CSCI291

Programming for Engineers

Lecture 1: Overview of the C language

Content:

I. Introduction: Software Development Cycle

II. Basic C Program structure

II.1. Comments

II.2. Pre- processor Directives

II.3. C Functions

II.4. Variables

II.5. Statements

II.6. Formatted Input/Output

II.7. The return Statement

II.8. Operators

II.9. Expressions

Appendix

Please check the note sections for further details

Acknowledgment: these slides are amendment to Dr Obada Al Khatib 2019 notes.

Introduction: Software Development Cycle

- Before starting to write your code, you need to follow these steps:
  1. Analyse carefully the problem definition to specify unambiguously the requirements of the system
  2. Design an algorithm solution to implement the requirements of the system
  3. Document your design solution using pseudo code/ flowchart
  4. Translate your design description into code

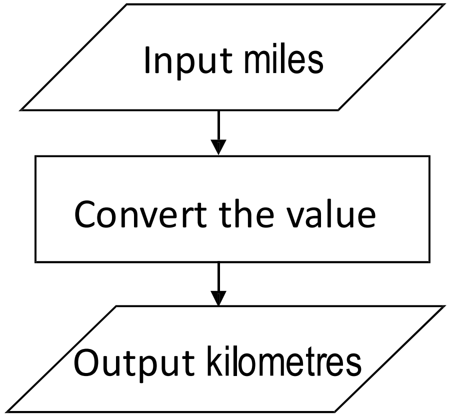
- Example: Write a program that converts distance measured in miles into kilometres

1. Design solution

Input: `miles`

Output: `kilometres`

Algorithm: `kilometres = 1.609*miles`



Code in Matlab

```
% MATLAB - Convert distance
miles = input(‘Enter the distance in miles > ’);

kilometres = 1.609 *miles;

fprintf(‘That equals to %.3f km\n’, kilometers);
```

II. Basic C Program structure

- The below program gives the basic structure of a program in C

C Code of slide 1 problem

Comments

Pre-processor directives

main function

```
/* C program – convert distance  */

#include <stdio.h>
#define KMS_PER_MILE 1.609 /*conversion constant*/

int main(void)
{
    float miles, klms;
    /* Get the distance in miles */
    printf("Enter the distance in miles > ");
    scanf("%f", &miles);

    /* Convert the distance to kilometers */
    klms = KMS_PER_MILE * miles;

    /* Display the distance in kilometers */
    printf("That equals %.3f km\n", klms);

    return (0);
}
```

II.1: Comments

- Comments help to understand/explain the code
- In C, they begin with `/*` and end with `*/` **and can span multi lines**

```
/*________________
      A function that calculates the area of a rectangle
       It takes two parameters of type float
      and returns a value of type float
__________________*/
float calculateArea (float width, float height)
{
     float  rectArea;                /* the area of a square */

     rectArea = width * height;      /* calculate the area */
     return rectArea;
}
```

Introductory comments.
The underlined text is extra!

Descriptive comments

- Nested comments are not supported

start

```
/* calculate the area of a square
/* input - width and length */
   and return a float type value */
```

end

- To comment a single line, you can prefix it with //

II.2. Pre-processor Directives

- A **pre-processor** is a program that processes our source program before it is passed to the compiler

- The pre-processor works on the source code and creates an expanded source code!

- The pre-processor offers several features called **pre-processor directives**
- Each of these pre-processor **directives** begin with **a #symbol**
  - No **;** at the end of line

- **The directives can be placed anywhere in a program but most often placed at the beginning of the program, before the function definition**

```
#include <stdio.h>
#include <math.h>
#define PI    3.1415926
#define MAX_LENGTH   200

int main(void)
{



}
```

#include directive

Syntax:

```
        #include <a header file name>
```

Example:

```
        #include <stdio.h>
        #include <math.h>
```

- `stdio.h` contains definitions of standard input/output functions `scanf(…)` and `printf(…)`
- `math.h` contains definitions of math functions such as `sin()`, `cos()`, `log()`, `exp()`, etc.

- Include files (header files)

  #include <*filename*>

  Causes the preprocessor to look for *filename* in system defined places and replace the #include line with a copy of contents of *filename*.

  #include "filename"

  Same as above, but the preprocessor looks in the current directory before looking in the system defined directory locations.

#define directive

Syntax:

```
        #define NAME value
```

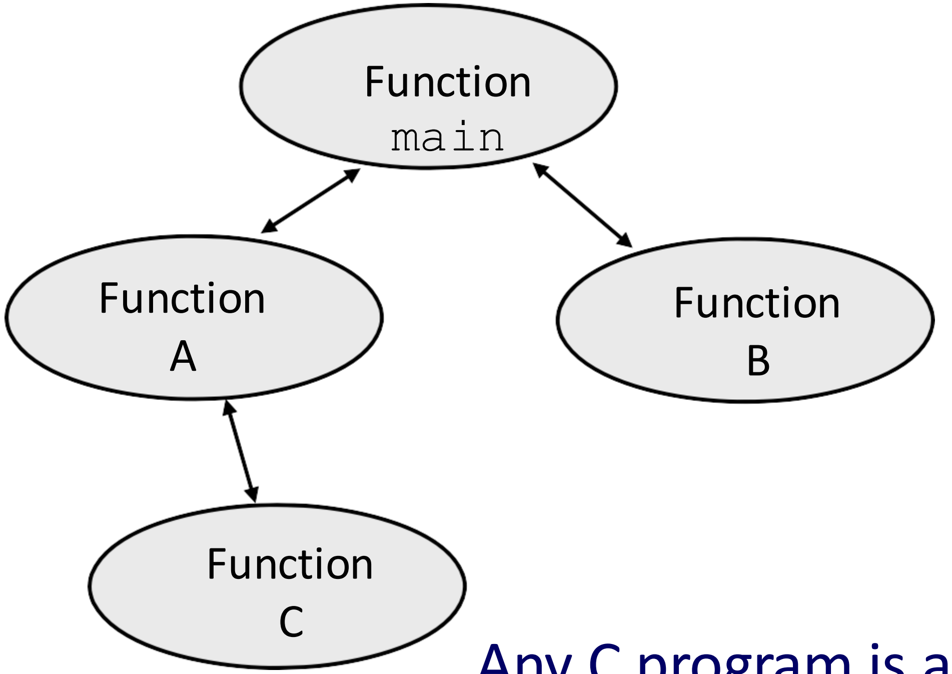
Example:

```
        #define PI   3.1415926
        #define MAX_LENGTH   2000
```

- You can use the **symbolic constants** instead of actual values (makes C code more readable and simplifies further modification)

II.3 C Functions

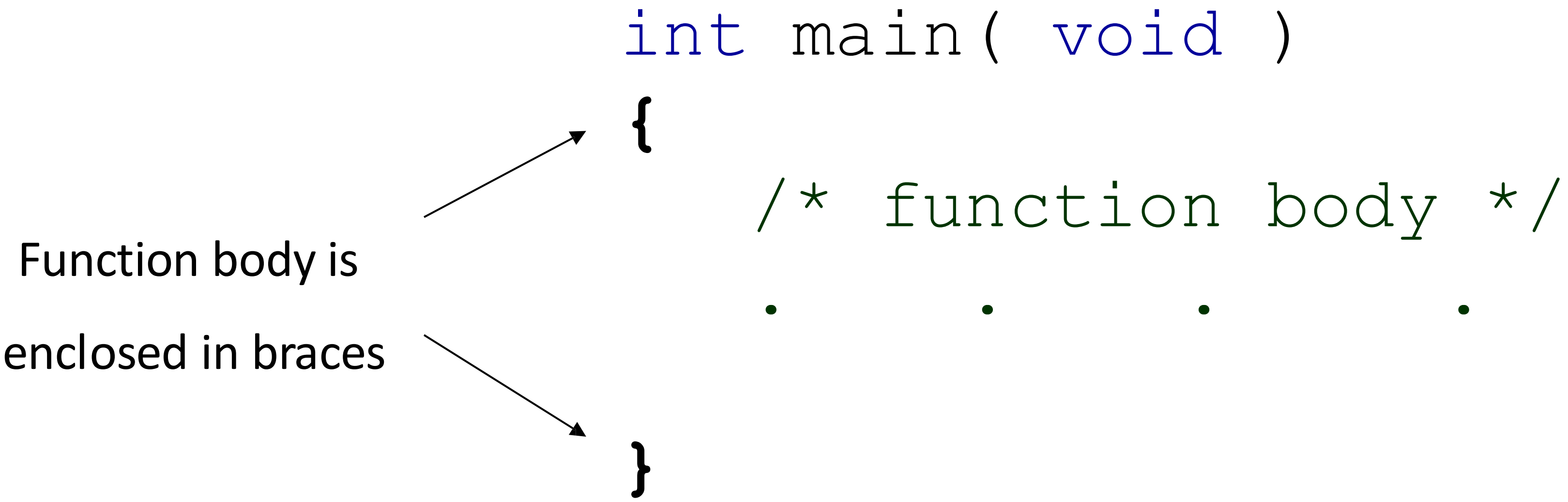
- C language is based on the concept of **Structured Programming.**
- The building blocks of C programs are **functions**
- Types of functions
  - **`main` function**
  - **standard functions**
  - **user-defined functions**



