Overview of C Language

Reading Assignment

- Chapter 2: Sections 1-6.
- Chapter 3: Section 1.
- Chapter 4: Sections 1-4 and 7.
- Chapter 5: Sections 1-8.

Learning Outcomes

After going through these chapters, it is expected that we

- become familiar with the general form of a **C program** and the basic elements in a program.
- understand the use of **data types** and the differences between the data types `int`, `double` and `char`.
- learn how **C evaluates arithmetic expressions** and how to write them in C.
- develop C code using **Control structures, conditions, relational and logic operators**.
- develop C code using The **while, for and do-while** loops.

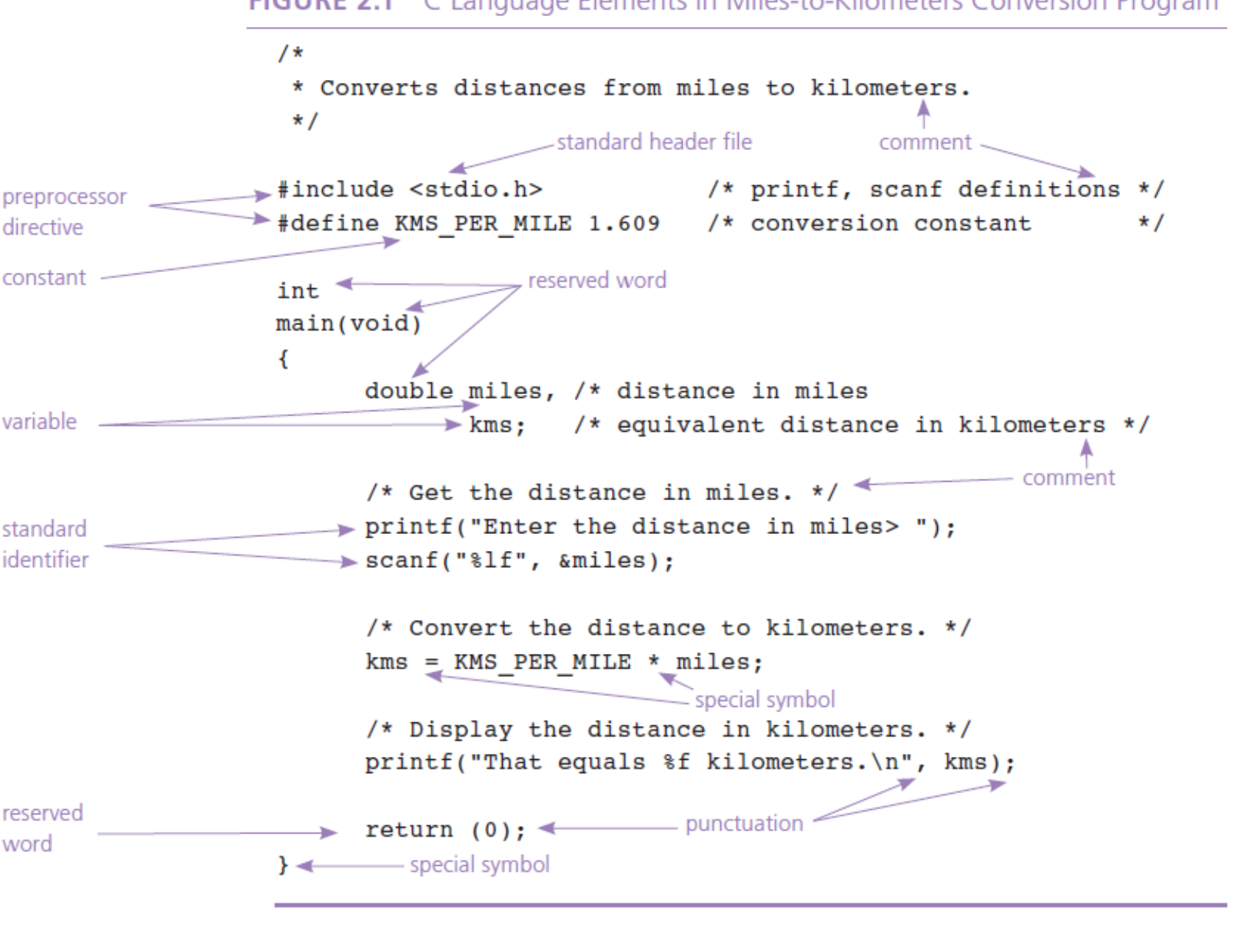
Why Learn C?

