



ICS-103

Computer Programming in C

Lectures 3-6

Chapter 2:

Overview of C Programming Language

Dr. Tarek Ahmed Helmy El-Basuny

Topics to be Discussed

- Sequential/Procedural vs. Object-Oriented Programming
- Why it is important to learn C-language?
- Writing and Running C Programs
- General form of a C program
- Pre-Processor Directives (i.e. #include, #define, etc.)
- The main Function
- Standard Libraries
- Reserved words, Identifiers (Standard and User-defined),
- Simple C Program example (Adding Two Integers),
- Data Types (int, double, char, void), Constant and Variable Declarations,
- Assignment and Executable Statements
- **Input and Output Functions,**
- **Arithmetic Expressions, Arithmetic Operators**
- **Data Type of an Expression,**
- **Mixed-Type Assignment Statement**
- **Type Conversion Through Casts**
- **Unary and Binary Operators**
- **Rules for Evaluating Arithmetic Expressions with Multiple Operators**
- **Examples of Evaluating Arithmetic Expressions**
- **Formatting Numbers in Program Output**

Procedural vs. Object-Oriented Programming

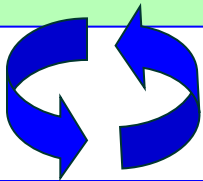
- ❑ The unit in Procedural Programming (PP) like C is a function, and the unit in Object-Oriented Programming (OOP) like C++ is a class.
- ❑ PP concentrates on creating functions, while OOP concentrates on the classes, and the methods inside them.
- ❑ PP separates the data of the program from the operations that manipulate the data, while OOP encapsulates both of them.
- ❑ In PP, the program is composed of a collection of instructions to the computer.
- ❑ In OOP, the program is composed of a collection of objects that communicate with each other.

Why is the C Language Important?

- ❑ C is the mother of all popular programming languages.
 - C programming language is used widely in coding operating systems, language compilers, network drivers, language interpreters, and system utilities.
- ❑ C is really simple to learn and practically does not require any dependencies.
- ❑ C offers a very flexible and dynamic memory management.
 - Memory is allocated statically, automatically, or dynamically in C programming with the help of **malloc** (single variable), **calloc** (two variable allocation with zeros initialized), **realloc** (change the allocated size), and **free** (de-allocate) library functions.
- ❑ Whatever the platform, C is probably available.
- ❑ C is portable language; this means that C programs written for one computer system can be run on another system, **with little or no modification**.
- ❑ C language is well suited for structured programming and commonly used programming language in industry.
- ❑ Produces optimized programs that run fast.
- ❑ Many companies and software projects do their programming in C.
- ❑ Once you have learned C, **you can learn any other languages by yourself**.
- ❑ It is a robust language with rich built in functions and operators that can be used to write any complex program.

Writing and Running C Programs

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```



1. Write the text of program (source code) using an editor such as **Dev C++** or **emacs**, save it as a file e.g. **my_program.c**

2. Run the compiler to convert program from source to an “executable” or “binary”:

```
$ gcc -Wall -g my_program.c -o my_program
```

```
$ gcc -Wall -g my_program.c -o my_program
tt.c: In function `main':
tt.c:6: parse error before `x'
tt.c:5: parm types given both in parmlist and separately
tt.c:8: `x' undeclared (first use in this function)
tt.c:8: (Each undeclared identifier is reported on ly once
tt.c:8: for each function it appears in.)
tt.c:10: warning: control reaches end of non-void function
tt.c: At top level:
tt.c:11: parse error before `return'
```

- **-Wall** option **means** to report all Warnings during the compilation.
- **-g** option means to generate debug information to be used by the debugger.
- **-o** option means write the build output to an output file.

3. If the Compiler gives errors and warnings; then Re-edit the source file, fix it, and re-compile.

4. If the compilation process succeeded then Run it and see if it works.

```
$ ./my_program
```

```
Hello World
```

```
$
```

What if it doesn't work?

my_program

General form of a C Program

Preprocessor directives

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

```
/*Include the source code for library file stdio.h into your program*/
```

```
#define KMS_PER_MILE 1.609
```

```
/*Substitute the name KMS_PER_MILE wherever it appears with 1.609 */
```

Comments

Statements that clarify the program, ignored by compiler but "read" by humans.
i.e.

```
/* Calculate cost of trip */
```

Main function and other functions where each Function body may contain:

```
{
```

Declarations

```
int kids, courses; /*Declares the variables kids and courses that can store integer values */
```

```
char initial; /* Declares the variable initial that can store a single character */
```

Executable statements

```
printf("Enter dist in miles> ");
```

```
scanf("%lf", &miles);
```

```
printf("That equals %f kms.\n", kms);
```

```
}
```

The **main** Function

- ❑ Every C program has a **main** function and it is the entry point (**execution begins**) of a C program. It is "**called**" by the operating system when the user runs the C program. The correct signature of the function is:

`main(void) { /* body */ }` // if there are no arguments and return value

`int main(void) { /* body */ }` // if there is a return value but no arguments

`int main(int argc, char *argv[]) { /* body */ }` // if there are arguments and return value

- ❑ Braces { and } mark the beginning and the end of the body of function **main**.
- ❑ **int** means that **main** "returns" an integer value.
- ❑ If we want to know whether the program has terminated successfully or not, we need a return value which can be **zero or a non zero value**.
- ❑ Hence the function becomes **int** main () and is recommended
- ❑ **A function body has two parts:**
 - **Declarations** part that tells the compiler what memory cells/**variables** are needed in the function.
 - **Executable statements** (derived from the algorithm) are translated into machine language and later executed by the computer.

General Form of a C program

#include directive will include header files that have the definitions of functions used in the program. Example: **printf** function is defined in the header file **stdio.h**

Can your program have more than one .c file?

This is a comment line where the compiler ignores it.

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```

- The variables are named **argc** (**argument count**) and **argv** (**argument vector**), but they can be given any valid identifier.
- **int main(int num_args, char**arg_strings)** is equally valid.
- **argc** is the number of arguments being passed into your program from the command line.
- **argv** is a one-dimensional array of strings.
- Each string is one of the arguments that was passed to the program.

Blocks of code (“lexical scopes”) are marked by { ... }

Return ‘0’ from this function

Print out a message. ‘\n’ means “new line”.

General Form of a C program

■ General form of a simple C program

```
preprocessor directives
main function header
{
    declaration of variables
    executable statements
}
```

"Executable statements"
usually consists of 3 parts:

- Input data
- Computation
- Output results

- Executable statements are translated into machine language and eventually executed.

- ❑ Preprocessor directives modify the text of a C program before compilation.
- ❑ Every variable has to be declared before using it.

```
#include <stdio.h>
int main(void) /* function main begins program execution */
{
    int num1,num2,result;
    num1 = 15;
    num2 = 20;
    result = num1 * num2;
    printf("result= %d\n", result);
    printf("Enter any character to terminate . . .");
    return 0;
}
```

The **main** Function

- ❑ Every C program has a **main** function.

```
#include <stdio.h>

int main(void) /* function main begins program execution */
{
    int num1,num2,result;
    num1 = 15;
    num2 = 20;
    result = num1 * num2;
    printf("result= %d\n", result);
    printf("Enter any character to terminate . . .");
    return 0;
}
```

**main function
body**

main function

- Any function body usually has two parts:
 - **declarations** - tell the compiler what memory cells are needed in the function
 - **executable statements** - (derived from the algorithm) are translated into machine language and later executed by the compiler.
 - Another function may be called to perform a certain task.

What is a Function?

- A **Function** is a series of instructions to perform a certain task.
 - You pass **arguments** to a function and it returns a result **value**.
- “**main()**” is a Function. It always gets called first when you run your program.