Any C program is a collection of interacting functions

Function `main`

- Every program has one `main(…)` function
- This is the first function to be executed when you run a program.
- There is only one main function in a program



```
int main( void )
{
     /* function body */
     .     .     .     .

}
```

Function body is

enclosed in braces

II.4 Variables

- To **process data** you need **to store them somewhere in memory**
- In order to **reserve memory space,** you **need to declare/ define variables.**
- When a **variable is declared**, a **memory location is reserved for this variable.**
- **Variable is a named location in memory.**

- In MATLAB, memory is reserved when you use a variable
- In C, memory is reserved when you define/declare a variable

```
#include <stdio.h>

int main(void)
{
      int age;
      float height;

      height = 1.81;
      age = 23;
      ~~weight~~ = 65;


}
```

You cannot use a variable if
It has not been declared

Fundamental data types

integral	**char**	1. a character
		2. a small integer number
			`char unit=‘m’;`
			`char errCode = 4;`

	int	An integer number	`int year = 2016;`

floating point	**float**	a single-precision real number	`float area = 12.54;`
			`float mass =1.7e+05;`

	double	a double-precision real number	`double weight=2.395;`
boolean	**bool**	true or false	`bool  flagOn=true;`
	void	typeless and valueless	*mainly used for function parameters & return*

- **Sizes and ranges of the fundamental data types** are **platform dependent** even when you use ANSI / ISO C.
- Actual memory size of a type can be obtained using the operator **sizeof(*type*)**
- Ranges are specified as constants in `<limits.h>` and `<float.h>`

An integer number is is a whole number can be positive, negative, or zero

Example: sizeof(type)

```c
/* Display size and range of C data types */

#include <stdio.h>
#include <limits.h>
#include <float.h>
#include <stdbool.h>

int main(void)
{

    printf("char size = %d, Min=%d, Max=%d\n", sizeof(char),  CHAR_MIN, CHAR_MAX);
    printf("int size = %d, Min=%d, Max=%d\n", sizeof(int),INT_MIN, INT_MAX);
    printf("float size = %d, Min=%e, Max=%e\n", sizeof(float), FLT_MIN, FLT_MAX);
    printf("double size = %d, Min=%e, Max=%e\n", sizeof(double), DBL_MIN, DBL_MAX);
    printf("bool size = %d \n", sizeof(bool) );

    return (0);
}
```

UNIVERSITY
OF WOLLONGONG
IN DUBAI

Size and Range (32/64 Windows)

Type	Size (bytes)	Min	Max
`char`	1	CHAR_MIN -128	CHAR_MAX 127
`int`	4	INT_MIN -2147483648	INT_MAX 2147483647
`float`	4	FLT_MIN 1.18e-38	FLT_MAX 3.40e+38
`double`	8	DBL_MIN 2.23e-308	DBL_MAX 1.80e+308
`bool`	1	-	-

A memory block of this size is allocated when you declare a variable

char Data type

Simple data type `char` can represent:

1. A whole number in the range: [-128...127]

```
char date = 24;
char hours = -97;
```

2. A text character:

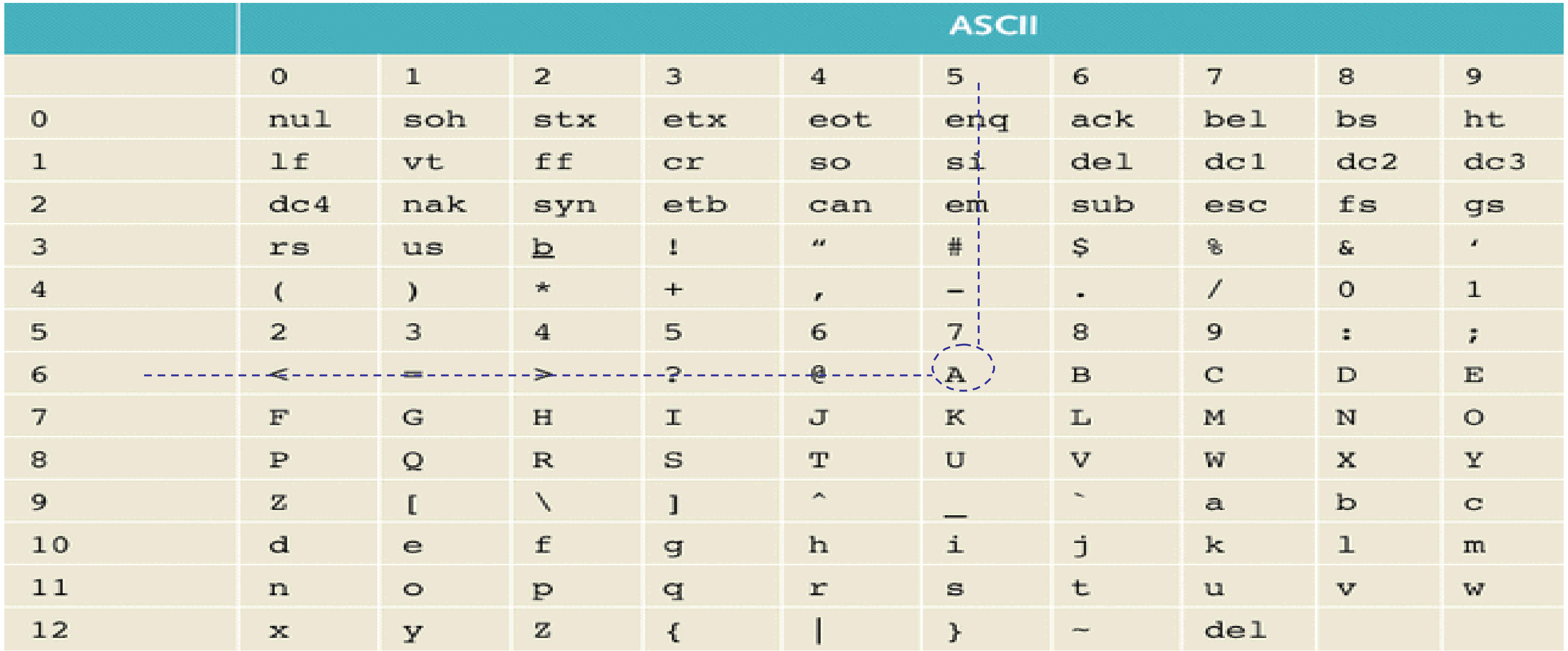
```
'A', 'c', '#', '+', 'S'
char grade = 'A';
char pattern = '#';
```

Each text character is ASCII coded

```
'A' = 65, 'd' = 100, '#' = 35, '2' = 50, '7' = 55
```

(see Appendix A , J. R. Hanly, "C Program Design for Engineers")

ASCII Codes



	ASCII									
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	del	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	b	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Example: ASCII code for ‘A’ is 65

ASCII code for ‘1’ is 49

Special ASCII codes

‘\b’ – backspace (bs) **‘\t’ – horizontal tab(ht)** **‘\n’ – new line (lf)**

‘\r’ – return (cr) ‘\v’ – vertical tab (vt) ‘\0’ – null (nul)

II.5 Statements

Any C program is made of functions. What are functions made of?

