BIRZEIT UNIVERSITY

Overview of C

PROBLEM SOLVING AND PROGRAM DESIGN In C
7th EDITION
Jeri R. Hanly, Elliot B. Koffman

By: Mamoun Nawahdah (PhD)
2013/2014

FIGURE 2.1 C Language Elements in Miles-to-Kilometers Conversion Program

```
/*
 * Converts distances from miles to kilometers.
 */
#include <stdio.h>           /* printf, scanf definitions */
#define KMS_PER_MILE 1.609   /* conversion constant       */

int
main(void)
{
        double miles, /* distance in miles
              kms;    /* equivalent distance in kilometers */

        /* Get the distance in miles. */
        printf("Enter the distance in miles> ");
        scanf("%lf", &miles);

        /* Convert the distance to kilometers. */
        kms = KMS_PER_MILE * miles;

        /* Display the distance in kilometers. */
        printf("That equals %f kilometers.\n", kms);

        return (0);
}
```

standard header file; comment; preprocessor directive; constant; reserved word; variable; comment; standard identifier; special symbol; reserved word; punctuation; special symbol

1. Pre-processor Directives

- Commands that give instructions to the **C** processor, whose job is to modify the text of **C** program before it is compiled.
- Begins with number symbol (**#**) as its first nonblank character.
- Two most common directives:

#include and **#define**

#include

- Gives a program access to a **library**.
- Notifies the pre-processor that some names used in the program are found in the standard header library.
- Examples:

#include <stdio.h>

  - The **stdio.h** header file contains the **scanf** and **printf** definitions.

#define

- Associates the **constant** macro with its value.
- Instructs the pre-processor to replace each occurrence of the constant macro in the text of the **C** program.
- Example:

```
#define  kms_per_mile  1.609
#define  max  100
```

2. Function main ()

- Marks the beginning of the program execution.
- The body of the function is enclosed in braces { .. }
- The function body has two parts:
  - **Declaration statements:** tells the compiler that **memory cells** are needed in the function.
  - **Executable statements:** are translated in machine language and later executed.

The main function definition (syntax)

```
int  main ( void ) {
     // function body
}
```

- The line **int** indicates that the main function **returns** an integer value to the operating system after normal termination.
- The symbol (**void**) indicates that the main function **receives** no data from the operating system before it begins execution.

3. Reserved Words

- Have special meaning in **C** and cannot be used for other purposes.
- All reserved words appear in **lowercase**.

ANSI C Reserved Words

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

4. C Identifiers

- Are **names** of entities in the program.
- Two Varieties:
  1. **Standard Identifiers**: can be redefined and used by the programmer.
  2. **User-defined Identifiers**: are own name to memory cell that will hold data program results.

SYNTAX rules for user-defined identifier

- Identifier **must** consist only of **letters**, **digits**, and **underscore**.
- An identifier cannot begin with a digit.
- A **C** reserved word cannot be used as identifier.
- An identifier defined in **C** standard library should not be redefined.
- An identifier may consist of as many as **127** characters
- Note: **C** is **case-sensitive**.

Example

Legal identifiers	Illegal identifiers
unit_weight	123a
num10	total amount
_counter	num#1
var1	percent%
name	int

TABLE 2.2 Invalid Identifiers

Invalid Identifier	Reason Invalid
`1Letter`	begins with a letter
`double`	reserved word
`int`	reserved word
`TWO*FOUR`	character * not allowed
`joe's`	character ' not allowed

5. Simple Data Types

- Each variable is assigned a data type since it tells the program that the variable can only be assigned that particular variable.
- **Four** simple data types:
  - The **int** Data Type.
  - The **float** and **double** Data Type.
  - The **char** Data Type.
  - The **enum** Data Type.

The int Data Type

- An **int** (for **int**eger) is a whole number consisting of an optional sign (**+** or **-**) followed by a sequence of digits.
- Example:

  12 -234 0 +678

- If the sign is omitted, the **+** sign is understood.
- The **int** type can contain no decimal point or fractional part and commas.