- Many companies and software projects do their programming in C.
- C Produces optimized programs that run fast.
- Low-Level access to computer memory via pointers.
- Can be complied to run on a variety of computers.

C Language Elements - Example

- We will use the online C compiler in https://www.onlinegdb.com/online_c_compiler

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




```c
/*
 * 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);
}
```

Preprocessor Directives

- The C program has two parts:
  - preprocessor directives; and
  - the main function

- The preprocessor directives are commands that give instructions to the C preprocessor, whose job it is to modify the text of a C program before it is compiled.
- A preprocessor directive begins with a number symbol (#) as its first nonblank character.
- The two most common directives appearing in the previous figure are:
  - `#include`
  - `#define.`

- The C language explicitly defines only a small number of operations: Many actions that are necessary in a computer program are not defined directly by C.
- Instead, every C implementation contains collections of useful functions and symbols called libraries.
- The ANSI (American National Standards Institute) standard for C requires that certain standard libraries be provided in every ANSI C implementation.
- A C system may expand the number of operations available by supplying additional libraries; an individual programmer can also create libraries of functions.
- Each library has a standard header file whose name ends with the symbols `.h`.

- The `#include` directive gives a program access to a library.
- This directive causes the preprocessor to insert definitions from a standard header file into a program before compilation. The directive
  - `#include <stdio.h> /*printf, scanf definitions*/`
- notifies the preprocessor that some names used in the program (such as `scanf` and `printf`) are found in the standard header file `<stdio.h>`.

- The other preprocessor directive
  - `#define KMS_PER_MILE 1.609 /*conversion constant*/`
- associates the constant macro `KMS_PER_MILE` with the meaning 1.609. This directive instructs the preprocessor to replace each occurrence of `KMS_PER_MILE` in the text of the C program by 1.609 before compilation begins.

Syntax Displays for Preprocessor Directives

#include Directive for Defining Identifiers from Standard Libraries

SYNTAX: `#include <standard header file>`

EXAMPLES:
```c
#include <stdio.h>
#include <math.h>
```

INTERPRETATION: `#include` directives tell the preprocessor where to find the meanings of standard identifiers used in the program. These meanings are collected in files called *standard header files.* The header file `stdio.h` contains information about standard input and output functions such as `scanf` and `printf`. Descriptions of common mathematical functions are found in the header file `math.h`. We will investigate header files associated with other standard libraries in later chapters.

#define Directive for Creating Constant Macros

SYNTAX: `#define` *NAME value*

EXAMPLES:
```c
#define MILES_PER_KM 0.62137
#define PI 3.141593
#define MAX_LENGTH 100
```

INTERPRETATION: The C preprocessor is notified that it is to replace each use of the identifier *NAME* by *value*. C program statements cannot change the value associated with *NAME*.

Function main

main Function Definition

SYNTAX:
```
int
main(void)
{
      function body
}
```

- **int main(void)** marks the beginning of the main function where program execution begins.
- **Every C program has a main function.**
- **Braces {** and **}** mark the beginning and end of the body of function main.
- A function body 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 computer

Reserved Words

- 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
- Built-in Types: **int, double, char, void, etc.**
- Control flow: **if, else, for, while, return, etc.**
- Always lower case.

Standard Identifiers

- Standard identifier is a word having special meaning but one that a programmer may redefine
  - But redefinition is not recommended.
- Examples:
  - `printf`, `scanf`
- For example, if you define 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 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
  - An identifier cannot begin with a digit
  - A C reserved word cannot be used as an identifier
  - A standard C identifier should not be redefined
- Examples of Valid identifiers:
  - **letter1, inches, KM_PER_MILE**
- Examples of Invalid identifiers:
  - **1letter, Happy$trout, return**

Hello World! Example

```c
#include <stdio.h>

int main() {

    // printf() displays the string inside quotation

    printf("Hello, World!");

    return 0;

}
```

Key Points for Python Programmers

- The first thing a Python programmer notices is significant overhead to simply print a string.
- The **#include** works like import in Python (not really, but the analogy works for now) so in this case the input/output stream library is brought in.
- The `int main()` begins the declaration of the main function, which in **C** indicates the first function to be executed, i.e. "begin here". In Python, `main` was the start of the program **by convention only**.
- The curly braces { and } delimit the beginning and end of a block of code (compound statement), in this case the beginning and end of the function `main`. Python identifies blocks through indentation.