- C functions are composed of **statements.**
- A statement expresses an **action/command to be carried out**:
  1. **non-executable statements** : **actions which are NOT translated into microprocessor instructions**
  2. **executable statements :** **actions which are translated into microprocessor instructions**
- The semicolon (**;**) is a **statement terminator**

Example:

int age**;** this is a **non-executable statement** that causes a memory allocation action for the variable `age`

#define MAX_VOLTAGE 12 this is **not a statement.** This is a definition of a symbolic constant MAX_VOLTAGE that will not be placed in memory

Non-executable statements:

Definition of variables

Syntax:

```
        datatype variable_identifier ;
```

Example:

```
        int numOfDays, iterationCounter=0;
        float inputVoltage=0.35, output1, output2;
        char grade='D';
```

- C language requires that all variables be given a data type and an identifier (name)
- Data Types determine:
  - how much memory is allocated to store a variable
  - what are legal values that can be assigned to variables
  - what kind of operations are allowed with this variable
- A valid identifier must
  - consist of only letters, digits, or underscores
  - not begin with a digit
  - not be a C-reserved word

~~#ofItems~~ ~~low-rate~~

~~4thElement~~

~~switch~~

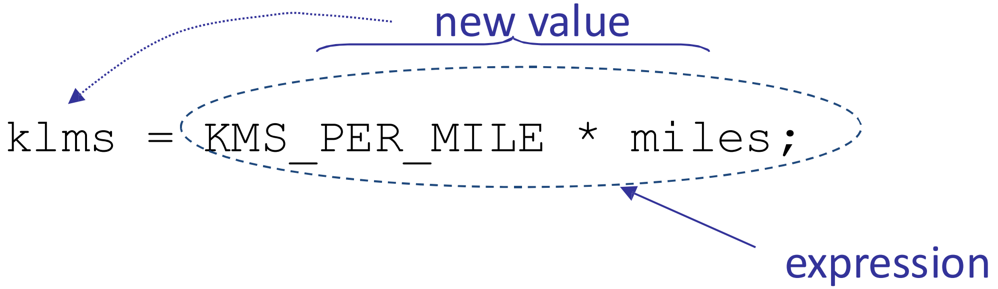
- Variables can be initialized at the time of declaration

```
 float speed=0.5;
```

Executable statements

There are several types of executable statements in C

- Expression statement :



new value

```
klms = KMS_PER_MILE * miles;
```

expression

- Null statement : ;
- Function call : `printf("D equals %.3f km\n", klms);`
- Execution control :

selection (`if…else, switch`)

repetition (`for, while, do…while`)

return from a function (`return 0;`)

. . . .

Expression statements

Syntax:

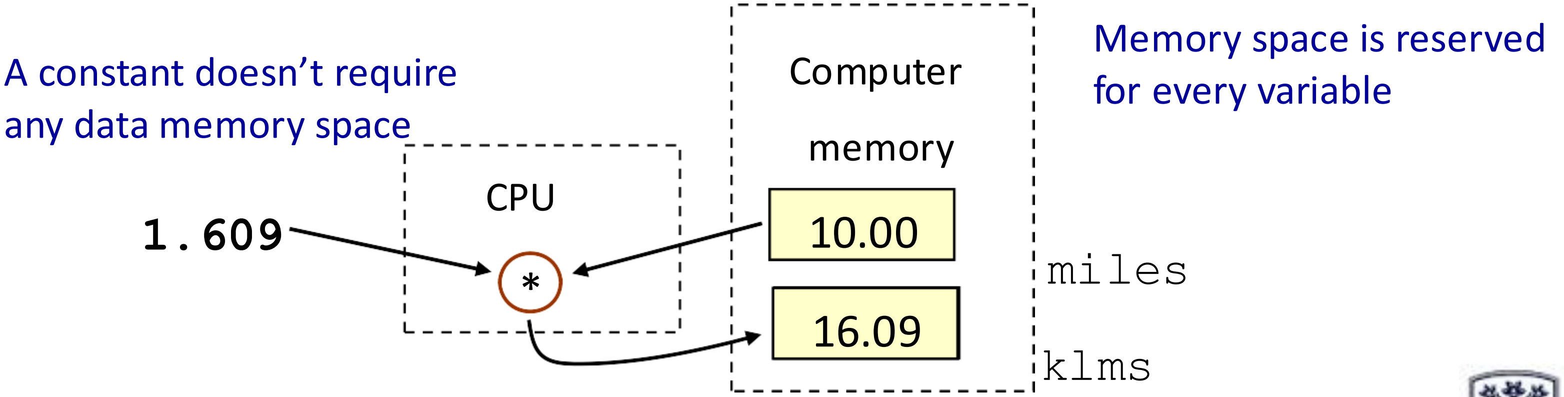
```
        variable = expression;
```

Examples:

```
        klms = KMS_PER_MILE * miles;
        area = calculateArea(radius);
        x = -x;
         y += 5;
```

```
#define KMS_PER_MILE 1.609

float miles=10.0, klms;
klms = KMS_PER_MILE * miles;
```



Quiz 1

Find a bug in this C code and explain how this code can be interpreted using C syntax

```
int a, b, c;

a = 10;
b = 5 + a
c = b * 2;
```

b = 5 + a

Semicolon is missing

Quiz 2

- What is the difference between these two definitions?

```c
char a = '1';
```

1 here is a character

```c
char b = 1;
```

1 here is an ASCII code

- What is the output?

```c
char value = 82;

printf("%d \n", value);
```

82

```c
printf("%c \n", value);
```

R

- What is wrong with this definition?

```c
char numOfDays = 230;
```

Out of char range (-128,127) - Warning

II.6 Formatted Input/Output

- Input/output is carried out using functions defined in *stdio.h* header file

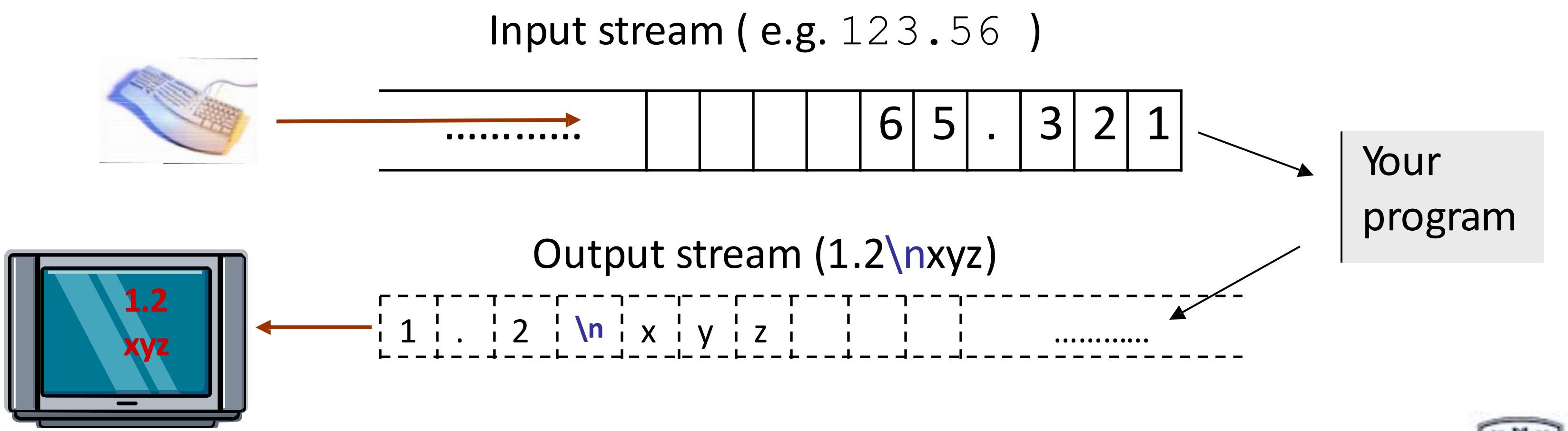
  `scanf(…)` – input from the keyboard

  `printf(…)` – output to the screen

- To use these functions in your program you need to include *stdio.h* header file

```
#include <stdio.h>
```

- These functions provide interface with input/output devices through **buffers:**



II.6.a printf() Function

- This C function looks a lot like `fprinf()` in MATLAB Syntax:

```
    printf(“format string” [,list_of_values])
```

Examples:

```
    printf(“Hello World !\n”);
    printf("Enter the distance in miles> ");
    printf("That equals %f kilometers \n", kms);
```

- **Format specification**

```
General Syntax:
%[flags][width][.prec][hlL]type
```

Fields in bracket are **optional**. If they are used, they must appear in the order shown