Return type, or void

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

```
/* The simplest C Program */
```

```
int main(int argc, char **argv)
```

```
{
```

```
    printf("Hello world\n");
```

```
    return 0;
```

```
}
```

Function Arguments

Returning a value

- “**printf()**” is another function, like **main()**.
- It’s defined for you in a “**c.library**”.
- “**c.library**”, is a collection of functions you can call from your program.

Standard Libraries

- ❑ Standard Libraries contains useful functions and symbols that are predefined by the C language developers.
 - You must include `<stdio.h>` if you want to use the `printf` and `scanf` library functions.
 - It contains information about `standard input and output functions` that are inserted into your program before compilation.
 - You must include `<math.h>` if you want to do some mathematical operations in your program.
 - You must include `<time.h>` if you are going to defines date and time handling functions.
 - You must include `<string.h>` if you are going to deal with string handling functions.

Pre-Processor Directives

- ❑ Preprocessor directives are commands that give instructions to the C preprocessor to modify a C program prior to its compilation.
- ❑ Preprocessor directives begin with #

#include <stdio.h>

- Include Standard I/O Library header file (.h file)
 - if you want to use the *printf* and *scanf* library functions.

#include <math.h>

- Include Standard Math Library header file (.h file)
 - if you want to use the *sqrt* and *abs* library functions.

#define PI 3.141593

- Define the constant PI

```
#include <stdio.h>
#define PI 3.14
int main()
{
    double area, radius;
    area = PI * radius * radius;
    printf("Area of circle = %lf", area);
    return 0;
}
```

#define Directive

- ❑ The #define directive instructs the preprocessor to replace each occurrence of a text by a particular constant value before compilation.
- ❑ **Should be placed outside main function.**

- ❑ #define replaces all occurrences of the text you specify with the constant value you specify.

#define NAME of the constant value

- ❑ Examples:

```
#define KMS_PER_MILES 1.609
```

```
#define PI 3.141593
```

Reserved Words

- ❑ **A reserved word** means a word that has special meaning to **C** and can not be used for other purposes.
- ❑ These are words that **C** reserves for its own uses.
- ❑ **Examples:**
 - **Built-in Types:** int, double, char, void, etc.
 - **Control flow:** if, else, for, while, return, etc.
 - **Reserved words** always appear in lower case.

Example of C Reserved Keywords

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

Example: Miles-to-Kilometers Conversion C 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);
}
```

Diagram labels and arrows:

- comment**: points to the first multi-line comment at the top.
- standard header file**: points to `<stdio.h>`.
- preprocessor directive**: points to `#include` and `#define`.
- constant**: points to the value `1.609` in the `#define` line.
- reserved word**: points to `int` and `main`.
- variable**: points to `miles` and `kms`.
- comment**: points to the comment `/* Get the distance in miles. */`.
- standard identifier**: points to `printf` and `scanf`.
- special symbol**: points to the asterisk `*` in `KMS_PER_MILE * miles`.
- reserved word**: points to `return`.
- punctuation**: points to the parentheses `()` in `return (0);`.
- special symbol**: points to the closing curly brace `}`.

Identifiers and Standard Identifiers

- ❑ An Identifier means a name given to a variable or a function in your program.
- ❑ Standard Identifier: An identifier defined in a standard C library and has special meaning in C.
 - Examples of standard identifiers are: printf, scanf, sqrt, string.h, time.h, etc.
 - Standard identifiers are not reserved words.
 - You can redefine standard identifiers if you want to, but it is not recommended.
 - For example, if you defined your own function printf, then you cannot use the C library function printf.

User-Defined Identifiers

- ❑ We choose our own identifiers to
 - Name memory-cells/**variables** that will hold data and program results.
 - Name functions that we define.
- ❑ **Rules for Naming Identifiers:**
 - An identifier consists only of **letters**, **digits**, and **underscores** (It can start with underscore)
 - **Commas** or **blank** spaces **are not allowed** within an identifier.
 - An identifier **cannot** begin with a **digit**.
 - **C** reserved keywords **cannot** be used as an identifier.
 - A standard **C** identifier **should not** be redefined.
 - Identifiers **should not be** of length more than 32 characters, **some compiler may accept identifiers with length more than 32 characters.**
 - **Uppercase** and **lowercase** letters are distinct (**identifiers are case sensitive**).
 - **No Special** Symbols other than **underscore(_)** **are allowed**.
 - **First Character** should be **alphabet or Underscore**.
- ❑ **Examples of Valid identifiers:** **letter1**, **inches**, **KM_PER_MILE**, **_NUM**, **Num1**,
- ❑ **Examples of Invalid identifiers:** please explain why?
 - 1letter, 1b a, Happy\$trout, !abc123, abc.123, return, char, int, continue, etc.

Guidelines for Naming Identifiers

- ❑ Uppercase and lowercase are different
 - LETTER, Letter, letter are different identifiers
 - Avoid names that only differ by case. They can lead to problems of finding bugs (errors) in the program.
- ❑ Choose meaningful identifiers (easy to understand)
- ❑ Example: distance = speed * time
 - Means a lot more than $z = x * y$
- ❑ Choose #define constants to be ALL UPPERCASE
 - Example: KMS_PER_MILE is a defined constant
 - As a variable, we can probably name it:
KmsPerMile or Kms_Per_Mile

Simple C Program: Adding Two Integers

