Computer Programming

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

The C Language Basic

C Language Elements

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

Some Definitions

- Statement - instruction for the computer to perform, usually ends in a semicolon (;).
- Variable - name given to a memory location that stores data that may change
- Constant - data that does not change

Programming Errors

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

Special Characters

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

Comments

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

Programming Process

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

Example

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

- Check Logic - does this algorithm fulfill the purpose.
- Write code, enter the code, and compile it.

Variables and Constants

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

Identifiers (Variables)

- Identifiers are names (or symbols) used by the programmer to refer to items such as variables, constants, functions.
- Identifiers should be descriptive of what they stand for
- The “Name” used for identifiers must follow specific guidelines for C++ to be valid.
- Identifier Naming Rules:

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

Valid and Invalid Names

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

Data Types

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

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

Key words for Data Types

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

Integers: Example (decleration)

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

Floating Points:

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

Symbols(letters):

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

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

Declarations

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

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

Assignment Statement

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

  identifier = expression

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

Assignment statement (2)

- The assignment statement can be used to initialize variables. Examples are:
- int num1 = 15; // initialize to constant value
- int num2 = num1; // initialize to variable
- char char = ‘A’; // initialize to character
- Note the use of quotations with charters to differentiate it from variables
- double sum = 13.2; // initialize double values
- int x = 13.2; // store only 13
- int z = ‘A’; // converts char to int stores 65 in z.
- double f = 12; // stores 12.0 in to f

Input Statement

- Allows data entered by the keyboard to be stored in variables.
- The general syntax is

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

- Examples are:
- int abc; cin >> abc; where an integer value read from the keyboard and stored in memory location abc.
- Can enter multiple values in one statement.

  cin>>length>>width;
- cin skips all white spaces blanks, newlines, and tabs.

  example cin >> x>>y; skips all spaces between x and y.
- cin requires the use of the pre-processor directive

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

Output Statement

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

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

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

  the house

  is full

Use endl to start printing on the next line.

Expression Definition

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

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

Expression Types and Values

- Expression may contain many simple, complex expressions and many operators applied on these expression that results in a single value.
- An expression can be either of two types: <u>Arithmetic or Logical</u>
- Arithmetic expression applies an arithmetic operator to an operands (expressions) (+,-,*,%, … more later)
- Logical expression applies logical operator to an operands (expressions) (>, < ,&& ||, more later)

- The Final value of an expression is either logical or arithmetic depending on last operator executed. If the last operator is logical the expression final value is either "true" 1 or "False" 0. If the last operator is arithmetic the expression final value is a arithmetic (integer or double)
- For any expression if the arithmetic value is zero its logically "false" otherwise it is logically "true".

Expression Evaluation

operand means the integer or floating-point constants and/or variables in the expression.

There are two kinds of numeric values

- Integers (0, 12, -17, 142)
- Floating-point numbers (3.14, -6.023e23)

Operators are things like addition, subtraction multiplication, greater than and less.

The value of an expression will depend on the data types and values and on the operators used

Additionally, the value assigned to a variable in an assignment statement will also depend on the type of the variable.

Arithemtic Operators

- Operators can be combined into complex expressions

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

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

- Operator types:

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

Unary: operates on one operand: *-23.4*

Sample Expressions

- Operators on doubles:

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

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

  Sample expressions:

  – 0.4 * income - children * 500

  – (A4.0 / 3.0) * 3.14 * radius * radius * radius

- Operators on integers:

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

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

  Sample expressions:

  – 5 + 4 * 2

  –Int x =10; x/2

int Division and Remainder

Integer operators include

integer division (/) and

integer remainder (%):

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

$299 / 100 \longrightarrow 2$

$6 / 4 \longrightarrow 1$

$5 / 6 \longrightarrow 0$

% is mod or remainder:

$299 \% 100 \longrightarrow 99$

$6 \% 4 \longrightarrow 2$

$5 \% 6 \longrightarrow 5$

A Cautionary Example

```
int radius;
double volume;
double pi = 3.141596;
volume  = ( 4/3 ) * pi * radius  *radius * radius;
 Result is (1) *  pi * radius  *radius * radius;
   result is 3.141596 * radius  *radius * radius
val= (3/4)* radius
 result is  0 * radius = 0.0 * radius
```

Order of Evaluation

Precedence determines the order of evaluation of operators.

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

And does it matter?