Format Specification: type

type	Interpretation
`d` or `i`	Integer
`f` or `F`	floating point number, to six decimal places by default
`e` or `E`	floating-point number in exponential format (e or E places before the exponent)
`c`	single character
`s`	**null-terminated** character string

printf() Function

%[flags][width][.prec][hlL]type

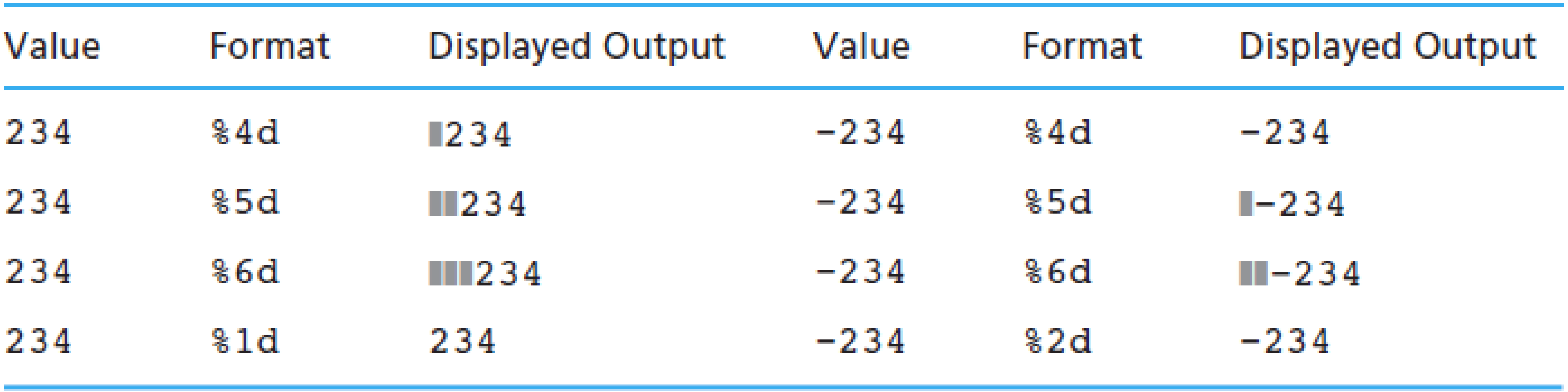
flag	Meaning
–	left-justified within the given field width
+	precede with + or -
0	Left-pads the number with zero if there is a space
space	Print a space before a **positive value.** It is not printed with the + flag

width & prec	Meaning
width	minimum size of field to print the value
.prec	minimum number of digits to display for d or i. number of decimal places for e or f formats. maximum number of characters for s.

Examples

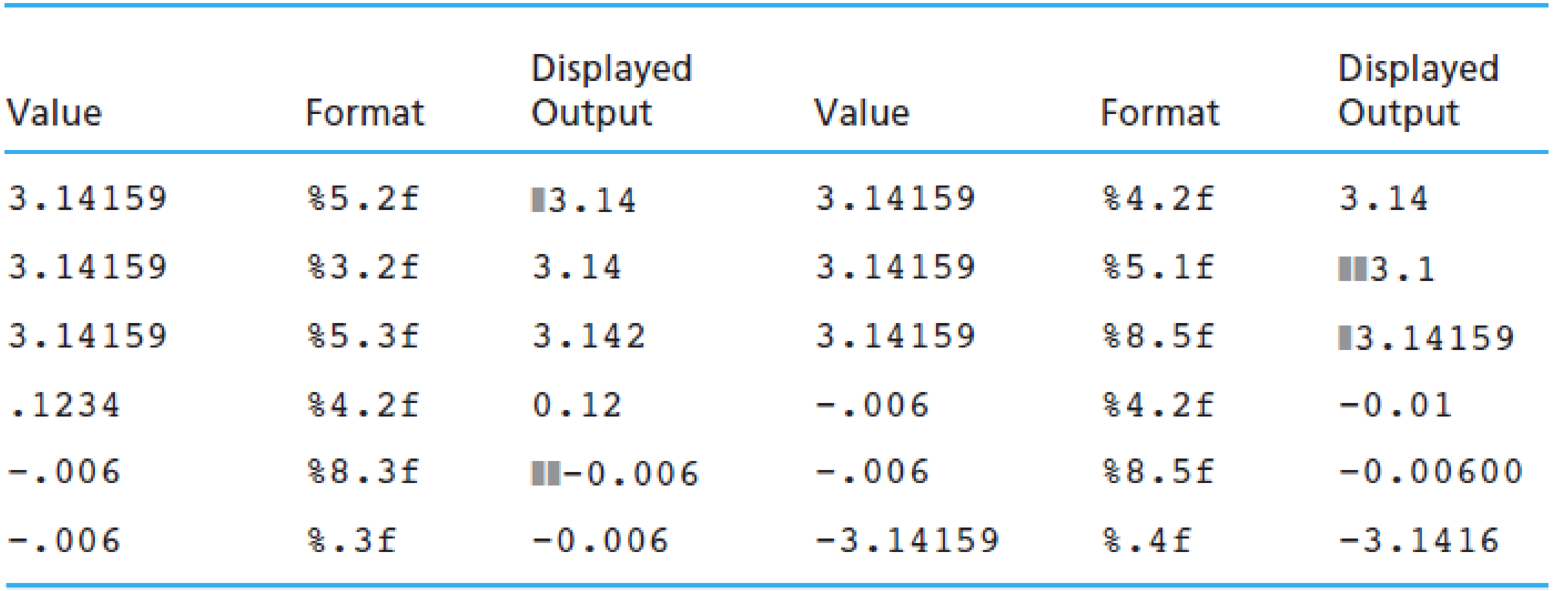
```
%[flags][width][.prec][hlL]type
```

TABLE 2.14 Displaying 234 and –234 Using Different Placeholders



Value	Format	Displayed Output	Value	Format	Displayed Output
`234`	`%4d`	`▮234`	`-234`	`%4d`	`-234`
`234`	`%5d`	`▮▮234`	`-234`	`%5d`	`▮-234`
`234`	`%6d`	`▮▮▮234`	`-234`	`%6d`	`▮▮-234`
`234`	`%1d`	`234`	`-234`	`%2d`	`-234`

TABLE 2.16 Formatting Type double Values



Value	Format	Displayed Output	Value	Format	Displayed Output
`3.14159`	`%5.2f`	`▮3.14`	`3.14159`	`%4.2f`	`3.14`
`3.14159`	`%3.2f`	`3.14`	`3.14159`	`%5.1f`	`▮▮3.1`
`3.14159`	`%5.3f`	`3.142`	`3.14159`	`%8.5f`	`▮3.14159`
`.1234`	`%4.2f`	`0.12`	`-.006`	`%4.2f`	`-0.01`
`-.006`	`%8.3f`	`▮▮-0.006`	`-.006`	`%8.5f`	`-0.00600`
`-.006`	`%.3f`	`-0.006`	`-3.14159`	`%.4f`	`-3.1416`

`II.6.b scanf()` function

- This C function looks a lot like `fscanf()` in MATLAB

Syntax:

```
        scanf("format string", list_of_variables );
```

Example:

```
        int width, height;
        scanf("%d%d", &width, &height);
```

- scanf(…) gets values from the input stream and stores them into variables
- data in the input stream must match the order of the variables in the list
- Each variable, if not a pointer (more later), must be preceded by **&**

keyboard input: `200 300` *Enter*

after scanf(…) `width = 200, height = 300`

keyboard input: `150 24` *Enter*

after scanf(…) `width = 150, height = 24`

Format Specifier

Format string: %[modifier]type

type	Action
d	read an integer expressed in decimal notation; **int**
e,f	read a float in floating-point or exponential notation (e.g. 3.45e-3); **float**
c	read single character even a *whitespace* character; **char**
s	read a sequence of characters, the sequence begins with first non-whitespace character and is terminated by the first whitespace character;

…Continued

Format string : %[modifier]type

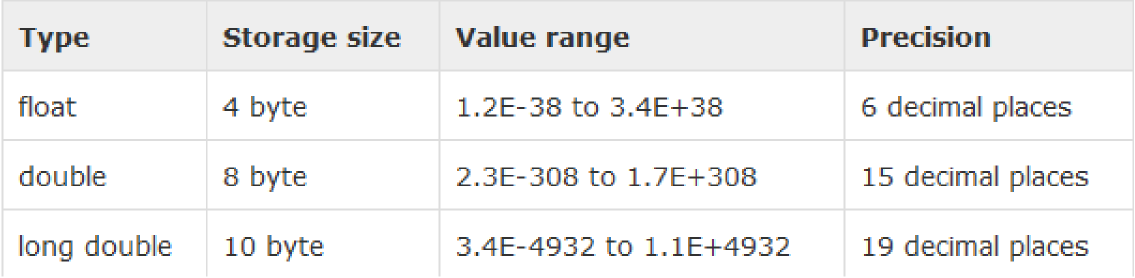
modifer	Meaning
*	field to be skipped and not assigned (read and ignore)
size	maximum size of input field

```
double weight;
scanf("%d", &weight);
```

Wrong

```
float weight;
scanf("%f", &weight);
```

Right



Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Whitepace Characters

Except with %c format, `scanf(…)` **bypasses any leading** *whitespace* characters.