The float and double Data Type

- A **float** (floating point) is a number which can be written as a finite decimal, such as

  5.0 , **-0.785** , or **157.2**

- A **double** (double precision) is a special **float** which can store more significant digits and gave larger exponent.
- **double** takes up more space in the memory because they store wider range of numbers.

The char Data Type

- A **char** (**cha**racter) is a single letter, digit, punctuation mark, or control symbol recognized by the computer such as 'a', 'B', '7', ';' , '@', '%' and so on.
- A character is written enclosed with single quotation marks.

Data Type Modifiers

- Modifiers are used to extend the data-handling power of **C**. The **four** **modifiers** defined are: **signed**, **unsigned**, **long**, and **short**.
- These modifiers maybe used only to the **char** and **int** types although **long** can also be applied to **double**.
- Examples are:

unsigned int

short int

long double

C Data Types with Modifiers

TABLE 2.5 Integer Types in C

Type	Range in Typical Microprocessor Implementation
`short`	–32,767 .. 32,767
`unsigned short`	0 .. 65,535
`int`	–2,147,483,647 .. 2,147,483,647
`unsigned int`	0 .. 4,294,967,295
`long`	–2,147,483,647 .. 2,147,483,647
`unsigned long`	0 .. 4,294,967,295

TABLE 2.6 Floating-Point Types in C

Type	Approximate Range
`float`	10^{-37} .. 10^{38}
`double`	10^{-307} .. 10^{308}
`long double`	10^{-4931} .. 10^{4932}

Type Conversions

- Numeric data types can be converted automatically by assignment statement.
- For example if **x** is an **int** and **y** is **double**, and the values for each given variable are **5** and **10.0** respectively, the following assignments are valid:

```
y = x + 10;
x = y + 2.5;
```

Type Conversions cont.

- In the first line of statement, **5 + 10** equals to **15**, **C** then automatically convert it to **double** that is, **15.0** then assign to **y**.
- The next line evaluates to **12.5** (**10.0 + 2.5**) and **x** takes **12** after the assignment because the fractional part is being truncated and will be lost.

cast Operator

- cast operator **(type)** is used to explicitly convert one data type to another type.
- The process of explicitly converting a data type to another is called **typecasting**.
- The syntax is:

(data_type) variable

  - Where the value of the variable will be converted to the specified data type.

cast Operator

- There are two possible types of explicit conversion of data types:
  - **Widening conversion:** is a conversion from a smaller numeric type to a larger numeric type without any loss of information.
  - Example: **float x; int y;**
    x = (float) y;
  - **Narrowing conversion:** is a conversion from a larger numeric type to a smaller numeric type and **might** loss some information.
  - Example: **double x; int y;**
    y = (int) x;

6. Comments

- **Comments**: are symbols used to place notes in the program and is **ignored** by the complier.
- They are added for the programmer to put some remarks on a particular line/s of code.
- Here are the symbols that are used for comments:

  /* --------- ***/** for **multiple** lines of comment

  // --------- for **single** line comment

6. Statements

- **Statements**: is one line or lines of code placed in the program terminated by a semicolon (**;**).
- Examples are shown below:

```
x = 9;          // an assignment statement
x = x + 8;      // another assignment
printf("Hello Everyone!");
```

6. Blocks

- **Block**: is a collection of statements bound by opening and closing braces { }. They are also called compound statements.
- The **main** function body is placed inside a block:

```
int main(void){
        // function body
}
```

Basic Structure of a C Program

- Example: **Hello World!**

C Program:

```
#include <stdio.h>

int main()
{
  printf("Hello World!");

  return 0;
}
```

C Program:

```
#include <stdio.h>

int main()
{
   printf("Hello World!");

   return 0;
}
```

Includes declarations for the standard input/output library of procedures.