Try this:

4 + 3 * 2 - 1

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

4 + (3 * 2) - 1 = 9 9

Operator Precedence Rules

Precedence rules:

- –1. do ( )'s first, starting with innermost
- –2. then do unary minus (negation): -
- –3. then do "multiplicative" ops: *, /, %
- –4. lastly do "additive" ops: binary +, -

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

Associativity Matters

- Associativity determines the order among consecutive operators of equal precedence

- Does it matter? Try this: 15 / 4 * 2

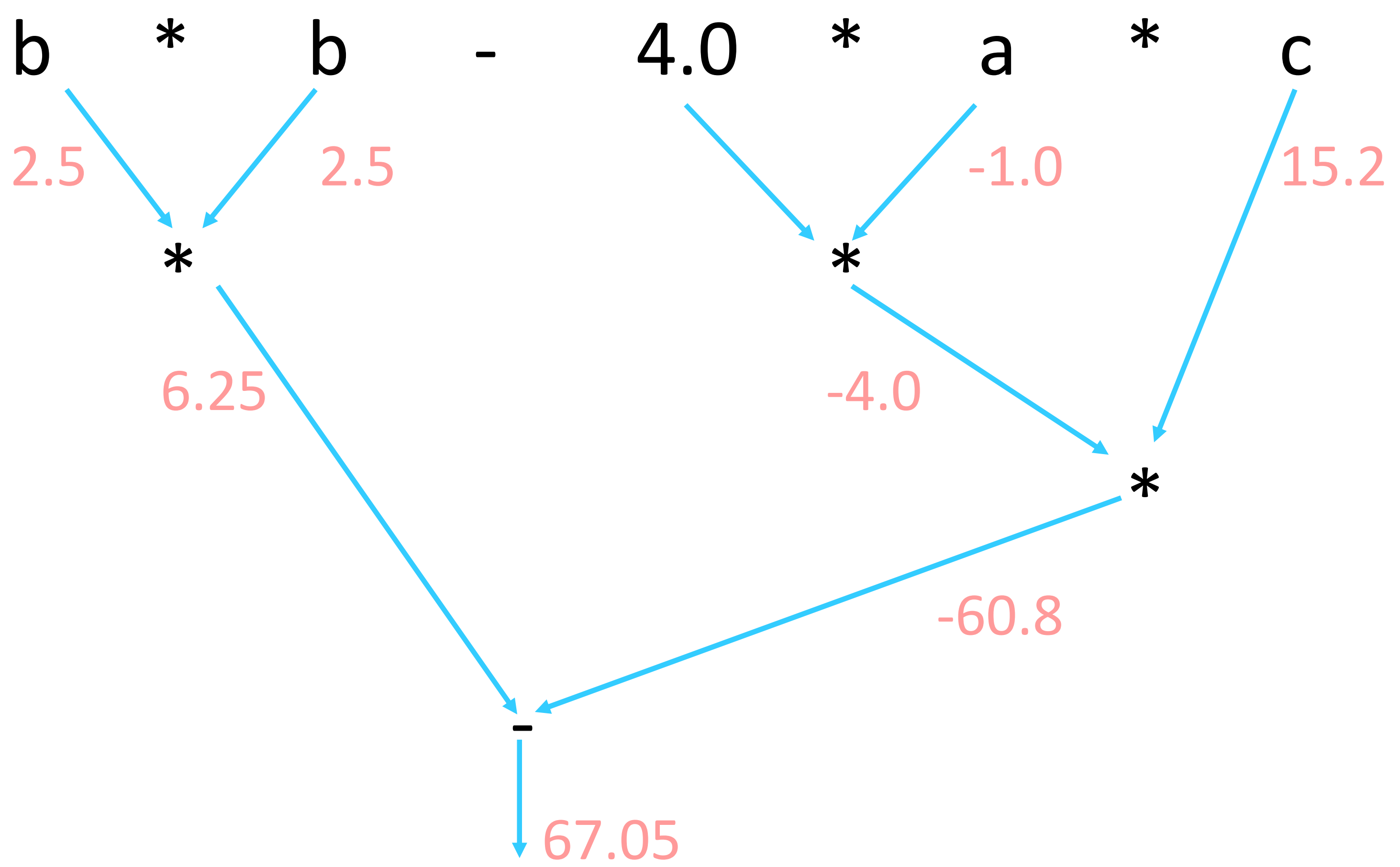
  (15 / 4) * 2 = 3 * 2 = 6

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

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

Depicting Expressions

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



Data Conversions

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

Data Conversions

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

Data Conversions: Casting

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

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

Conversions in Assignments

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

implicit conversion to double

implicit conversion to int – drops fraction with no warning

Explicit Conversions

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

Format:	(type) expression
Examples	*(double) myage*
	(int) (balance + deposit)