```
/* Addition program */
```

```
1  #include <stdio.h>
2  int main() // the main function without arguments can accept any parameters
3  {
4      int integer1, integer2, sum;      /* declaration */           ☐ Initialize variables
5      printf( "Enter first integer\n" ); /* prompt */
6      scanf( "%d", &integer1 );        /* read an integer */       ☐ Input
7      printf( "Enter second integer\n" ); /* prompt */
8      scanf( "%d", &integer2 );        /* read an integer */
9      sum = integer1 + integer2;        /* assignment of sum */     ☐ Sum
10     printf( "Sum is %d\n", sum );     /* print sum */             ☐ Print
11     return 0; /* indicate that program ended successfully */
12 }
```

Enter first integer

45

Enter second integer

72

Sum is 117

Simple C Program: Adding Two Integers

- ❑ The program contains
 - Comments, `#include <stdio.h>` and `main`
- ❑ Declaration of variables
 - **Variables**: locations in memory where a value can be stored
- ❑ `int integer1, integer2, sum;`
 - `int` means the variables can hold integers (-1, 3, 0, 47)
 - Variable names (**identifiers**)
 - `integer1, integer2, sum`
 - **Identifiers**: consist of letters, digits (cannot begin with a digit).
 - Case sensitive
 - **Declarations appear before executable statements**
 - If an **executable statement references undeclared** variable then it will produce a syntax (**compiler**) error.

Simple C Program: Adding Two Integers

□ `scanf("%d", &integer1);`

⇒ Obtains a value from the user

- `scanf` uses standard input (usually keyboard)

⇒ This `scanf` statement has two arguments

- `%d` - indicates data should be a decimal integer
- `&integer1` - location in memory to store variable
- The `&` (**ampersand symbol**) usually included with the variable name in `scanf` statements.

⇒ When executing the program:

- The user responds to the `scanf` statement by typing in a number, then **presses the enter** (return) key.

Simple C Program: Adding Two Integers

□ = (assignment operator)

- Assigns a value to a variable
- Is a binary operator (has two operands)

sum = **variable1** + **variable2**;

sum gets **variable1** + **variable2**;

- Usually the variable receiving value will be on the left

□ printf("Sum is %d\n", sum);

- Similar to **scanf**

- **%d** means decimal integer will be printed
- **\n** means print the sum value in a new line
- **sum** specifies what integer will be printed

- Calculations can be performed inside **printf** statements

printf("Sum is %d\n", integer1 + integer2);

Data Types

- ❑ **Data Types**: simply refers to the **type** and size of **data** associated with variables and functions. **Fundamental Data types in C** are:-
 - **int**: stores signed integer values: whole numbers, i.e. 65, -12345, ...
 - **float**: floating point value: i.e. a number with a fractional part (movable decimal point). i.e. 0.5, 0.71428, -33.33, 3.14, $\frac{1}{2}$, $\frac{5}{7}$.
 - A **float** number is a 32 bit max. (23 for the whole, 1 bit for the sign, 8 bits for the exponent), i.e. float has 6 decimal digits of precision.
 - **float** numbers take up less memory and are faster in processing.
 - A **double** number is a 64 bit max. (1 bit for the sign, 11 bits for the exponent, and 52 bits for the value),
 - **double** number has 15 decimal digits of precision and uses more memory than float.
 - **char**: Stores character values.
 - Each char value is enclosed in single quotes: 'A', '*'
 - A character can be a letter, digit, or special symbol.
 - Arithmetic (+, -, *, /) and comparison (<, >, ..) operations can be performed on **int**, **float**, and **double** types where compare operations can be performed on **char** type.

Integer and Floating-Point Data Types

Integer Types in C

Type	Size in Memory	Range (from ~ to ~)
short	2 bytes = 16 bits	-32768 to +32767
unsigned short	2 bytes = 16 bits	0 to 65535
int	4 bytes = 32 bits	-2147483648 to +2147483647
unsigned int	4 bytes = 32 bits	0 to 4294967295
long	4 bytes = 32 bits	Same as int
long long	8 bytes = 64 bits	-9×10^{18} to $+9 \times 10^{18}$

Floating-Point Types in C

Type	Size in Memory	Approximate Range	Significant Digits
float	4 bytes = 32 bits	10^{-38} to 10^{+38}	6
double	8 bytes = 64 bits	10^{-308} to 10^{+308}	15

Characters and ASCII Code

Character Type in C

Type	Size in Memory	ASCII Codes
char	1 byte = 8 bits	0 to 255

ASCII Codes and Special Characters

Character	ASCII Code	Special Characters	Meaning
'0'	48	' '	Space Character
'9'	57	'*'	Star Character
'A'	65	'\n'	Newline
'B'	66	'\t'	Horizontal Tab
'Z'	90	'\''	Single Quote
'a'	97	'\"'	Double Quote
'b'	98	'\\'	Backslash
'z'	122	'\0'	NULL Character

Constant Declarations

- ❑ **Constants** refer to values that cannot be changed during execution of the program, neither by the programmer nor by the computer. **Constants** are also called **literals**.
- ❑ **Constants can be of any basic data types** like an **integer** constant, a **floating** constant, a **character** constant.
- ❑ **Numeric constants** e.g. 3 5.25 -2.5428**e**5 3.682**E**-4
 - Note -2.5428**e**5 has the value $-2.5428 * 10^5$,
 - similarly 3.682**E**-4 has the value $3.682 * 10^{-4}$
- ❑ **The number after the **e** or **E** must be **integer**;**
 - ⇒ Hence the following constant is **invalid**: 2.5**e**3.0 .
- ❑ **A constant can be declared by using either of the following two methods:**
 - ⇒ The **#define** pre-processor directive or
 - ⇒ By using the **const** keyword in a declaration:
- ❑ Example:
 #define **PI** 3.14159
 or
 const **double** **PI** 3.14159;
 or
 double **const** **PI** 3.14159;

Variable Declarations

- ❑ **Variables:** The memory cells used for storing a program's input data and its computational results.
 - The Value of a variable can change at runtime.
- ❑ **Variable declarations:** Statements that communicate to the compiler the names of variables in the program and the type of data they can store.
 - Syntax: <variable_type> <variable_name> = <initial_value>;
 - Example: `int studentID;`
 `double age = 18.4;`
 `double miles, kms;`
 `int count;`
 `char answer;`
- ❑ **C** requires that you declare every variable in the program before using it.

Executable Statements

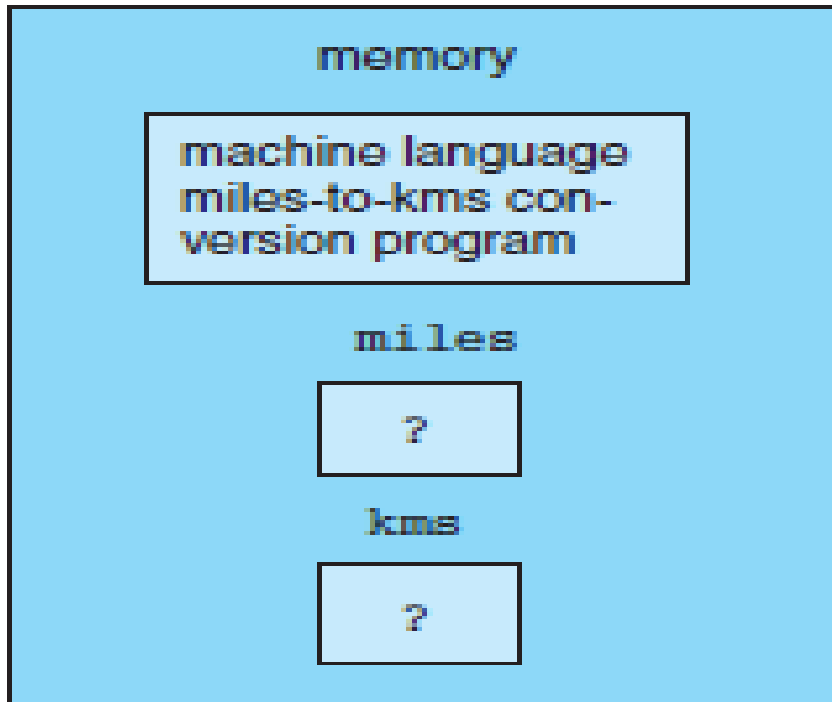
- ❑ **Executable Statements:** C statements used to write or code the algorithm.
 - C compiler translates the executable statements to machine code.
- ❑ **Examples of executable Statements:**
 - Assignment statements, such as **sum = variable1 + variable2;**
 - Function calls, such as calling **printf** and **scanf**.
 - **return** statement.
 - **if** and **switch** statements (**selection**) - will be explained later.
 - **for** and **while** statements (**iteration**) - will be explained later.

Assignment Statement

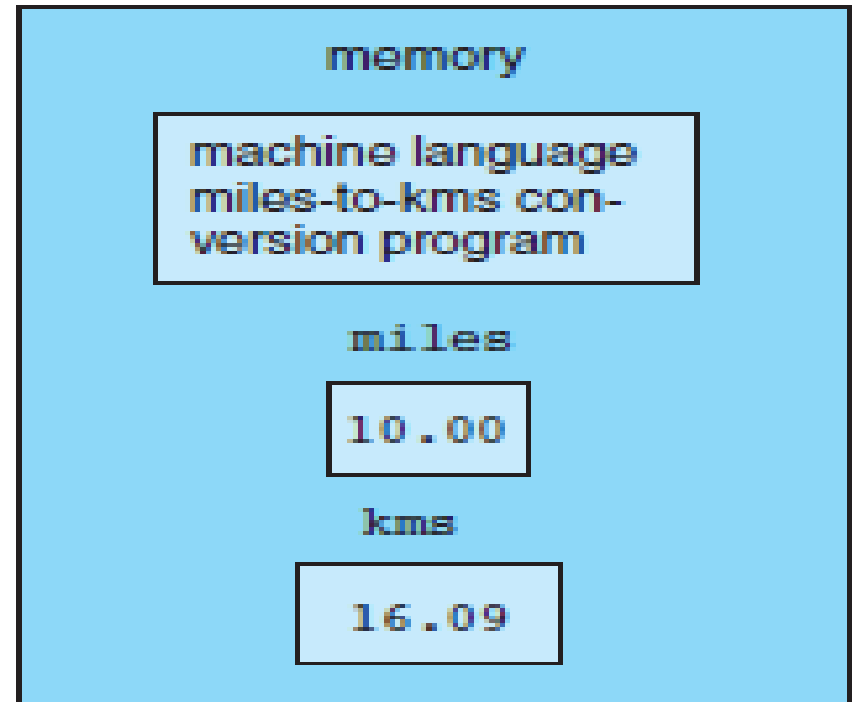
- ❑ The **assignment statement** computes the expression that appears after the assignment operator (=) and stores **its value in the variable that appears to the left**.
- ❑ Stores a value or a computational result in a variable
 - `variable = expression;`
 - `=` is the **assignment operator**
- ❑ **For example:**
 - `a=10;` /*Stores the value **10** in the **int** variable **a***/
 - `average = (a+b) / 2;` /* the result of (a+b)/2 will be stored in the variable named **average***/.

Programs in Memory

- ❑ Miles-to-Kilometers Conversion Program before and after executing the program.
- Program in Memory: Before execution (a) and After Execution (b).
- The ? In the memory cells miles and kms (a) indicate that the values of these cells are undefined before program execution begins.
- Once the values of these variables are read from the input device, will be written into the memory cells as shown in (b).



(a)



(b)

Assignment Statement

Effect OF $\text{kms} = \text{KMS_PER_MILE} * \text{miles}$

Before assignment

KMS_PER_MILE

miles

kms

1.609

10.00

?

*

16.090

After assignment

KMS_PER_MILE

miles

kms

1.609

10.00

16.090

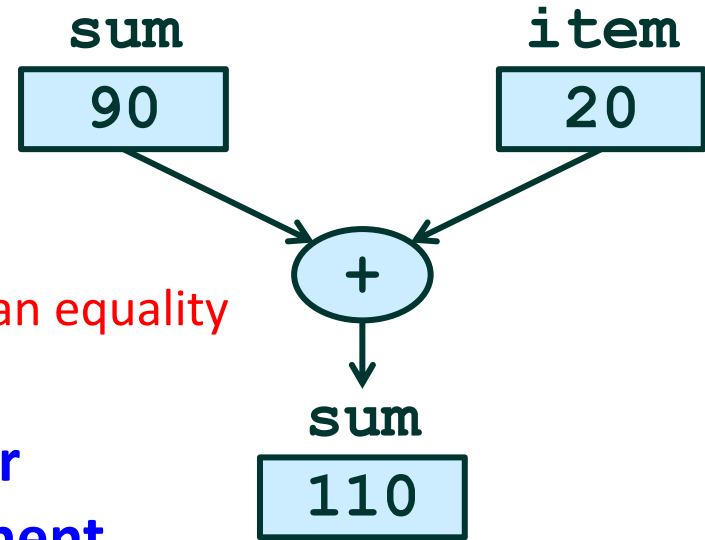
- The value assigned to **kms** is the result of multiplying the constant **KMS_PER_MILE** by the variable **miles**.

Assignment Statement

Effect OF: $\text{sum} = \text{sum} + \text{item}$

- Read $=$ as "becomes"

Before assignment



- **Note**, the assignment operator **does NOT** mean equality
- The equality operator in C is $==$

After
assignment

- Example of assignment statement:
 - $\text{next_letter} = \text{'A'}$;
 - $\text{new_x} = \text{x}$;

Input/Output Operations and Functions

- ❑ **Input Operation:** data transfer from the outside world (i.e. typed on the keyboard by the program's user, read from a file, received from another computer, sensed by sensors, etc.) into computer's memory.
- ❑ **Output Operation:** program results can be sent to the outside world (i.e. displayed on the monitor to the program's user, written into a file, sent to another computer, converted into action, etc.).
- ❑ **Input/output Functions:** special library functions that do all input/output operations.
 - **Printf:** output function
 - **scanf:** input function
- ❑ We have to include "**stdio.h**" header file to make use of the **printf()** and **scanf()** library functions in C language.
- ❑ **Function call/invoke:** used to call or activate a function for execution.
 - Asking another piece of code to do some work for you

The **printf** Function

- It is a **C** predefined **output** function in the "**stdio.h**" header file,
- By using the **printf** function, we can print the data or user defined message on console or monitor.
- To generate a new line, we use "**\n**" in C **printf()** statement.
- **The signature of the printf function is as following:**

function name function arguments

↓ ↓