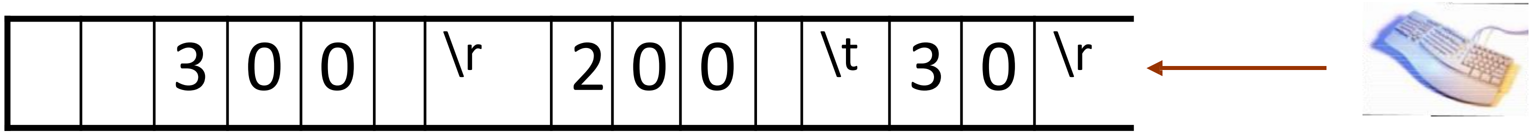
whitespace characters are: **blank_space**,

tab ('\t'),

return ('\r'),

new_line ('\n')

input stream buffer



		3	0	0		\r	2	0	0		\t	3	0	\r

```
scanf("%d%d%d", &w, &h, &d);
```

w=300, h=200, d=30

Quiz 3

`scanf(…)`	Input	Results
`(“%d%f”, &x, &y)`	`250 350.0`	x= y=
`(“%d%c”, &x, &y)`	`29 w`	x= y=
`(“%d %c”, &x, &y)`	`29 w`	x= y=
`(“%d %*f %d”, &x, &y)`	`144 736.54 288`	x= y=

Caution: Any white space in front of `%c` tells `scanf` to skip whitespaces stored in the buffer

Results
x = 250 y= 350.000000
x = 29 y= white space
x = 29 y= w
x = 144 y= 288

II.7. The return Statement

- A **return** statement ends the execution of a function, and **returns** control to the calling function.

- Execution resumes in the calling function at the point immediately following the call.

- A **return** statement can **return** a value to the calling function

II.8. Operators: Assignment operator

```
lvalue = rvalue;
```

- Assigns `rvalue` to `lvalue`
- `lvalue: a` **`variable`**`, i.e. a reference to a memory location.`
- `rvalue` does not need to have a location in memory and **can be a constant, a variable, or a value produced by an expression**
- Unlike lvalue, **rvalue can’t appear at the left of the operator (=)**

Example:

```
int number = 25;         // OK
number = number + 10;  // OK
25 = number;           // ERROR
(number+1) = 25;       // ERROR
```

A copy of the lvalue `number` is created as an rvalue, rvalue+rvalue produces an rvalue that is assigned then to an lvalue `number`

“number+1” is an expression and not a variable

II.8. Operators: Arithmetic operators

There are two types of operators in C

- binary – needs two operands : `+, -, *, /, %`
- unary – only needs one operand: `+, -, ++, --`

operators		meaning	examples	
unary	+	no change	`+x, +200`	
	-	inverse	`-x, -200`	
	++	Increment [prefix, postfix]	`++count, count++`	
	--	Decrement [prefix, postfix]	`--count, count--`	
binary	+	addition	5.0 + 2.0 is 7.0,	5 + 2 is 7
	-	subtraction	5.0 – 2.0 is 3.0	5 – 2 is 3
	*	multiplication	5.0 * 2.0 is 10.0,	5 * 2 is 10
	/	division	5.0/2.0 is 2.5	5/2 is 2
	%	remainder	not applicable	5%2 is 1

II.8. Operators: Integer division and remainder

If <u>both</u> operands are integers, then

- / calculates the integer part of the division
- % calculates the integer remainder of the division (`m%n` is always less than `n`)

7.0/2.0 = 3.5

7/2 = 3

23/5 = 4

-21/4 = -5

3

7 - (7/2)*2 = 1

$\begin{array}{c|c} 7 & 2 \\ \cline{2-2} 1 & 3 \end{array}$

7%2 = 1

23%5= 3

-21 - (-21/4)*4 = -1

-21%4= -1

~~21.0%4.0 = ?~~

% is not defined for float and double

II.8 Operators: increment and decrement

- Increment operator ++ (increase by 1)
  - Pre-increment*:* ++count

    increment happens before **the value to be used**
  - Post-increment *:* count++

    increment happens **after the value is used**
- Decrement operator – (decrease by 1)
  - Pre-decrement*:* --count

    decrement happens before the value to be used
  - Post-decremente*:* count--

    decrement happens after the value is used

```
Example:
x=1;
printf("%d and %d",++x,x++);
printf("and %d",x);

output: 2 and 2 and 3
```

```
int j = i++;     //  the value of i is first assigned to j, then i is incremented.
 int j = ++i;  // the value of i is first incremented, then the new value i is assigned to j
```

If used properly, the increment and the decrement operators can increase efficiency of expressions

…Continued

expressions	example (assume `sum` is `10`, `counter` is `5`)
`counter++;`	counter = 6
`++counter;`	counter = 6
`sum = sum + counter++;`	sum = 15 counter = 6
`sum = ++counter + sum;`	sum = 16 counter = 6

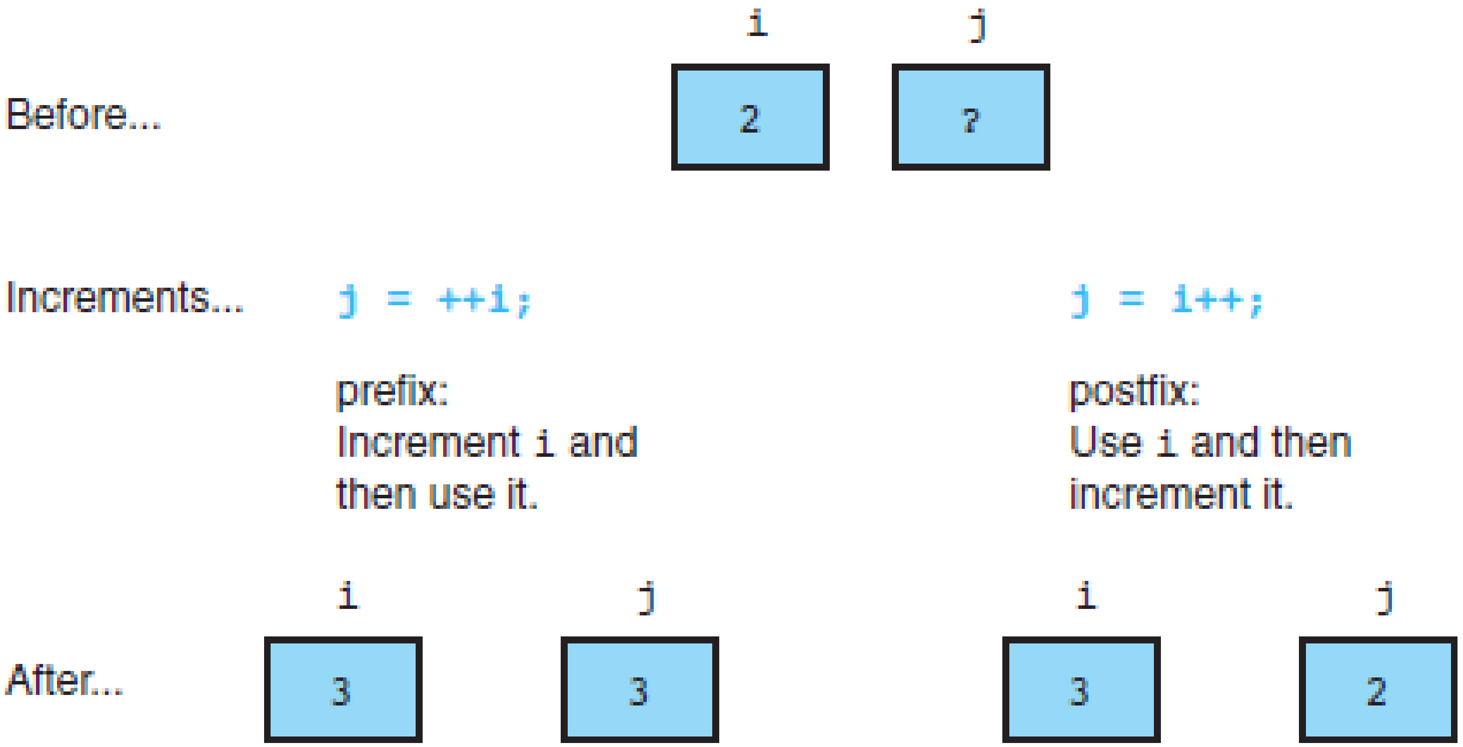
Equivalent to:

```
counter = counter +1;
sum = sum + counter;
```

Equivalent to:

```
sum = sum + counter;
counter = counter +1;
```

Comparison of pre-increment and post-increment



Before...