Read: "Hash-include"

C Program:

```
#include <stdio.h>

int main()
{
   printf("Hello World");

   return 0;
}
```

Curly braces mark the **beginning** and **end** of a block of instructions

Instruction (**function call**) to output "Hello World"

C Program:

```
#include <stdio.h>

int main()
{
   printf("Hello World");

   return 0;
}
```

"Statements" (lines of instructions) always end with a **semi-colon** (;)

C Program:

```
#include <stdio.h>

int main()
{
   printf("Hello World");

   return 0;
}
```

7. Variables

- Variables are **memory cells** used for storing a program's input data and its computational results.
- The values can change as the program executes.
- Here is the syntax for declaring variables:

<data_type> <identifier_list>;

  - where <**data_type**> is any data type which is standard to **C** or user-defined (like **enum**).
  - The variables listed in the <**identifier_list**> must be separated by commas.

Examples:

```
int    x ;
double  y , z ;
```

- Here, **x** is the name of the memory address that can only hold a value of **int** type and, **y** and **z** can contain double values.
- Since the variables declared have no specific values assigned, they will take any value from the memory (**garbage**).
- Declared variables can be assigned an initial value.

```
int    x = 100 ;
double  z = 10.0 ;
```

Global Variables / Local Variables

- **Global Variables**: are variables, which are defined in the **outermost** block.
  - They are placed **outside** any function of the program.
  - Such variables can be used in any part of the program in which it is not redefined.
- **Local Variables**: are variables defined **within** a function and is said to be local to that function.

Example 1:

```
#include <stdio.h>
int x;
void anotherfunction() {
   // function body
}
int main(){
   int y;
   // x can be used here
}
```

- Here, **x** has been declared globally and **y** is local to main.
- The variable **x** can be used by the **main** function or any other function aside from **main** (e.g., **anotherfunction**).

Example 2:

```
#include <stdio.h>
int x,z;
void function1(){
    int x;   // x is local to function1
          //global x is inexistent here, only local x
             //z can be used here
}
int main(){
    int z;      // z is local to main
    char c;   // c is local to main but not global
              //x can be used here
             //global z is hidden, local c is active
}
```

• During operation of a function, when the name of the local variable is the **same** to that of the global variable, the global variable is **hidden**.

8. Basic Input/Output Functions

- ❖ All input/output operations in **C** are being performed by special program units called input/output functions.
- ❖ The most common input/output functions are supplied as part of the **C** standard input/output library to which we gain access through the pre-processor directive: **#include <stdio.h>**.
- ❖ These functions then are activated or called by a function call.

The printf function

- To see the results of the program execution, the **printf** function defined in **stdio.h** is used.
- The statement forms:

printf(format string);

printf(format string, print list);

Example 1:

printf("Hello World!\n");

- Displays **Hello World!** in the screen and places the cursor to the next line.
- The complete program:

```
#include <stdio.h>
int main(void) {
        printf("Hello World!\n") ;
        return(0);
}
```

Example 2:

printf("%d degrees Celsius is equal to %d Fahrenheit", cel , fah);

- Assuming both are integer types whose values are **cel = 100**, **fah = 212** the screen output is:

100 degrees Celsius is equal to 212 Fahrenheit

and the cursor is placed after the word Fahrenheit.

Example 3:

printf("Total amount: %f ", amount);

- If amount is 100.0 output would be:

Total amount: 100.000000

- Since amount is a floating-point number, number of decimal places can be specified.

Example 4:

printf("Total amount: %.3f ", amount);

- If amount is **100.14** output would be:

Total amount: 100.140

Example 5:

printf("Value of x: %5d ", x);

- If **x** is 100 the output would be

Value of x: ##100

- The symbol **#** represents a space.
- The integer placed after % indicates the space provided for the value of the variable and if it is greater than the number of spaces the variable occupies, space/s are placed before it.

Example 6:

printf("Total amount: %10.2f ", amount);

- if **amount** is 100.1489 output would be

Total amount: ####100.15

- The integer 10 indicates the number of spaces allocated for **amount** and that includes the integer part, the period and the decimal part.
- Also, if it is lesser than what is needed, it will automatically enlarge and output will just fit.
- **C** is rounding off values if needed as shown above.