```
printf ("That equals %f kilometers.\n", kms) ;
```

format string place holder print list

↑ ↑ ↑

❑ If the value of **kms** is 16.0900000, then the output will be like:

That equals 16.0900000 kilometers.

Placeholders

- ❑ Placeholders always begin with the (percent) % symbol.
 - % marks the place in a format string where a value will be printed out or will be read.
- ❑ Format strings can have **multiple placeholders**, if you are printing multiple values.

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

- Note: the placeholder used with scanf are the same as those used with printf except with variables of type double.
- Type double variable use %f placeholder with printf and %lf with scanf.

Other Placeholder

Placeholder	Output Conversions
`%d',	Print an integer as a signed decimal number
`%ld',	Print a long integer as a signed decimal number
`%o'	Print an integer as an unsigned octal number
`%s'	Print string variable
`%u'	Print an integer as an unsigned decimal number
`%f'	Print a floating-point number in normal (fixed-point) notation
`%lf'	Print a Long double
`%x'	Print hexadecimal variable.
`%e', `%E'	Print a floating-point number in exponential notation.
`%c'	Print a single character
`%lc'	Print a single wide character.
`%p'	Print the value of a pointer
`%%'	Print a literal '%' character.

Multiple Placeholders

- Format strings can have multiple placeholders if printf or scanf call has several variables.
- `printf("Color %s, number1 %d\n", "red", 123456);`
- `printf("Hi %c %c %c – your age is %d\n", letter_1, letter_2, letter_3, age);`
- If `letter_1`, `letter_2`, `letter_3` are assigned `ABC` characters and the `age` variable assigned `35`, then last printf will display

Hi ABC – your age is 35

Topics to be Discussed

- ⇒ Sequential/Procedural vs. Object-Oriented Programming
- ⇒ Why it is important to learn C-language?
- ⇒ Writing and Running C Programs
- ⇒ General form of a C program
- ⇒ Pre-Processor Directives (i.e. #include, #define, etc.)
- ⇒ The main Function, Standard Libraries
- ⇒ Reserved words, Identifiers (Standard and User-defined),
- ⇒ Simple C Program example (Adding Two Integers),
- ⇒ Data Types (int, float, double, char),
- ⇒ Constant and Variable Declarations,
- ⇒ Assignment and Executable Statements,
- ⇒ Output Function, printf
- ⇒ Input Function, scanf
- ⇒ Arithmetic Expressions, Arithmetic Operators
- ⇒ Data Type of an Expression,
- ⇒ Mixed-Type Assignment Statement
- ⇒ Type Conversion through Casts
- ⇒ Unary and Binary Operators
- ⇒ Rules for Evaluating Arithmetic Expressions with Multiple Operators
- ⇒ Examples of Evaluating Arithmetic Expressions
- ⇒ Formatting Numbers in Program Output

Displaying Prompts

- When input data is needed in an interactive program, you should use the `printf` function to display a prompting message that tells the user what data to enter.