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.
  - Examples:

```c
double miles, kms;

    int count, large;

    char answer;
```

- C requires that you declare every variable in the program.

Syntax Display for Declarations

SYNTAX:
```
int variable_list;
double variable_list;
char variable_list;
```

EXAMPLES:
```c
int count,
    large;
double x, y, z;
char first_initial;
char ans;
```

INTERPRETATION: A memory cell is allocated for each name in the *variable_list*. The type of data (`double`, `int`, `char`) to be stored in each variable is specified at the beginning of the statement. One statement may extend over multiple lines. A single data type can appear in more than one variable declaration, so the following two declaration sections are equally acceptable ways of declaring the variables `rate`, `time`, and `age`.

```c
double rate, time;   |   double   rate;
int age;             |   int      age;
                     |   double   time;
```

Key Points for Python Programmers

- In **C**, all variables must be declared before use.
- Python figures out types based on the objects you assign names to, but **C** wants to know up front.
- Python allows names to refer to different types of objects while the program is running; in **C** the type remains fixed.

Data Types

- Data Types: a set of values and a set of operations that can be performed on those values
  - int: Stores signed integer values: whole numbers -Examples: 65, -12345
  - double: Stores real numbers that use a decimal point
    - Examples: 3.14159 or 1.23e5 (which equals 123000.0)

Key Points for Python Programmers

- Some C types are similar:
  - int, float, string
- Data type char is different:

Data Type char

- char represents an individual character value
  - a letter, a digit or a special symbol.
- Each value of type `char` is enclosed in apostrophes (single quotes):
  - For example, `'A','z', ':'`

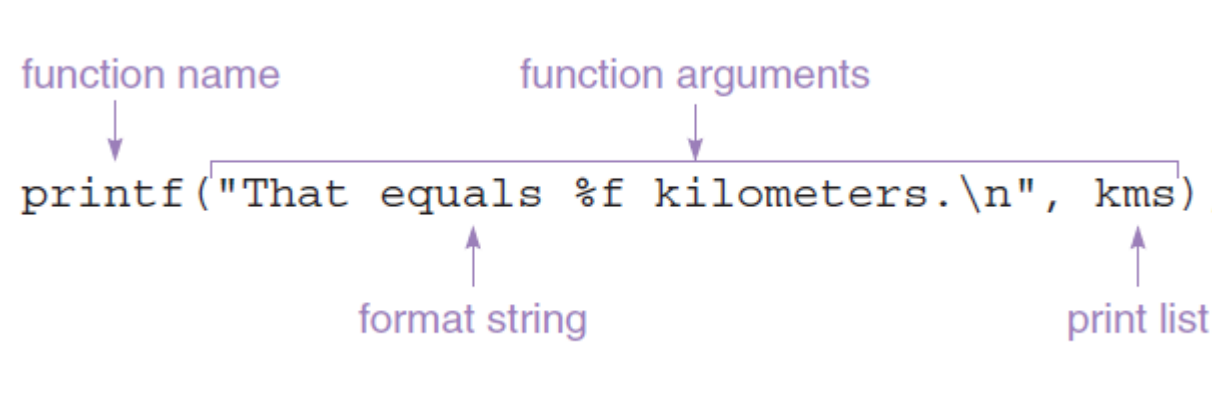
- ASCII code: a particular code that specifies the integer representing each char value.

TABLE 2.7 ASCII Codes for Characters

Character	ASCII Code
`' '`	32
`'*'`	42
`'A'`	65
`'B'`	66
`'Z'`	90
`'a'`	97
`'b'`	98
`'z'`	122
`'0'`	48
`'9'`	57

Input/Output Operations and Functions

- **Input operation**: data transfer from the outside world into computer memory
- **Output operation**: program results can be displayed to the program user
- **Input/Output functions**: special program units that do all input/output operations
  - printf : output function
  - scanf : input function
- **Function call**: used to call or activate a function
  - Asking another piece of code to do some work for you



- Output: That equals 16.090000 Kilometers.