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

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

Using Casts

```
int total, count ;
double avg;
total = 97 ;    count = 10 ;

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

Advice on Writing Expressions

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

Relational Operators

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

Relational Operators

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

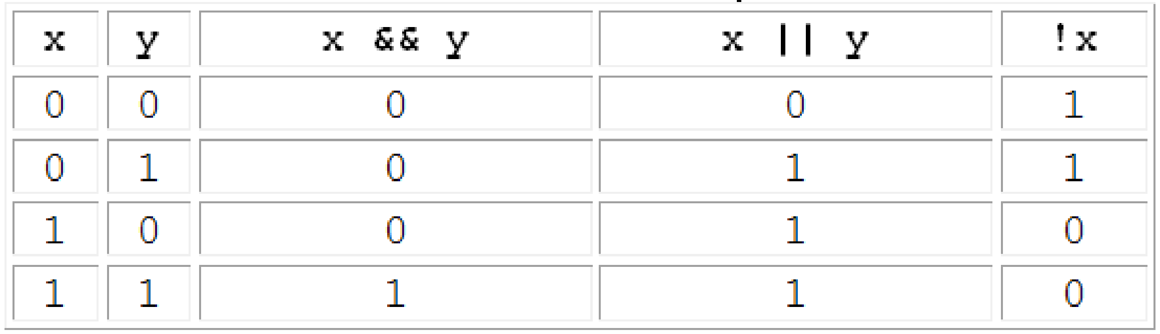
>	Is greater than
<	Is less than
> =	Is greater than or equal to
< =	Is less than or equal to
= =	Is equal to // This is a mathematical equals
! =	Is not equal to // An exclamation point means not in C++

Logical Operators

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

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

The following table summarizes these operators



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

Examples:

```
finalScore > 90 && midtermScore > 70

midtermGrade == 'A' || finalGrade == 'A'

!(hours > 40)is equivalent to hours <= 40
```

Short Circuits && and ||

- Short circuit evaluation looks at a compound expression and evaluates it until it reaches a conflict a final result of the expression

- A long expression filled with ANDs

  p && q && r // where p, q, r are boolean expressions

  if p is false, then the expression is false and therefore, the evaluation will stop at p, if p is true q is evaluated and so on. A series of ANDs will stop being evaluated when a false is reached

A long expression filled with Ors

p || q || r // where p, q, r are boolean expressions

- if p is true, then the expression is true and therefore, the evaluation stop there if p is false q is evaluated and so on. A series of ORs will stop being evaluated when a true is reached

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

Logical and Arithmetic Operator Precedence

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

Mixing Expressions

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

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

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

  <u>z = x +(3 < y).</u> The last operator evaluated is + since 3 < y is zero. Then z = x+0. This is an arithmetic expression and in 5 is stored in z.

Assignment Revisited

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

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

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

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

Short Hand Operators

Syntax:

Variable Op.= Expression;

Evaluated as

Variable = Variable Op. (Expression);

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

Increment and Decrement Operators

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

Fig. 4.13 The increment and decrement operators.

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

Difference between Pre and Post

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

<u>Pre</u>

```
int number = 5;  // declares number to be 5
++number; // increments number to 6.
```

<u>Post</u>