```
printf("Enter the distance in miles> ");
```

```
printf("Enter the object mass in grams> ");
```


Formatting Integers in Program Output

- ❑ You can specify how `printf` will display integers
- ❑ For integers, use `%nd`
 - `%` start of placeholder
 - `n` is the optional **field width** = number of columns to display
 - If `n` is less than integer size, it will be ignored
 - If `n` is greater than integer size, **spaces are added to the left.**

Value	Format	Output		Value	Format	Output
234	<code>%4d</code>	234		-234	<code>%4d</code>	-234
234	<code>%5d</code>	234		-234	<code>%5d</code>	-234
234	<code>%6d</code>	234		-234	<code>%6d</code>	-234
234	<code>%1d</code>	234		-234	<code>%2d</code>	-234

Formatting Type Double Values

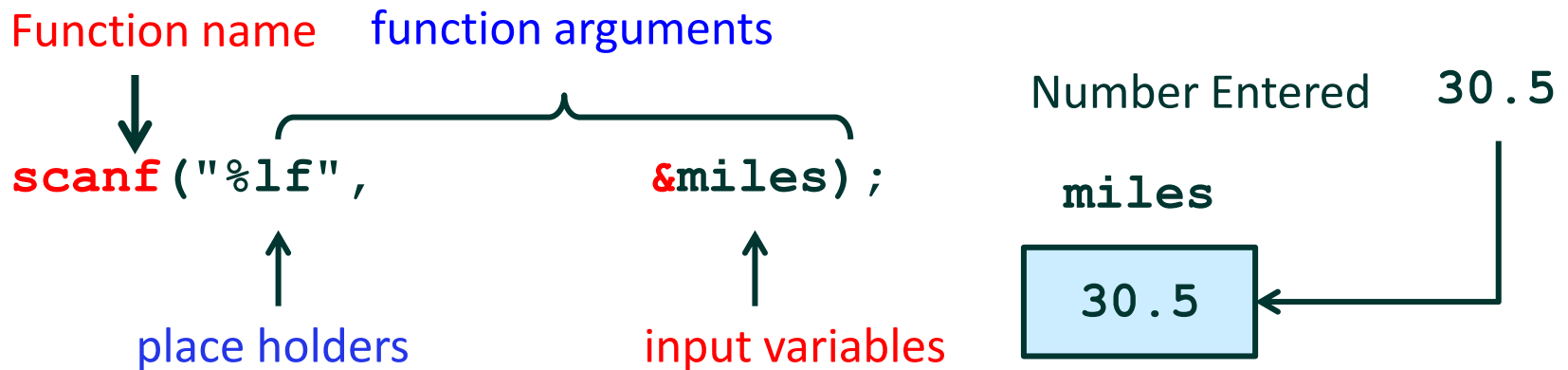
❑ Use **%n.mf** for double values

- **n** is the optional field width = number of digits in the whole number, the unary minus, decimal point, and fraction digits.
- If **n** is less than what the number needs it will be ignored
- **.m** is the number of decimal places (optional)

Value	Format	Output	Value	Format	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
0.1234	%4.2f	0.12	-0.006	%4.2f	-0.01
-0.006	%8.3f	-0.006	-0.006	%8.5f	-0.00600
-0.006	%.3f	-0.006	-3.14159	%.4f	-3.1416

The **scanf** Function

- It is a **C** predefined **input** function in the "**stdio.h**" header file.
- By using the **scanf** function, we can read the data typed by the user on the keyboard. **The signature of the scanf function is as following:**



- ❑ The **ampersand** symbol (&) is the **address operator**. It tells **scanf** the address of variable **miles** in memory.
- ❑ When user inputs a value, it is stored in **miles**.
- ❑ The placeholder **%lf** tells **scanf** the type of data to store into variable **miles**.

Multiple Placeholders

- Format strings can have multiple placeholders if `printf` or `scanf` call has several variables.

```
char letter1, letter2, letter3;
```

```
scanf ("%c%c%c",
```

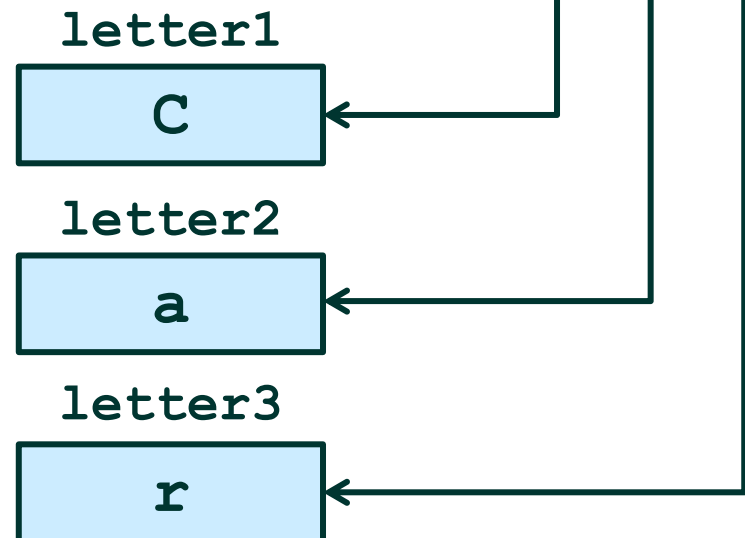
```
&letter1,
```

```
&letter2,
```

```
&letter3) ;
```

Letters Entered

C a r



scanf & printf example: Adding Two Integers

```
/* Addition program */
```

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

```
2 int main()
```

```
3 {
```

```
4     int integer1, integer2, sum;    /* declaration */
```

□ Initialize variables

```
5     printf( "Enter first integer\n" ); /* prompt */
```

```
6     scanf( "%d", &integer1 );    /* read an integer */
```

□ Input

```
7     printf( "Enter second integer\n" ); /* prompt */
```

```
8     scanf( "%d", &integer2 );    /* read an integer */
```

```
9     sum = integer1 + integer2;    /* assignment of sum */
```

□ Sum

```
10    printf( "Sum is %d\n", sum ); /* print sum */
```

□ Print

```
11    return 0; /* indicate that program ended successfully */
```

```
12 }
```

Enter first integer

45

Enter second integer

72

Sum is 117

A Program accepts two integers and check if they are equal

```
1. /*C program to accept two integers and check if they are equal*/
2. #include <stdio.h>
3. int main(void)
4. {
5.     int m, n;
6.     printf("Enter the values for M and N\n");
7.     scanf("%d %d", &m, &n);
8.     if (m == n)
9.         printf("M and N are equal\n");
10.    else
11.        printf("M and N are not equal\n");
12.}
```

Case:1

Enter the values for M and N

3 3

M and N are equal

Case:2

Enter the values for M and N

5 8

M and N are not equal

Return Statement

- ❑ The **return** statement transfers control from a function back to the caller function.
- ❑ Once you start writing your own functions, you will use the **return** statement to return the result of a function back to the caller.