Increments...

`j = ++i;`

prefix:
Increment i and
then use it.

`j = i++;`

postfix:
Use i and then
increment it.

After...

Compound Assignment Operator

assignment	compound assignment
`sum = sum + number;`	`sum` **`+=`** `number;`
`product = product * number;`	`product` **`*=`** `number;`

Syntax:

variable **op=** *expression;*

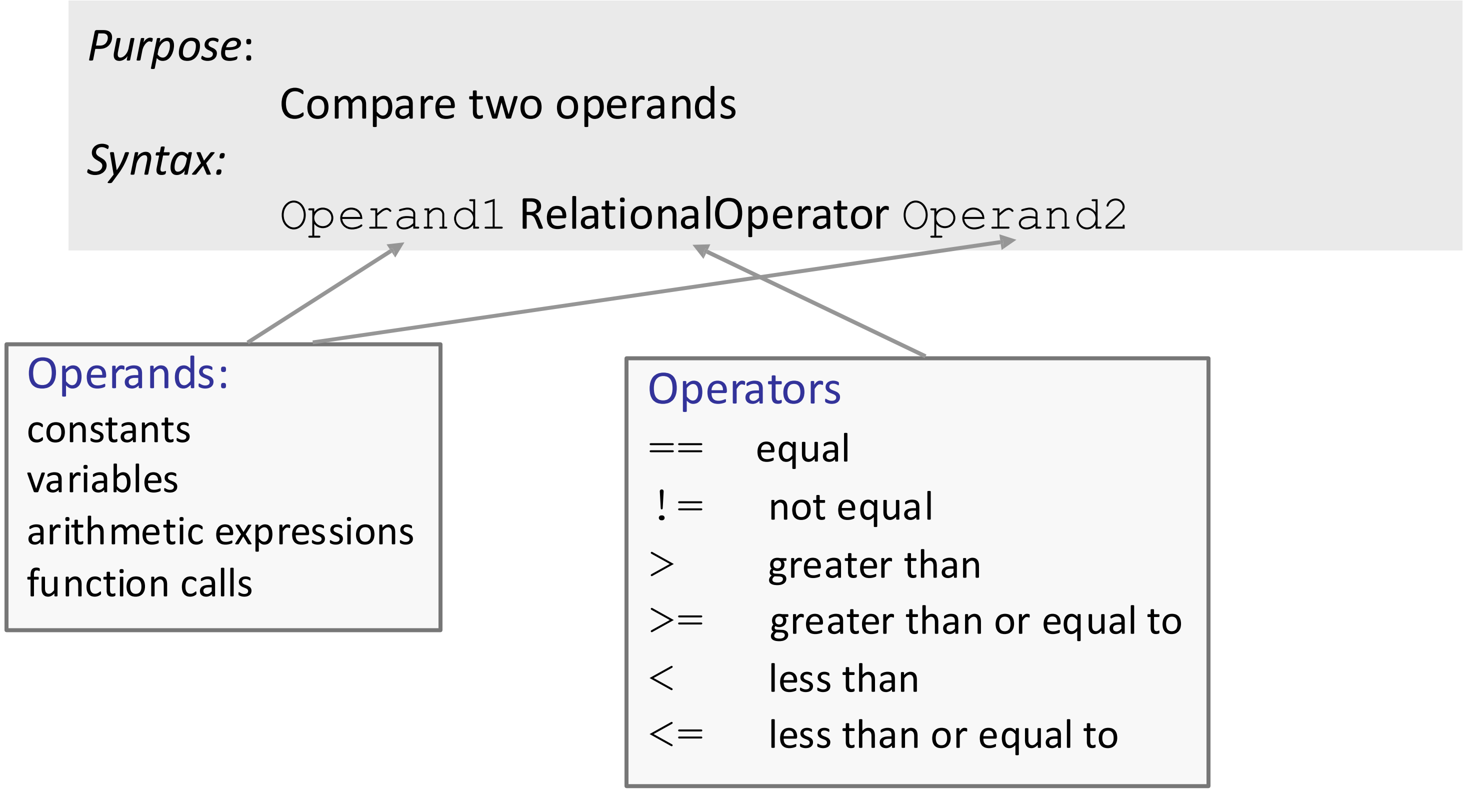
Meaning:

variable = variable ***op*** *expression*

Examples:

```
count += 2;            /* count = count + 2;  */
sum += 2+3;            /* sum = sum + (2+3)  */
stock -= quantity;     /* stock = stock – quantity */
power *= 2.71;         /* power = power * 2.71   */
div /= 10.0+20.0;      /* div = div/(10.0+20.0)  */
rem %= d;              /* rem = rem % d   */
```

Relational Operators



Purpose:

Compare two operands

Syntax:

`Operand1` RelationalOperator `Operand2`

Operands:

constants

variables

arithmetic expressions

function calls

Operators

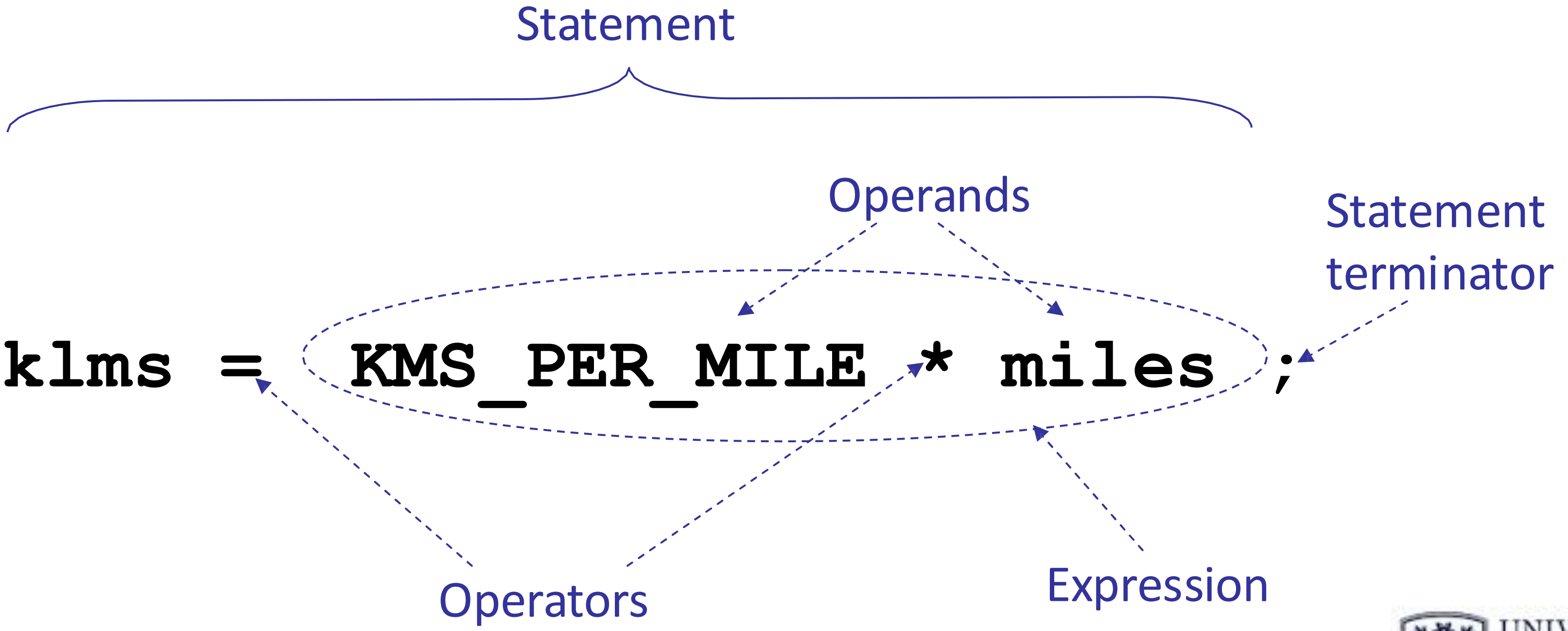
==	equal
!=	not equal
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Examples:

```
y >= 20
offset == (640 + x)
z < log(y)
```

