n");
        scanf("%f%f%f%f",&m1,&m2,&m3,&m4);
        perc=(m1+m2+m3+m4)/4;
        if(perc>=75)
                printf("\nDistinction");
        else  if(per<75 && per>=60)
                        printf("\nFirst Class");
                else if(per<60 && per>=50)
                                printf("\nSecond Class");
                        else  if(per<50 && per>=40)
                                        printf("\nThird Class");
                                else
                                        printf("\nFail");
}//main
```

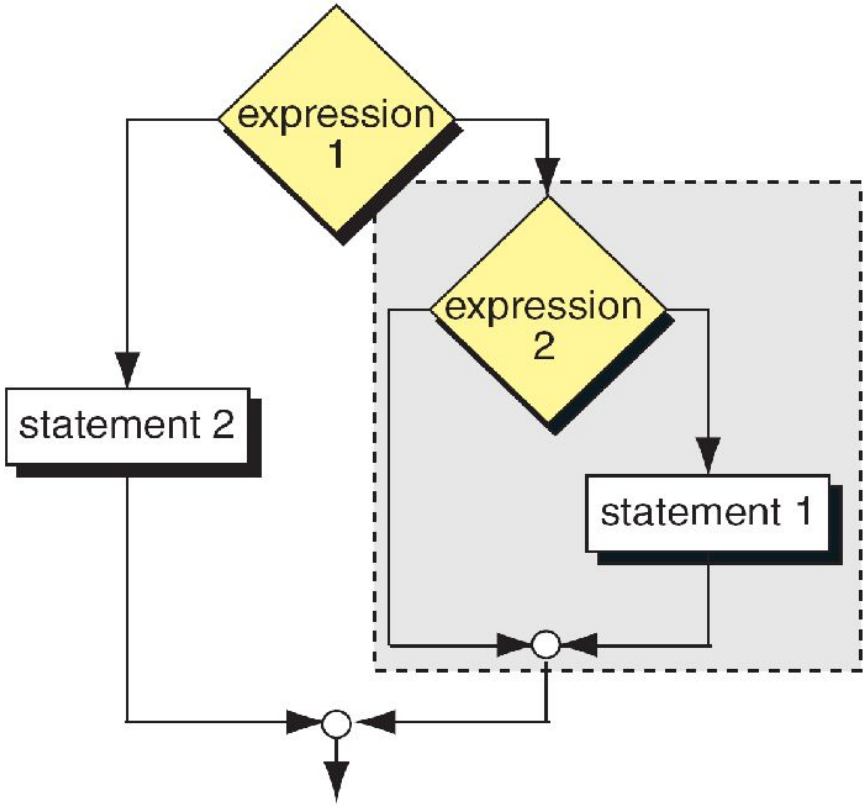
Dangling else

- **else** is always paired with the most recent unpaired **if**.

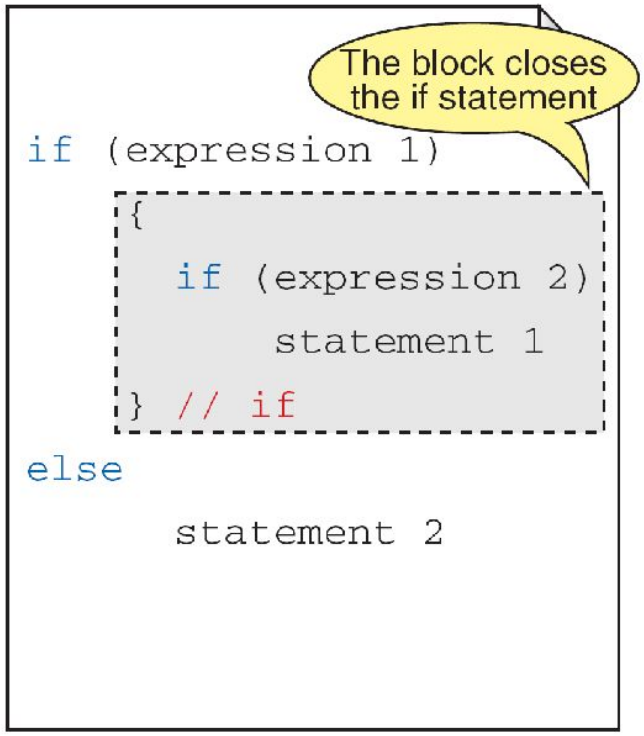


Dangling else (contd…)

- To avoid **dangling else** problem place the **inner if statement** with in the **curly braces**.



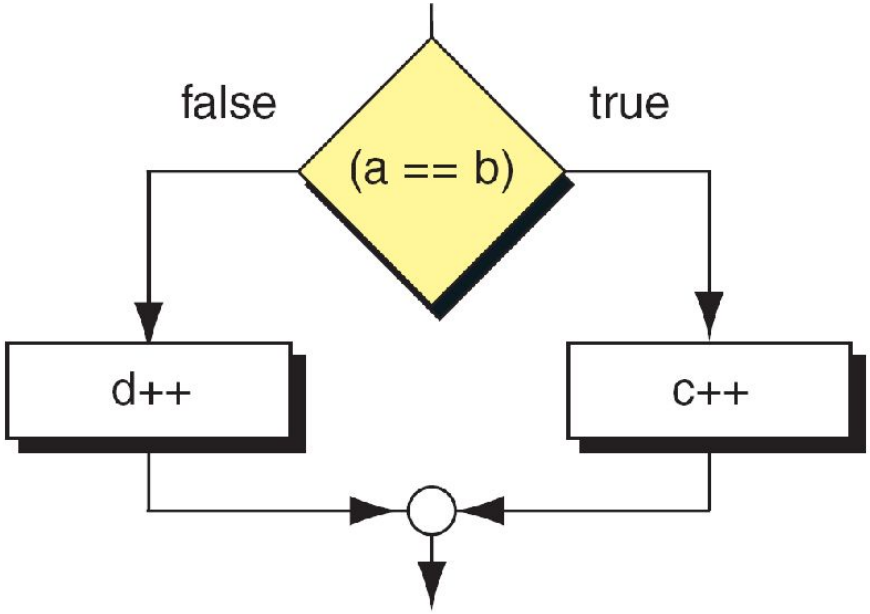
(a) Logic Flow



(b) Code

Dangling **else** Solution

- A simple **if…else** can be represented using the conditional (ternary) expression.



(a) Logic Flow