```
int main()
{
    int n = adder(25, 17);
    printf("adder's result is = %d", n);
}

int adder(int a, int b)
{
    int c = a + b;
    return c;
}
```

- ❑ Syntax: **return expression**;
- ❑ Example: **return (0)**;
- ❑ Returning from the **main** function terminates the program and transfers control back to the operating system. **Value returned is 0.**

Comments

- ❑ Comments make it easier for us to understand the program, but are ignored by the **C** compiler.
- ❑ Comments are used to create **Program Documentation and Help others read and understand the program.**
- ❑ The start of the program should consist of a comment that includes **programmer's name**, date, **current version**, and **a brief description of what the program does.**
- ❑ **Two forms of comments:**
 - **/* C comment */** anything between **/*** and ***/** is considered a comment, **even if it spans on multiple lines.**

```
/*  
 * Author: kfupm student  
 * Purpose: To show a comment that spans multiple lines.  
 * Language: C  
 */
```
 - **// C++ comment** anything after **//** is considered a comment until the end of the line.
 - **#define AGE 6** **// This constant is called AGE**
- ❑ **Always Comment your Code!**

Programming Style

❑ Why we need to follow conventions?

- A program that looks good is easier to read and understand than one that is sloppy.
- 80% of the cost of software goes to maintenance.
- Hardly any software is maintained for its whole lifetime by the original programmer.
- Programs that follow the typical conventions are more readable and allow engineers to understand the code more quickly and thoroughly.

❑ Check your text book on how to improve your programming style.

White Spaces

- ❑ The compiler ignores extra blanks between words and symbols, but **you may insert space to improve the readability and style of a program.**
- ❑ You should always leave a blank **space after a comma** and **before and after operators such as**: + - * / and =
- ❑ You should Indent/**align/shift** the lines of code in the body of a function.
- ❑ **Indent** means arrange statements relative to their neighboring statements,
 - ➔ How many spaces should you shift the bodies of the statements?
 - At least 2 spaces.
 - No more than 8 spaces (1 tab).
 - The same amount for all statement bodies.

❑ **Example:**

```
int sum(int a, int b)
{
    int result;
    result = a + b;
    return result;
}
```

```
int sum(int a, int b)
{
int result;
result=a+b;
return result;
}
```

White Space Example

Bad alignment :

```
int main(void)
{ int foo,blah;
scanf("%d",&foo);
blah=foo+1;
printf("%d", blah);
return 0;}
```

Good alignment :

```
int main(void)
{
    int foo, blah;
    scanf("%d", &foo);
    blah = foo + 1;
    printf("%d", blah);
    return 0;
}
```

Bad Programming Practices

- ☐ Missing statement of purpose
- ☐ Inadequate commenting
- ☐ Variables names are not meaningful
- ☐ Use of unnamed constant
- ☐ Alignment does not represent program structure
- ☐ Algorithm is inefficient or difficult to follow
- ☐ Program does not compile
- ☐ Program produces incorrect results
- ☐ Insufficient testing (test case results are different than expected, program is not fully tested for all cases).

Arithmetic Expressions

- ❑ To solve most programming problems, you need to write **arithmetic expressions** that compute data of different types (i.e. **int**, **double**, sometimes **char**).
- ❑ Arithmetic expressions contain **variables**, **constants**, **function calls**, **arithmetic operators**, as well as **sub-expressions written within parentheses**.
- ❑ Examples:
 - ➡ $\text{sum} + 1$
 - ➡ $(a + b) * (c - d)$
 - ➡ $(-b + \text{sqrt}(\text{delta})) / (2.0 * a)$

Arithmetic Operators

Operator	Meaning	Examples
+	Addition	5 + 2 is 7 5.0 + 2.0 is 7.0 'B' + 1 is 'C'
-	Subtraction	5 - 2 is 3 5.0 - 2.0 is 3.0 'B' - 1 is 'A'
*	Multiplication	5 * 2 is 10 5.0 * 2.0 is 10.0
/	Division	5 / 2 is 2 5.0 / 2.0 is 2.5
%	Modulus Operator and remainder of after an integer division.	5 % 2 is 1

Operators / and %

Example	Result	Explanation
$8 \text{ (dividend)} / 5 \text{ (divisor)}$	1	Integer operands \rightarrow integer result
$8.0 / 5.0$	1.6	floating-point operands \rightarrow floating-point result
$8 / -5$	-1	One operand is negative \rightarrow negative result
$-8 / -5$	1	Both operands are negative \rightarrow positive result
$8 \% 5$	3	Integer remainder of dividing 8 by 5
$8 \% -5$	3	Positive dividend \rightarrow positive remainder
$-8 \% 5$	-3	Negative dividend \rightarrow Negative remainder

□ */ and % are undefined when the divisor is 0.*

Data Type of an Expression

- ❑ What is the type of expression $x+y$ when x and y are both of type int ?
 - (answer: type of $x+y$ is int)
- ❑ The data type of an expression depends on the type(s) of its operands.
 - If both are of type int , then the expression is of type int .
- ❑ **Mixed-type** expression: is an expression that has mixed operands of type int and double .
 - If either one or both operands are of type double , then the expression is of type double .

Mixed-Type Assignment Statement

- ❑ If the expression being evaluated and the variable to which it is assigned have **different data types**.
- ❑ The expression is first evaluated; and the **result is assigned to the variable to the left side of = operator**.
 - Example: what is the value of $y = 5/2$ when y is of type **double**? (answer: $5/2$ is **2**; $y = 2.0$)
- ❑ **Warning**: assignment of a type **double** expression to a type **int** variable **causes the fractional part of the expression to be lost**.
 - Example: what is the type of the assignment $y = 5.0 / 2.0$ when y is of type **int**?
 - (answer: $5.0/2.0$ is **2.5**; $y = 2$)

Type Conversion Through Casts

- ❑ C allows the programmer to convert the type of an expression by placing the desired type in parentheses before the expression.
- ❑ This operation is called a **type cast**.
 - ➡ `(double) 5 / (double) 2` is the **double** value 2.5
 - ➡ `(int) (9 * 0.5)` is the **int** value 4
- ❑ When casting from **double** to **int**, **the decimal fraction is truncated** (NOT rounded).

Example of The Use of Type Casts

```
/* Computes a test average */
#include <stdio.h>
int main(void)
{
    int    total;        /* total score */
    int    students;     /* number of students */
    double average;      /* average score */
    printf("Enter total students score>\n ");
    scanf("%d", &total);
    printf("Enter number of students>\n ");
    scanf("%d", &students);
    average = (double) total / (double) students;
    printf("Average score is %.2f\n", average);
    return 0;
}
```

Unary and Binary Operators

- ❑ Operators are of two types: **unary** and **binary**
- ❑ **Unary operators** take only one operand (**variable**)
 - **Unary minus** (-) **and Unary plus** (+) operators
 - **++variable**: **prefix** increment, **example**: ++x is a shorthand for $x = x + 1$
 - **--variable**: **prefix** decrement, **similarly** --x is a shorthand for $x = x - 1$
 - **variable++**: **postfix** increment, x++ is a shorthand for the statement $x = x + 1$ but the result of x++ is the value of x **BEFORE** the value is changed.
 - **Example**: assume x is 8, x++ changes x to 9 but returns the value 8.
 - **variable--**: **postfix** decrement, x-- is a shorthand for the statement $x = x - 1$ but the result of x-- is the value of x **BEFORE** the value is changed.
 - **Example**: assume x is 8, x-- changes x to 7 but returns the value 8.
- ❑ **Binary operators** take two operands
 - Examples: addition (+), subtraction (-), multiplication (*), division (/) and integer remainder (%) operators.
- ❑ **A single expression could have multiple operators**
 - $v = u + a * t$, we are multiplying two numbers and result is added to 'u' and total result is assigned to v.

Position of Operators in an expression

Type	Explanation	Example
Infix	Expression in which Operator is in between Operands	$a + b$
Prefix	Expression in which Operator is written before Operands	$+ a b$
Postfix	Expression in which Operator is written after Operands	$a b +$

Operator name	Syntax	Meaning
Addition assignment	$a += b$	$a = a + b$
Subtraction assignment	$a -= b$	$a = a - b$
Multiplication assignment	$a *= b$	$a = a * b$
Division assignment	$a /= b$	$a = a / b$
Modulo assignment	$a \% = b$	$a = a \% b$

Priority Rank	Operator Description	Operator	Associativity
1	Multiplication	*	Left to Right
1	Division	/	Left to Right
1	Modulo	%	Left to Right
2	Addition	+	Left to Right
2	Subtraction	-	Left to Right

Arithmetic Operations Program