II. 9. Expressions

- An “expression” is a combination of values, variables, operators, functions which returns a value



Statement

Operands

Statement terminator

```
klms =  KMS_PER_MILE * miles ;
```

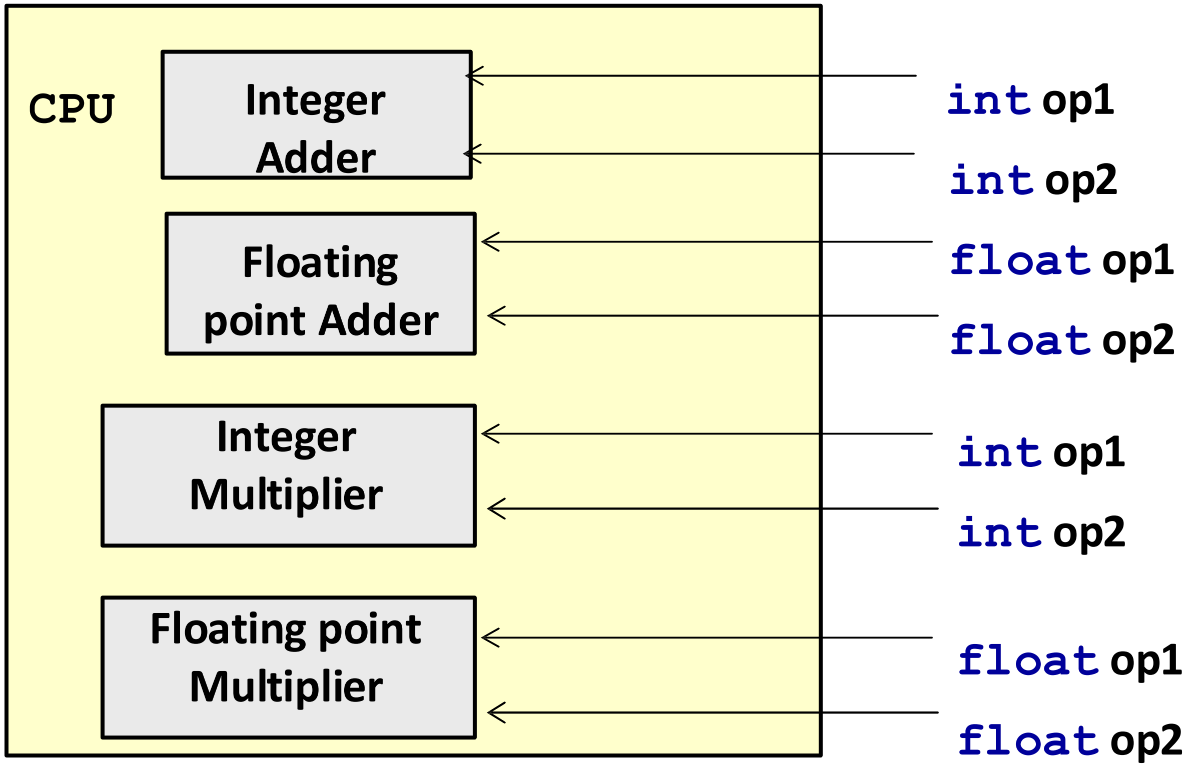
Operators

Expression

Mixed type expressions

- What is the type of the result of an expression that includes different data types? `2*12.25` is `24.5` or `24` or `25 ?`

- Basic arithmetic operations are supported by hardware modules in the CPU



Operands op1 and op2 <u>must</u> <u>have the same data type</u> in order to be processed by CPU

Data type conversion

- Operands of different types must be converted to a **common data type** before they can be sent to the CPU execution
  - Upward : conversion to a data type with a higher precision
  - Downward: conversion to a data type with a lower precision
- **C does upward conversion automatically**

```
wage = 2 * 14.8;    /* 2 is auto converted to 2.0 */
```

- You can **explicitly** convert a value to any data type upward or downward depending on your needs through **casting**

Syntax:

```
        (cast_type)expression;
```

Examples:

```
        (int)12.8;        /* 12.8 is converted to int 12 */
        (float)length;    /* length is converted to float */
```

Quiz 4

What values are assigned to `result ?`

```
int a = 5, b = 6;
float fa=5.0, fb=6.0, result;


result = (a + b)/2;             /* result = 5    */
result = (a + b)/2.0;           /* result = 5.5  */
result = (fa + b)/2;            /* result = 5.5  */
result = (float)(a + b)/2;      /* result = 5.5  */
result = (float)((a + b)/2);    /* result = 5.0  */
```

Evaluation of complex expressions

Calculate the value of the following C expression

```
-2 * -3/4%5 + -6 + 4
```

Rules used in C for evaluation of expressions

1. **Precedence rules**: describe how an underparenthesized expression should be parenthesized when the **expression mixes different kinds of operators**

2. **Associativity**:when two operators in an expression have the same precedence, their associativity is used to determine how the expression is evaluated.

Arithmetic operators precedence

Operators	Associativity
function calls , (…),postfix ++, --	left to right
(prefix) ++ --, unary(+ -), (type cast),sizeof( )	right to left
binary: *, /, %	left to right
binary: +, -	left to right
= , += , -= , *= , /= , %=	right to left

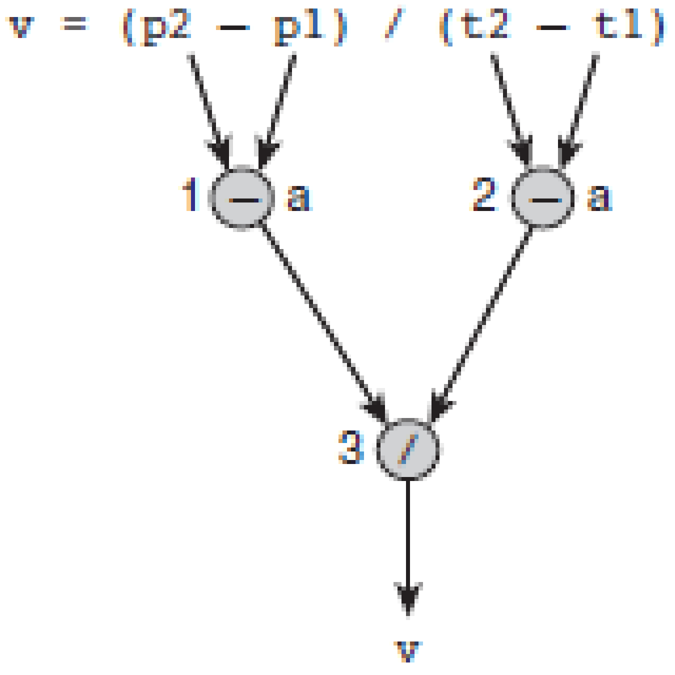
Expressions inside parentheses are evaluated first.