```
int number = 5;  //declares number to be 5
number++;  // increments number to 6
```

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

<u>Pre</u>

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

<u>Post</u>

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

Precedence and Associativity

high

Operators	Associativity	Type

`()`
`++ --`
	left to right
	right to left
		parentheses
		unary postfix

`++ -- + -` *(type)*	right to left	unary prefix
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`= += -= *= /= %=`	right to left	assignment

low

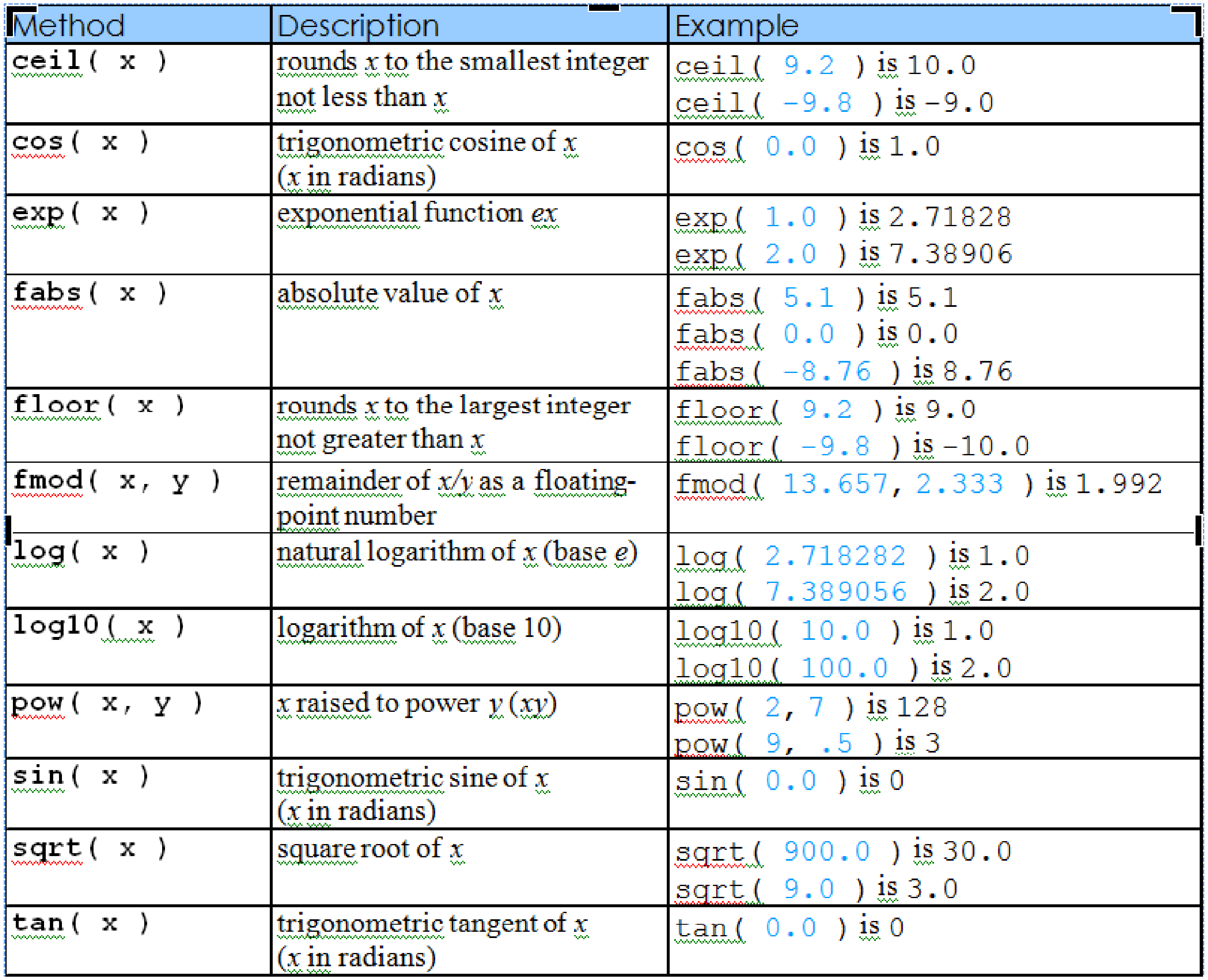
Math Functions

- An expression in C/C++ may need to perform a mathematical functions.
- C provides build in functions to perform common operations.
- To use these functions we must insert **#include <math.h >** at the of program file.

- These functions are called by writing **functionName (argument);** or
  functionName(argument1, argument2, …);
- Example**cout << sqrt(900.0);** **//** would print 30

- All functions in math library return a **double.**
  Function arguments can be
  - Constants: sqrt(4);
  - Variables: sqrt(x);
  - Expressions: sqrt(sqrt(x)) ; or sqrt(3 - 6x);

The following table contains the most popular functions:



Method	Description	Example

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0`
		`ceil( -9.8 )` is `-9.0`

`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`

`exp( x )`	exponential function *ex*	`exp( 1.0 )` is `2.71828`
		`exp( 2.0 )` is `7.38906`

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

`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0`
		`floor( -9.8 )` is `-10.0`

`fmod( x, y )`	remainder of *x/y* as a floating-point number	`fmod( 13.657, 2.333 )` is `1.992`

`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0`
		`log( 7.389056 )` is `2.0`

`log10( x )`	logarithm of *x* (base 10)	`log10( 10.0 )` is `1.0`
		`log10( 100.0 )` is `2.0`

`pow( x, y )`	*x* raised to power *y* (*xy*)	`pow( 2, 7 )` is `128`
		`pow( 9, .5 )` is `3`

`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0`

`sqrt( x )`	square root of *x*	`sqrt( 900.0 )` is `30.0`
		`sqrt( 9.0 )` is `3.0`

`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`