```
#include <stdio.h>
main() {
    int a = 21;
    int b = 10;
    int c ;
    c = a + b;        //Addition
    printf("Line 1 - Value of c is %d\n", c );
    c = a - b;        //Subtraction
    printf("Line 2 - Value of c is %d\n", c );
    c = a * b;        //Multiplication
    printf("Line 3 - Value of c is %d\n", c );
    c = a / b;        //Division
    printf("Line 4 - Value of c is %d\n", c );
    c = a % b;        //Reminder
    printf("Line 5 - Value of c is %d\n", c );
    c = a++;          //Post increment
    printf("Line 6 - Value of c is %d\n", c );
    c = a--;          //Post Decrement
    printf("Line 7 - Value of c is %d\n", c );
}
```

Arithmetic Operations Program

```
#include <stdio.h>
main() {
    int a = 21;
    int b = 10;
    int c ;
    c = a + b;
    printf("Line 1 - Value of c is %d\n", c );
    c = a - b;
    printf("Line 2 - Value of c is %d\n", c );
    c = a * b;
    printf("Line 3 - Value of c is %d\n", c );
    c = a / b;
    printf("Line 4 - Value of c is %d\n", c );
    c = a % b;
    printf("Line 5 - Value of c is %d\n", c );
    c = a++;
    printf("Line 6 - Value of c is %d\n", c );
    c = a--;
    printf("Line 7 - Value of c is %d\n", c );
}
```

Output:

Line 1 - Value of c is 31

Line 2 - Value of c is 11

Line 3 - Value of c is 210

Line 4 - Value of c is 2

Line 5 - Value of c is 1

Line 6 - Value of c is 21

Line 7 - Value of c is 22

<http://www.learn-c.org/>

Rules for Evaluating Expressions with Multiple Operators

- ❑ **Parentheses rule:** All expressions in parentheses must be evaluated separately.
 - Nested parenthesized expressions must be evaluated **from the inside out**, with the innermost expression evaluated first.
- ❑ **Operator precedence rule:** Multiple operators in the same expression are evaluated in the following order:
 - **First:** unary $+$, $-$
 - **Second:** $*$, $/$, $\%$
 - **Third:** binary $+$, $-$
- ❑ **Associativity rule**
 - **Unary operators** in the same sub-expression and at the same precedence level are **evaluated right to left**.
 - **Binary operators** in the same sub-expression and at the same precedence level are **evaluated left to right**.

Example: Expression Evaluation

1. What the value of x in the following expression?

$$x = 9 - 12 / 3 + 3 * 2 - 1$$

- The $*$ and $/$ operators are having higher precedence than $+$ and $-$ operators.
- The $*$ and $/$ operators are at the same level of precedence,
- We go left-to-right, and apply the $/$ operator first so:

$$x = 9 - 4 + 3 * 2 - 1$$

- Next, we apply the $*$ operator and the expression becomes:

$$x = 9 - 4 + 6 - 1$$

- Next, we apply the first $-$ operator as the $-$ and $+$ operators are at the same level and we go from left to right. The expression becomes:

$$x = 5 + 6 - 1$$

- Now, we apply the $+$ operator and the expression become:

$$x = 11 - 1$$

- Finally, we apply the $-$ operator and the result is:

$$x = 10$$

Example: Expression Evaluation

1. Which of the following correctly shows the hierarchy of arithmetic operations in C?

- A. $/ + * -$
- B. $* - / +$
- C. $+ - / *$
- D. $/ * + -$

2. Which of the following is the correct order of evaluation for the below expression?

$z = x + y * z / 4 \% 2 - 1.$

- A. $* / \% + - =$
- B. $= * / \% + -$
- C. $/ * \% - + =$
- D. $* \% / - + =$

Example: Expression Evaluation

1. Which of the following correctly shows the hierarchy of arithmetic operations in C?

A. $/ + * -$

B. $* - / +$

C. $+ - / *$

D. $/ * + -$

Answer: D

2. Which of the following is the correct order of evaluation for the below expression?

$z = x + y * z / 4 \% 2 - 1.$

A. $* / \% + - =$

B. $= * / \% + -$

C. $/ * \% - + =$

D. $* \% / - + =$

Answer: A

Example of Postfix and Prefix Unary Plus

```
#include<stdio.h>
void main()
{
    int i = 0, j = 0;
    j = i++ + ++i;
    printf("%d\n", i);
    printf("%d\n", j);
}
```

<http://www.learn-c.org/>

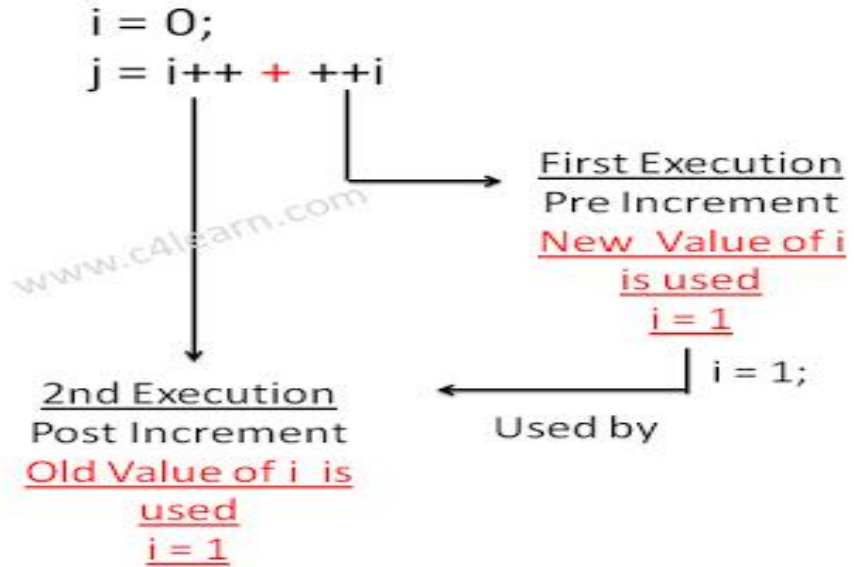
What is the Output :

Example of Postfix and Prefix Unary Plus

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i = 0, j = 0;
    j = i++ + ++i;
    printf("%d\n", i);
    printf("%d\n", j);
}
```

Output :

2
2



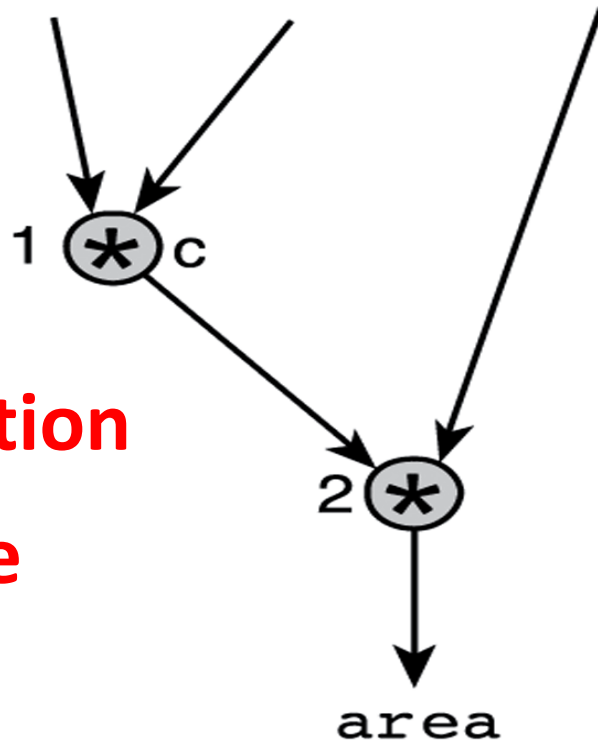
Operator	Precedence Rank
Pre Increment	1
Post Increment	2
Arithmetic Operator	3
Assignment Operator	4