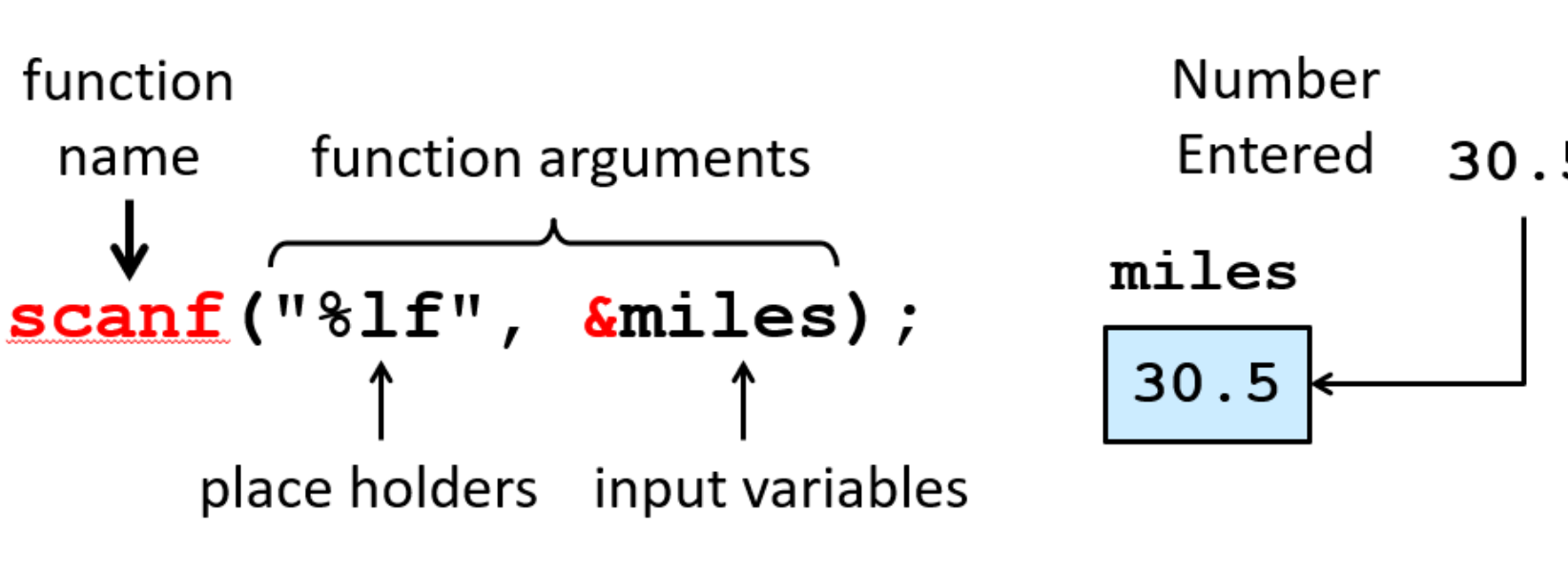
Placeholders

- Placeholders always begin with the 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

TABLE 2.8 Placeholders in Format Strings

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

The scanf Function



- The **&** is the address (reference) operator. It tells scanf the address of variable miles in memory.
  - The reason is that parameters in function calls in C are all called **by value**.
    - Exactly like Python.
- When user inputs a value, it is stored in miles.
- The placeholder **%lf** tells **scanf** the type of data to store into variable miles.

Arithmetic Expressions

- To solve most programming problems, you need to write arithmetic expressions that compute data of type **int** and **double** (and sometimes **char**)
- Arithmetic expressions contain variables, constants, function calls, arithmetic operators, as well as sub-expressions written within parentheses.

TABLE 2.9 Arithmetic Operators

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

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**.
  - If either one or both operands are of type **double**, then the expression is of type **double**.
- An expression that has mixed operands of type **int** and **double** is a mixed-type expression.

Mixed-Type Assignment Statement

- The expression being evaluated and the variable to which it is assigned have **different data types**
- The expression is first evaluated; then 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 Conversions 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).

```c
/* 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> ");
    scanf("%d", &total);
    printf("Enter number of students> ");
    scanf("%d", &students);
    average = (double) total / (double) students;
    printf("Average score is %.2f\n", average);
    return 0;
}
```