Example 7:

```
#include <stdio.h>
float y = 150.2546;
int main(){
    int x = 100;
    printf("Sample");
    printf(" program.\n");
    printf("The value of x: %d and y: %1.2f\n",x,y);
    printf("Bye.");
    return(0);
}
```

The output:
Sample program.
The value of x: 100 and y: 150.25
Bye.

Placeholders in Format Strings

TABLE 2.8 Placeholders in Format Strings

Placeholder	Variable Type	Function Use
`%c`	`char`	`printf/scanf`
`%d`	`int`	`printf/scanf`
`%f`	`double`	`printf`
`%lf`	`double`	`scanf`

The printf Backslash Codes

Code	Meaning
\a	alert (ring bell)
\n	newline (<cr><lf> pair on screen)
\b	backspace
\r	carriage return (<cr>)
\t	tab
****	backslash (two needed)
\'	single quote mark
\"	double quote mark
\0	null

Quick-check Exercise:

❖ Show the exact form of the output line displayed for the given statements. Represent spaces by # symbol:

printf("3 values of x are %4.1f * %5.2f * %.3f\n", x , x , x);

where x = 3.456

3 values of x are #3.5 * #3.46 * 3.456

printf("3 values of x are %4d*%5d*%d\n",x ,x ,x);

where x = 345

3 values of x are #345* ##345*345

printf("%8.2f ", y);

where y = 7.5

Quick-check Exercise:

printf("%8.5f ", x + (2 * y));

where x = 5 and y = 7.5

20.00000

printf("%8.3f ", x * y);

where x = 5 and y = 7.5

##37.500

The scanf function

- The statement form:

  scanf(format string, input list);

- Calls the function **scanf** (pronounced "scanf-eff") is used to copy data **from** the input device **to** the variable/s defined in the input list.

The scanf function

- The format string is a quoted string of placeholders, one placeholder for each variable in the input list.
- Each variable is preceded by an ampersand (**&**), an address-of operator. Commas are used to separate variable names.
- The order of the placeholder must correspond to the order of the variables in the input list.

The scanf function

- When there is more than one input in one **scanf**, they **must** be separated by a space during entry .
- Sample code for **scanf** are shown below:

```
scanf("%d",&x);

scanf("%c %d",&ch,&x);
```

Sample programs using printf and scanf:

```
#include <stdio.h>
int main(){
   int x;
   printf("Enter value for x: ");
   scanf("%d",&x);
   printf("Value entered is: %d",x);
   return(0);
}
```

The output (assume entry is 10):
Enter value for x: 10
Value entered is: 10

Example2:

```
#include <stdio.h>
int main(){
   int x,y;
   printf("Enter value for x and y: ");
   scanf("%d %d",&x,&y);
   printf("x = %d and y = %d",x,y);
   return(0);
}
```

The output :
Enter value for x and y: 10 20
x = 10 and y = 20

8. Operators and Expressions

- An operator is a symbol which represents an operation to be performed.
- Some commonly used operators in **C** are known to be **unary**, **arithmetic**, **relational**, **logical**, and **bitwise** operators.
- On the other hand, an expression is the combination of numbers, variables, and operators that form a statement and has meaning or value.
  - Expressions are needed in manipulating data during programming.

Unary Operators

- The unary operators are: unary plus (+) and unary minus (–), and they require a single operand.
- They usually precede **double** and **int** arithmetic, numbers, variables, or expressions as follows:

+x, –Alpha, –(x+y)

- The unary plus operator causes no change to the quantity which follows, whereas the unary minus operator causes the sign of the following quantity to be changed. Thus, for example, if **x** is 5 and **y** is 2 the expression

–(x+y) results to –7

Arithmetic Operators

- The most commonly used operators for floating-point and integer arithmetic are:

TABLE 2.9 Arithmetic Operators

Arithmetic Operator	Meaning	Examples
+	addition	`5 + 2 is 7` `5.0 + 2.0 is 7.0`
-	subtraction	`5 - 2 is 3` `5.0 - 2.0 is 3.0`
*	multiplication	`5 * 2 is 10` `5.0 * 2.0 is 10.0`
/	division	`5.0 / 2.0 is 2.5` `5 / 2 is 2`
%	remainder	`5 % 2 is 1`