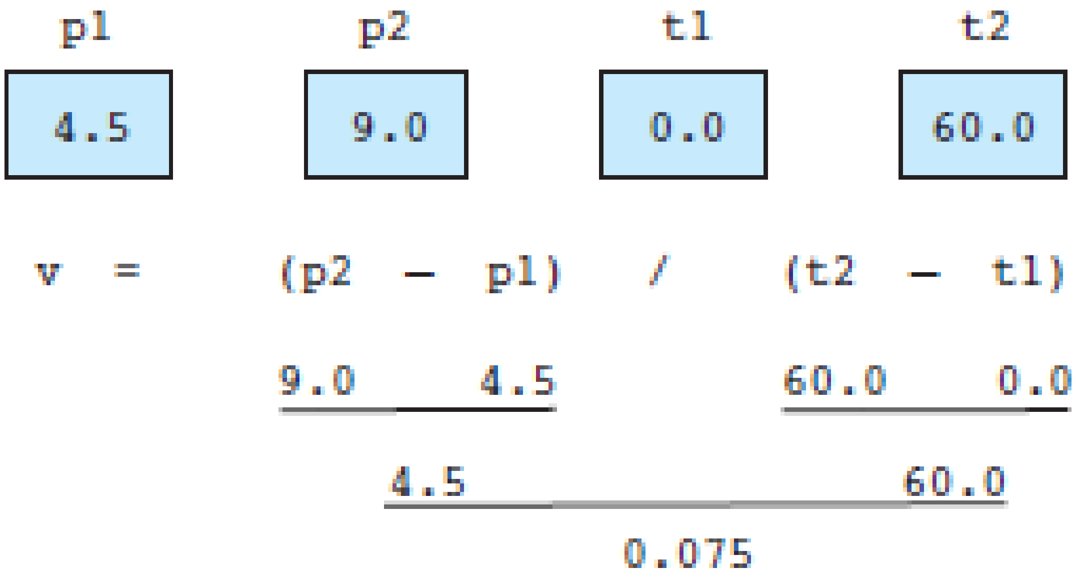
Example: In the expression “- a * b – c” the first minus is unary and the second is binary.

We can **use parentheses** to write an equivalent expression that is less likely to be misinterpreted:

```
((- a) * b) - c
```

Evaluation Tree and Evaluation for v = (p2 - p1) / (t2 - t1);





p1 4.5

p2 9.0

t1 0.0

t2 60.0

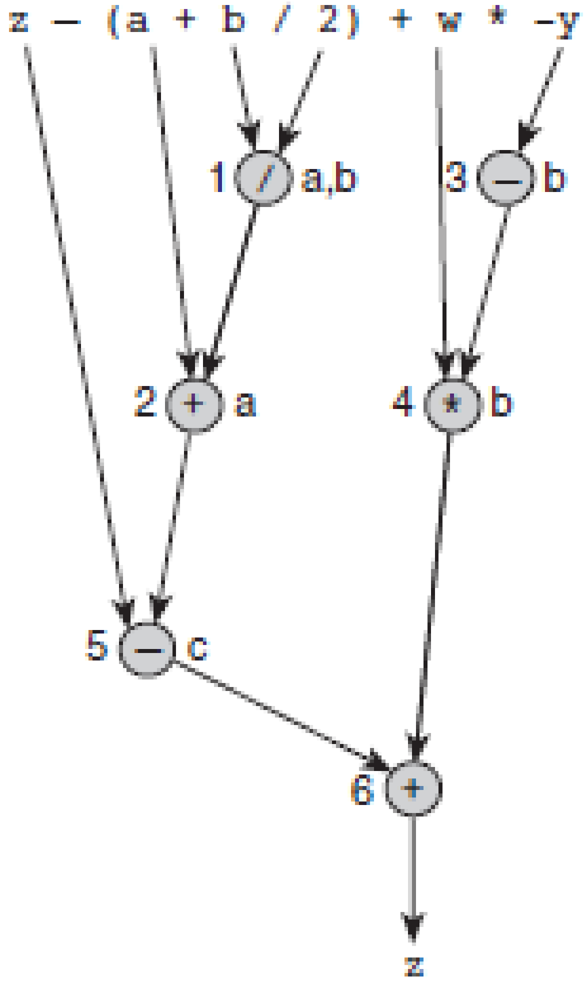
v = (p2 – p1) / (t2 – t1)

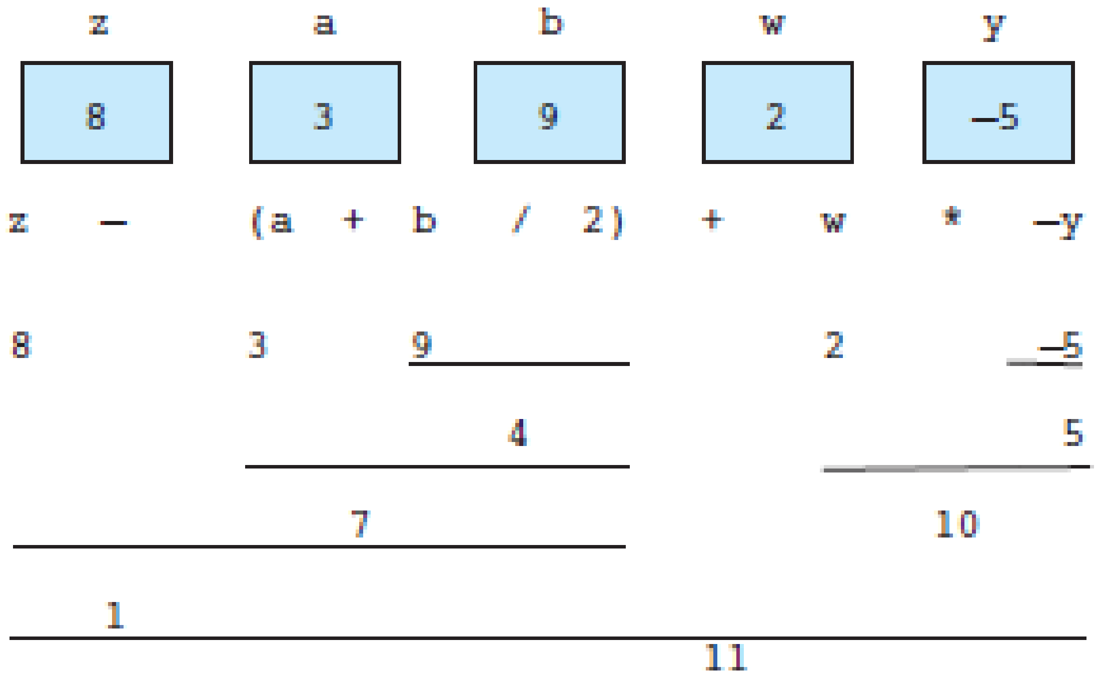
9.0 4.5 60.0 0.0

4.5 60.0

0.075

Evaluation Tree and Evaluation for z - (a + b / 2) + w * -y





Practice with Operators and Expressions

Declarations and Initializations

int   a = 1, b = 2, c = 3, d = 4;

Expression	Equivalent expression	Value
a * b / c	(a * b) / c	0
a * b % c + 1	((a * b) % c) + 1	3
++a * b - c --	((++a) * b) - (c--)	1
7 - -b * ++d	7 - ((-b) * (++d))	17

Quiz 5

Which expressions are not implemented correctly?

$$\frac{ab}{a+b} \longleftrightarrow$$ `a * b / ( a + b )`

$$a+\frac{b}{c^2} \longleftrightarrow$$ `a + b / c * c` X

$$\frac{\frac{a}{b}+c}{a-\frac{b}{c}} \longleftrightarrow$$ `a/b + c/( a - b/c )` X

$$c = p(1+r)^y \longleftrightarrow$$ `c = p * pow( (1+r), y )`

Appendix: Programming Style

- C language is case sensitive
- All Identifiers in your program should be meaningful, having descriptive names:

  `a1, b, x3, y` – confusing

  `colourPalet, lightIntensity, reflectionAngle` – OK

- Use CAPITAL for constants

  `LIMIT, THRESHOLD, GBP`

- Avoid names that differ only in case, like *foo* and *Foo*. Similarly, avoid *foobar* and *foo_bar*. The potential for confusion is considerable.
- Indentation

  Use spaces wisely

- Comments

  Concise and clear

  Consistent style

Indentation

- Chaotic indentation makes a program code messy

```
  #include <stdio.h>
int main( void )
 {
float x, y, tmp;
    printf("Input two numbers >");
 scanf("%f%f", &x, &y);
printf(“Before swap: x=%.2f, y=%.2f\n”, x,
y);
     tmp=x; y=tmp;
     x=y; printf(“After swap: x=%.2f,
y=%.2f\n”, x, y);
return 0;
   }
```

Indentation

- It is easier to follow a program code when it is well organised

```
#include <stdio.h>

int main( void )
{
    float x, y, tmp;
    printf("Input two numbers >");
    scanf("%f%f", &x, &y);    /* input x and y */
    printf(“Before swap: x=%.2f, y=%.2f\n”, x, y);
    /* swap x and y using a temporary buffer */
    tmp=x;
    x=y;
    y=tmp;
    /* Output the result */
    printf(“After swap: x=%.2f, y=%.2f\n”, x, y);
    return 0;
}
```