```
a == b ? c++ : d++;
```

(b) Code

Conditional Expression

Decision Control Statement: switch

- It is a **multi-way** conditional statement generalizing the **if…else** statement.
- It is a conditional control statement that allows some particular **group of statements to be chosen** from several available groups.
- A switch statement allows a single variable to be compared with several possible **case** labels, which are represented by constant values.
- If the variable matches with one of the constants, then an execution jump is made to that point.
- A case label cannot appear more than once and there can only be one default expression.

Decision Control Statement: switch

- Note: **switch** statement does not allow less than (<), greater than (>).
- ONLY the equality operator (==) is used with a switch statement.
- The control variable must be integral (int or char) only.
- When the switch statement is encountered, the control variable is evaluated.
- Then, if that evaluated value is equal to any of the values specified in a **case** clause, the statements immediately following the colon (":") begin to run.
- **Default case** is optional and if specified, default statements will be executed, if there is no match for the case labels.
- Once the program flow enters a **case** label, the statements associated with **case** have been executed, the program flow continues with the statement for the next case. (if there is no **break** statement after case label.)

Decision Control Statement: switch

General format of switch:

```
switch (expression)
   {
    case constant-1: statement
                     :
                     statement
    case constant-2: statement
                     :
                     statement
    case constant-n: statement
                     :
                     statement
    default        : statement
                     :
                     statement
   }
```

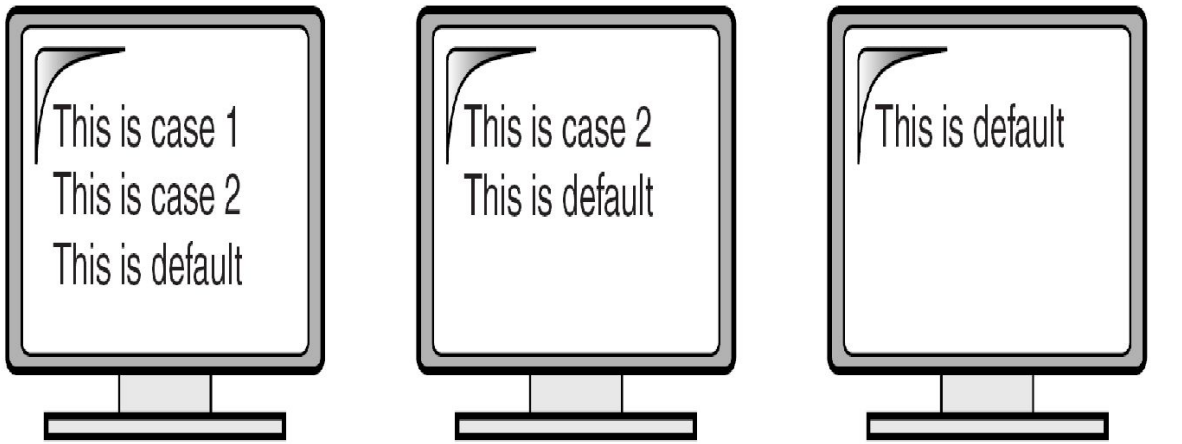
Decision Control Statement: switch

- The following results are possible, depending on the value of printFlag.
- If printFlag is 1, then all three printf statements are executed.
- If printFlag is 2, then the first print statement is skipped and the last two are executed.
- Finally, if printFlag is neither 1 nor 2, then only the statement defined by the default is executed.

Decision Control Statement: switch

Example1 for switch statement:

```
switch (printFlag)
   {
    case 1:  printf("This is case 1\n");

    case 2:  printf("This is case 2\n");

    default: printf("This is default\n");
   }
```



(a) printFlag is 1 (b) printFlag is 2 (c) printFlag is not 1 or 2

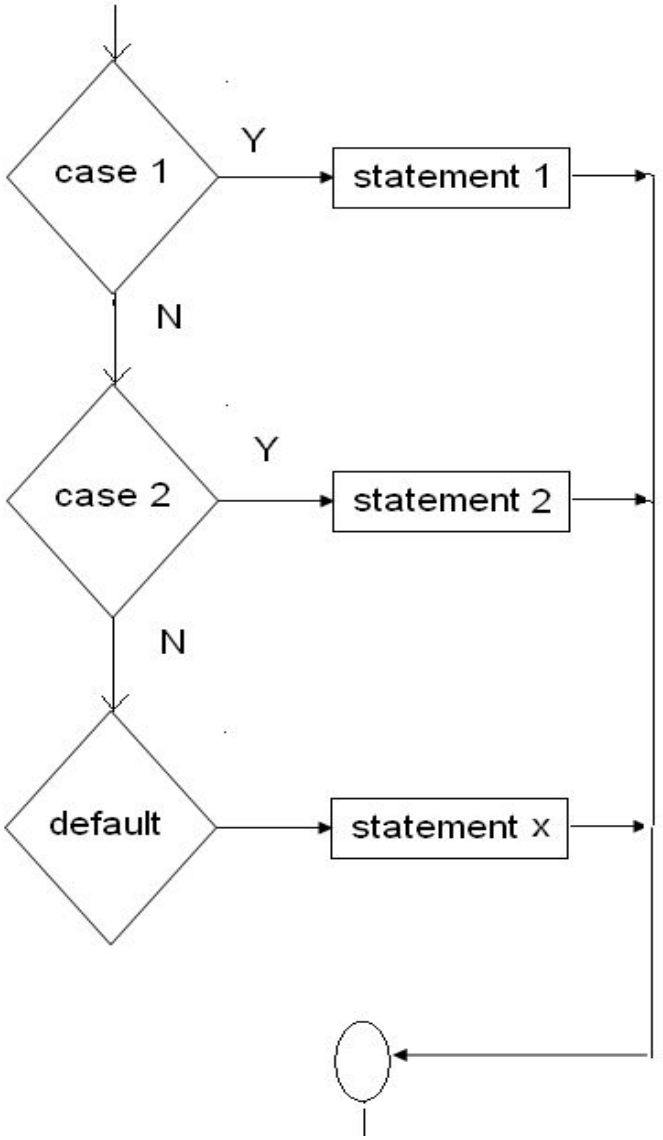
Decision Control Statement: switch

- If you want to execute only one case-label, C provides break statement.
- It causes the program to jump out of the switch statement, that is go to the closing braces (}) and continues the remaining code of the program.
- If we add break to the last statement of the case, the general form of switch case is as follows:

Decision Control Statement: switch

General format of switch:

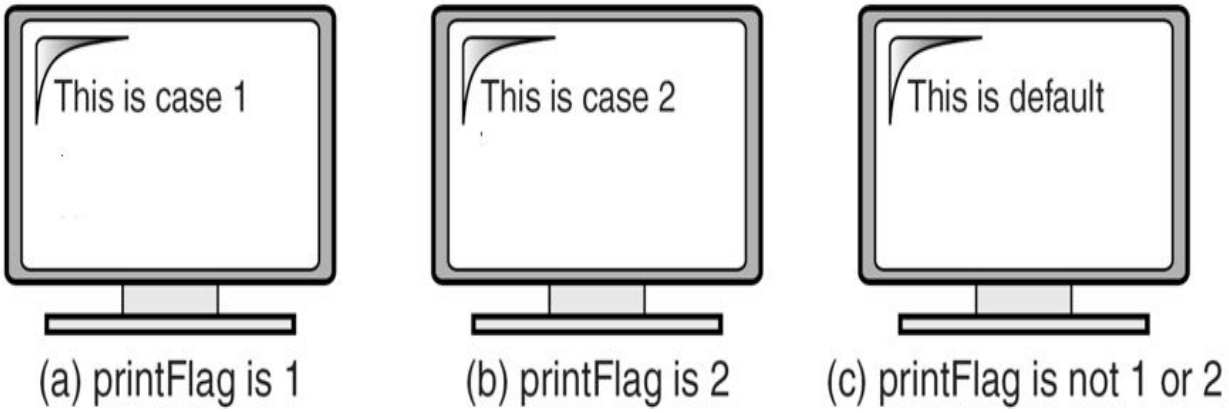
```
switch (expression)
  {
   case constant-1: statement
                    :
                   break;
   case constant-2: statement
                    :
                   break;
   case constant-n: statement
                    :
                    break;
   default        : statement
                    :
                   break;

  }
```



Decision Control Statement: switch

Example2 for switch statement:

```
switch (printFlag)
  {
   case 1:
            printf
              ("This is case 1");
            break;
   case 2:
            printf
              ("This is case 2");
            break;
   default:
            printf
              ("This is default");
            break;
}   // switch
```



(a) printFlag is 1 (b) printFlag is 2 (c) printFlag is not 1 or 2