Operators / and %

TABLE 2.10 Results of Integer Division

3 / 15 = 0	18 / 3 = 6
15 / 3 = 5	16 / −3 varies
16 / 3 = 5	0 / 4 = 0
17 / 3 = 5	4 / 0 is undefined

TABLE 2.11 Results of % Operation

3 % 5 = 3	5 % 3 = 2
4 % 5 = 4	5 % 4 = 1
5 % 5 = 0	15 % 5 = 0
6 % 5 = 1	15 % 6 = 3
7 % 5 = 2	15 % −7 varies
8 % 5 = 3	15 % 0 is undefined

Operator Precedence

- In evaluating an arithmetic expression, operator precedence or the order by which they perform should be observed.
- Below is the table showing the precedence of various arithmetic operators.

Precedence	Operator	Description
Highest	+ –	(unary)
	/ * %	
Lowest	+ –	(binary)

Operator Precedence

- Operators on the same line have equal precedence and they are performed proceeding in **left-to-right** order.
- Parentheses **( )** can also be present to expressions. In the case of expressions using parentheses (not nested), the expressions inside the parentheses are evaluated first (**left-to-right**).
- When parentheses are nested (a parenthesis within a parenthesis), expressions are evaluated from innermost to outermost.

Examples

- The following examples give some practice in applying the precedence in evaluating various expressions:
- Evaluate the following expressions:
  - **2 * 5 + 5 * 3 + 4 * 10**
  - **–8 * 5 / 3 * 2**
  - **4 % 2 + 3 – 4 * 5**
  - **x * y – z / 2** **// where x = 5, y = 4, z = 12**

Solution 1:

2 * 5 + 5 * 3 + 4 * 10

- The operator with the highest precedence is *, performing operation from left to right results to

10 + 15 + 40

- proceed addition left to right

25 + 40

- results to **65**.

Solution 2:

–8 * 5 / 3 * 2

- First, applying unary minus to 8 yields

(–8) * 5 / 3 * 2

- Next doing * and / left to right,

–40 / 3 * 2

–13 * 2

- The result is **–26**.
- Note that applying division (/) to **int** values results to an **int** so that –40/3 equals –13.

Solution 3:

4 % 2 + 3 – 4 * 5

- Evaluate % and * left to right since they have equal precedence step by step resulting to

0 + 3 – 4 * 5

0 + 3 – 20

- then proceed to binary + and – (left to right)
- resulting to **–17**.

Solution 4:

x * y – z / 2, where x = 5, y = 4, z = 12

- Substitute the values of the variables first

5 * 4 – 12 / 2

- then do * and / left to right

20 – 12 / 2

20 – 6

- resulting to **14**.

Evaluate the following expressions:

- 5 + 3 * (4 + 2) — 23
- (8 – 5) / 3 * (2 + 6) — 8
- (1 + 7 * (3 + 8)) * (5 – 3) — 156

Quick-check Exercise:

- Evaluate the following arithmetic expressions:

1. 2 * x + y % 2, where x = 3 and y = 2 _____
2. 2 * 3 – 10 % 3 + 6 – 2 _____
3. (12–3) / 2 + (4 % (2 + 1) * 5) _____
4. 25.0 * 3.0 / 2.5 _____
5. 2 + 5 * 2 – 6 / 2 _____

Mathematical Formulas as C Expressions

- Expressions in the program may be mathematical formulas. When writing the formulas in its **C** expression form, consider the following guidelines:
  - Always specify multiplication explicitly by using the operator (*) where needed.
  - Use parentheses when required to control the order of operator evaluation.
  - Two arithmetic operators can be written in succession if the second is a unary operator.