Rules for Evaluating Expressions

area = PI * radius * radius

**Step-by-Step
Expression
Evaluation**

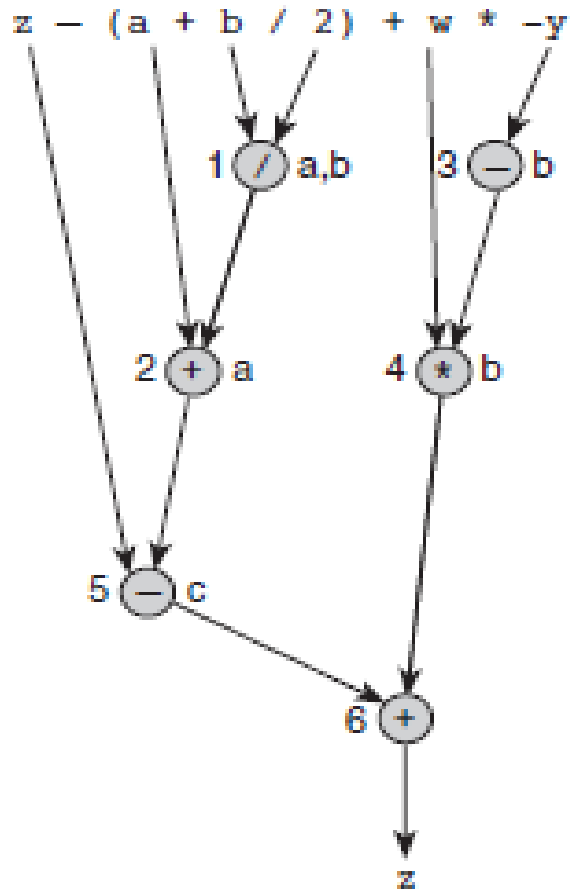
**Evaluation
Tree**



area	=	PI	*	radius	*	radius
		3.14159		2.0		2.0
		<hr/>				
		6.28318				
		<hr/>				
					12.56636	

Rules for Evaluating Expressions

Evaluate: $z - (a + b/2) + w * -y$

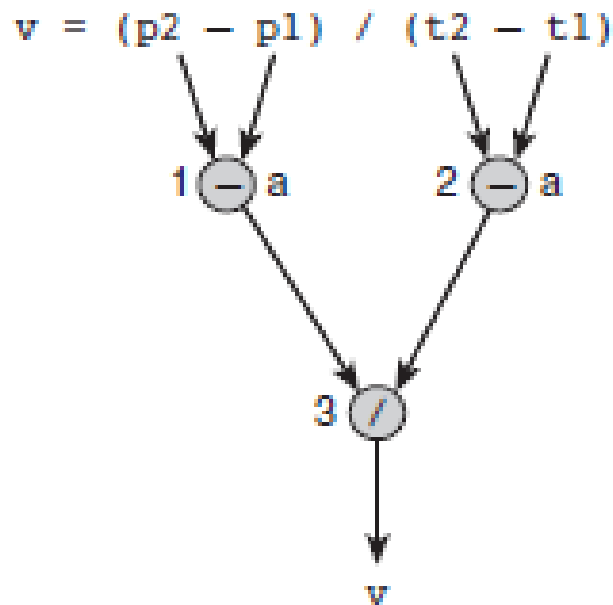


z	a	b	w	y
8	3	9	2	-5
z	-	(a + b / 2)	+	w * -y
8	3	9	2	-5
		4		5
		7		10
1				11

Evaluation Tree

Rules for Evaluating Expressions

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



p1	p2	t1	t2
4.5	9.0	0.0	60.0

$$v = \frac{(p2 - p1)}{(t2 - t1)}$$
$$\frac{9.0 - 4.5}{60.0 - 0.0}$$
$$\frac{4.5}{60.0}$$
$$0.075$$

Writing Mathematical Formulas in C

- ❑ You may encounter two problems in writing a mathematical formula in C.
- ❑ **First**, multiplication often can be implied in a formula by writing two letters to be multiplied next to each other.
- ❑ In C, you must state the ***** operator
 - For example, **2a** should be written as **2 * a**.
- ❑ **Second**, when dealing with division we often have:

$$\frac{a + b}{c + d}$$

- This should be coded as $(a + b) / (c + d)$.

Supermarket Coin Processor

```
1.  /*
2.   * Determines the value of a collection of coins.
3.   */
4.  #include <stdio.h>
5.  int
6.  main(void)
7.  {
8.      char first, middle, last; /* input - 3 initials          */
9.      int pennies, nickels;    /* input - count of each coin type */
10.     int dimes, quarters;     /* input - count of each coin type */
11.     int dollars;             /* input - count of each coin type */
12.     int change;              /* output - change amount          */
13.     int total_dollars;        /* output - dollar amount          */
14.     int total_cents;          /* total cents                     */
15.
16.     /* Get and display the customer's initials. */
17.     printf("Type in 3 initials and press return> ");
18.     scanf("%c%c%c", &first, &middle, &last);
19.     printf("\n%c%c%c, please enter your coin information.\n",
20.           first, middle, last);
21.
22.     /* Get the count of each kind of coin. */
23.     printf("Number of $ coins > ");
24.     scanf("%d", &dollars);
25.     printf("Number of quarters> ");
```

(continued)

Supermarket Coin Processor (cont'd)

```
26.     scanf("%d", &quarters);
27.     printf("Number of dimes    > ");
28.     scanf("%d", &dimes);
29.     printf("Number of nickels > ");
30.     scanf("%d", &nickels);
31.     printf("Number of pennies > ");
32.     scanf("%d", &pennies);
33.
34.     /* Compute the total value in cents. */
35.     total_cents = 100 * dollars + 25 * quarters + 10 * dimes +
36.                 5 * nickels + pennies;
37.
38.     /* Find the value in dollars and change. */
39.     dollars = total_cents / 100;
40.     change = total_cents % 100;
41.
42.     /* Display the credit slip with value in dollars and change. */
43.     printf("\n\n%c%c%c Coin Credit\nDollars: %d\nChange:  %d cents\n",
44.           first, middle, last, dollars, change);
45.
46.     return (0);
47. }
```

Type in 3 initials and press return> JRH

JRH, please enter your coin information.

Number of \$ coins > 2

Number of quarters> 14

Number of dimes > 12

Number of nickels > 25

Number of pennies > 131

JRH Coin Credit

Dollars: 9

Change: 26 cents

Common Programming Errors

❑ Syntax Errors (Detected by the Compiler)

- Violating one or more grammar rules.
- Missing semicolon (end of variable declaration or statement).
- Undeclared variable (using a variable without declaration).
- Comment not closed (missing `*/` at end of comment).

❑ Run-Time Errors (NOT detected by compiler)

- Detected by the computer when running the program.
- Illegal operation, such as dividing a number by zero.
- Program cannot run to completion.

❑ Undetected and Logic Errors

- Program runs to completion but computes wrong results.
- Input was not read properly.
- Wrong algorithm and computation.



The End!!

Thank you

Any Questions?