Examples

TABLE 2.13 Mathematical Formulas as C Expressions

Mathematical Formula	C Expression
1. $b^2 - 4ac$	`b * b - 4 * a * c`
2. $a + b - c$	`a + b - c`
3. $\frac{a+b}{c+d}$	`(a + b) / (c + d)`
4. $\frac{1}{1+x^2}$	`1 / (1 + x * x)`
5. $a \times -(b+c)$	`a * -(b + c)`

Increment Operator (++)

- This operator is used to add 1 to the value of the variable. For example, if **x = 5**, the statement can be written as

  x++; (**postfix**) // this means x = x+1

  ++x; (**prefix**) // still x = x+1

- These given statements means the same thing, that is, the value of **x** is added by 1, hence, x = 6.
- But when they are used inside an expression, they are different.

++

- For example, given the variables:

  int i = 5, j = 6, x=0;

- the statement

  ++i;

- increments the value of **i** by 1 or this is equivalent to

  i = i + 1;

++

- However, in the given assignment statement

x = j++; // (postfix)

- first, the value of **j** will be assigned to **x**, then increment **j** by 1. So that after executing the statement the value of **x** = 6 and **j** = 7.
- If the operator is placed before the variable **j**,

x = ++j;

- first increments the variable **j** and assign the result of the expression to **x**. The value of the variables are **x** = 7, and **j** = 7 after the operation.

++

- Another example expression is shown below:

x = i + j++;

- This statement means add **i** and **j**, assign the value to **x** and increment **j**. So, the value of the variables after the statement would be:

x = 11, i = 5, and j = 7.

- If the statement is written as:

x = i + ++j;

- **j** is incremented first, add **i** and **j** and proceed to assignment. Hence, the variables has the values **i**=5, **j**=7, and **x**=12 after executing the statement.

Decrement Operator (--)

- This operator is used to deduct 1 from the variable where it is being applied.
- This works the same way with the increment operator when placed before (**prefix**) or after (**postfix**) the variable in an expression.

Example

- Given the variables:

  int i = 5, j = 6, y=0, z=0;

- The statement:

  y = i- -;

- Assigns **i** to **j**, then decrement **i** by 1.
- The value of each variable after the operation:

  y = 5, and i = 4.

- Also, the statement

  z = - -j;

- decrements **j** by 1, and assign the value to the variable **z**. So, **j**=5 and **z**=5.

- But when placed in an expression, as shown

z = i + j- -;

- the value of each variable would be **z**=11, **i**=5, and **j**=5.
- But if the operator is placed before **j:**

z = i + - -j;

- **j** is decremented first, add **i** and **j** and assign the value to **z**. Thus, **i**=5, **j**=5, and **z**=10 after the assignment statement.

Quick-check Exercise:

- What values are assigned to **n, m**, and **p** given these initial values?

i = 3; and j = 9

1. n = ++i * --j; _____
2. m = i + j--; _____
3. p = i-- + j++; _____

-- ++

❖ If these lines of statement will be executed in sequence, what are the finals values of **i** and **j**?

```
n = ++i * --j;
m = i + j--;
p = i-- + j++;
i++;
j--;
```

i = _____

j = _____

Combined/Assignment Operators

❖ Combined operators are commonly known as "**short hand**" for simple arithmetic and assignment operations. Instead of writing

x = x + y ;

❖ this can be written as

x += y;

❖ Similarly, the assignment statement

x = x + 5;

❖ can be written as

x += 5.

Short Hand

- All binary operators in **C** can be combined. Here are some of the combined operators that can be used in the program.

Combined Expression	Equivalent Long-hand Notation
x += y	x = x + y
x -= y	x = x - y
x *= y	x = x * y
x /= y	x = x / y
x %= y	x = x % y

Examples:

- Given **x** = 20 and **y** = 2

The operations		**equals to**
x+=y	➔	x = 20 + 2
		x = 22 after assignment
x%=y	➔	x = 20 % 2
		x = 0 after assignment

Relational Operators

- Used to **compare** constants, variables, or expressions.
- The use of relational operators result to conditional expressions (**boolean** expressions) which gives the value of **true (1)** or **false (0)**.

Relational Operators

- The following are the relational operators used in **C** that can be used to compare **A** and **B** of any data type:

Operator	Meaning	Example
==	equal	A ==B
!=	not equal to	A != B
>	greater than	A > B
<	less than	A < B
>=	greater than or equal to	A >= B
<=	less than or equal to	A <= B

Examples:

- Here are some examples given that **x** = 10 and **y**=5:

x < y **result: 0 (false)**

x > y **result: 1 (true)**

x >= y **result: 1 (true)**

x == y **result: 0 (false)**

x != y **result: 1 (true)**

Logical Operators

- Used to compare or combine conditional expressions/relational expressions even complex ones.
- There are three logical operators used by C: and (**&&**), or (**||**) and not (**!**).
- If an expression uses one or more of these operators, it is called **logical expression**.

The && Operator (and)

op1	op2	op1 && op2
nonzero (true)	nonzero (true)	1 (true)
nonzero (true)	0 (false)	0 (false)
0 (false)	nonzero (true)	0 (false)
0 (false)	0 (false)	0 (false)

The || Operator (or)

op1	op2	op1 || op2
nonzero (true)	nonzero (true)	1 (true)
nonzero (true)	0 (false)	1 (true)
0 (false)	nonzero (true)	1 (true)
0 (false)	0 (false)	0 (false)

The ! Operator (not)

op1	! Op1
0 (false)	1 (true)
nonzero (true)	0 (false)

Short-circuit Evaluation

- Done to the expression when the operators **&&** and **||** are used.
- When the operator is **&&**, whenever **op1** is **0** (false) the result of the expression can be determined without evaluating the **op2**.
- The **op2** will be evaluated only if the **op1** is nonzero (true).
- In the case of using an operator **||**, when the **op1** is nonzero (true), the expression is determined to be 1 (true) and the **op2** is not evaluated.

Bitwise Operators

- Are used to directly to compare and/or manipulate equivalent bit patterns of data.
- Following are the bitwise operators used in **C** language programming:
  - Bitwise negation.
  - Shift operators.
  - Bitwise **and**, **xor**, and **or**.

Bitwise Negation (~)

- Application of the **~** operator to an integer produces a value in which each bit of the operand has been replaced by its negation– that is, each **0** is replaced by a **1**, and each **1** is replaced by a **0**.
- Using the **n=13** value as shown, we compute **~n** as follows:

n → 00000000 00001101

~n → 11111111 11110010

Shift operators

- The shift operators **<<** (left) and **>>** (right) take two integer operands:
  - The value of the left operand is the number to be shifted and is viewed as a collection of bits that can be moved.
  - The right operand is an nonnegative number telling how far to move the bits left or right depending on the operator used.

Examples

Left Shift Operator (<<)

2 << 3 // shifts the bits of 2 three bits to the left.

= (00000000 00000010) << 3

= (00000000 00010000)

= 16

Right Shift Operator (>>)

7 >> 2 // shifts the bits of 7 two bits to the right.

= (00000000 00000111) >> 2

= (00000000 00000001)

= 1

Bitwise and, xor, and or

- The bitwise operators **&** (and), **^** (xor), and **|** (or) all take two integer operands that are viewed as strings of bits.

The bitwise operator & (and)

- The operator is being applied to each pair of bits.
- Considering the **i**th bit of the operands **n** and **m**, the result will be given according to the truth table given below:

operand ni	operand mi	ni & mi
1	1	1
1	0	0
0	1	0
0	0	0

Example:
4 & 5 = (00000000 00000100) & (00000000 00000101)
= (00000000 00000100)
= 4

The bitwise operator | (or)

operand ni	operand mi	ni | mi
1	1	1
1	0	1
0	1	1
0	0	0

Example:
4 | 5 = (00000000 00000100) | (00000000 00000101)
= (00000000 00000101)
= 5

The bitwise operator ^ (xor)

operand ni	operand mi	ni ^ mi
1	1	0
1	0	1
0	1	1
0	0	0

Example:
4 ^ 5 = (00000000 00000100) ^ (00000000 00000101)
= (00000000 00000001)
= 1