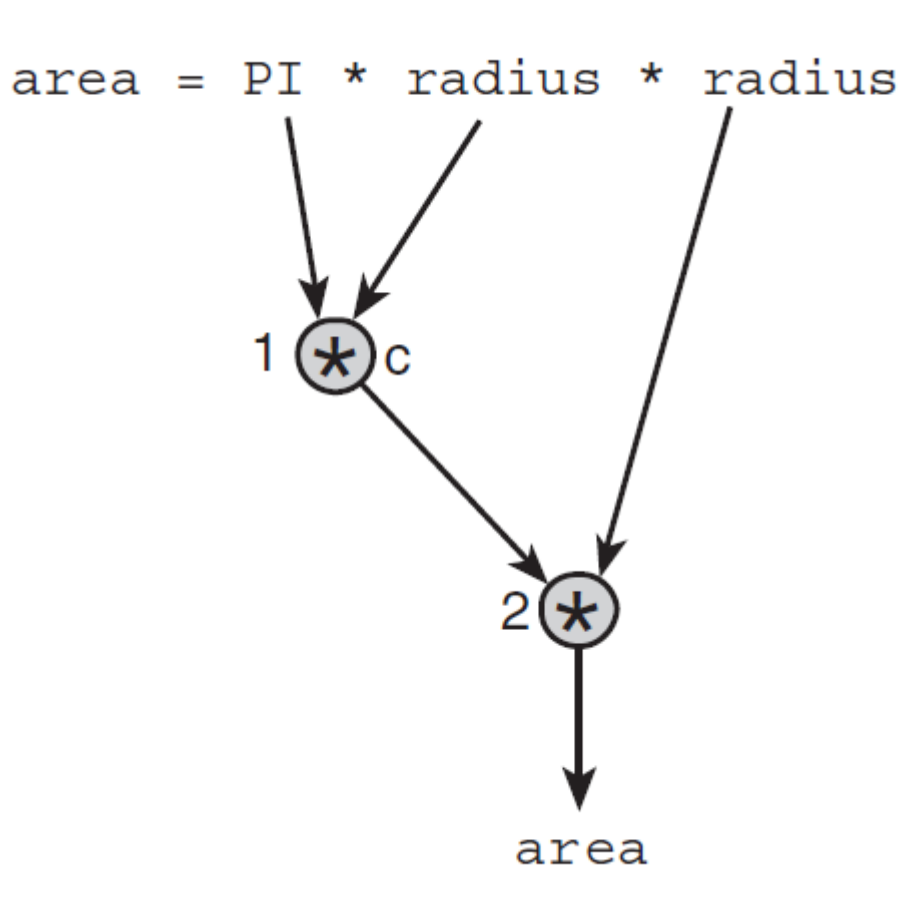
Expression with Multiple Operators

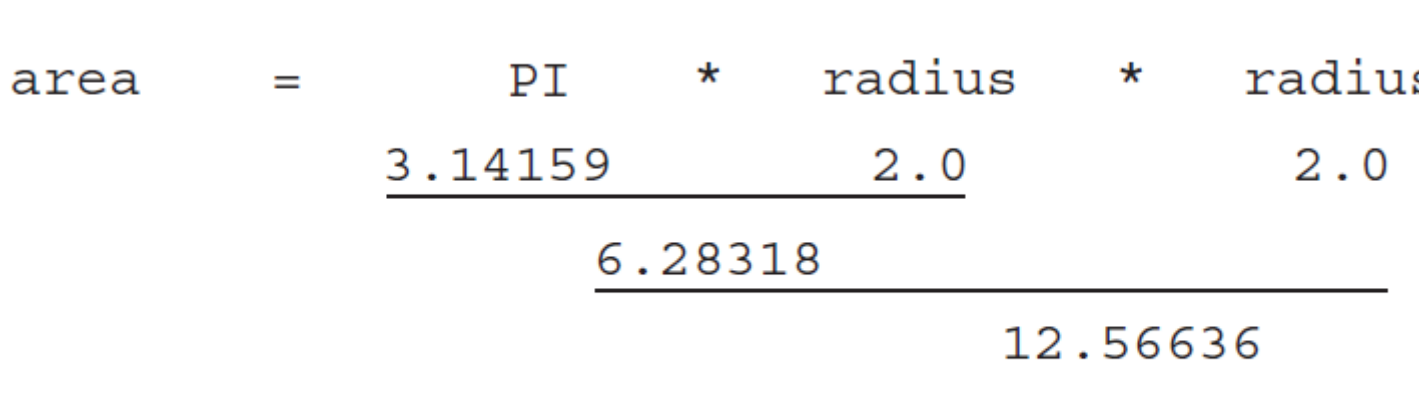
- Operators are of two types: **unary** and **binary**
- **Unary operators** take only one operand
  - Unary minus (**-**) and Unary plus (**+**) operators
- **Binary operators** take two operands
  - Examples: addition (**+**), subtraction (**–**), multiplication (*****), division (**/**) and integer remainder (**%**) operators
- A single expression could have multiple operators
  - Example: **-a + b*c – d/2**

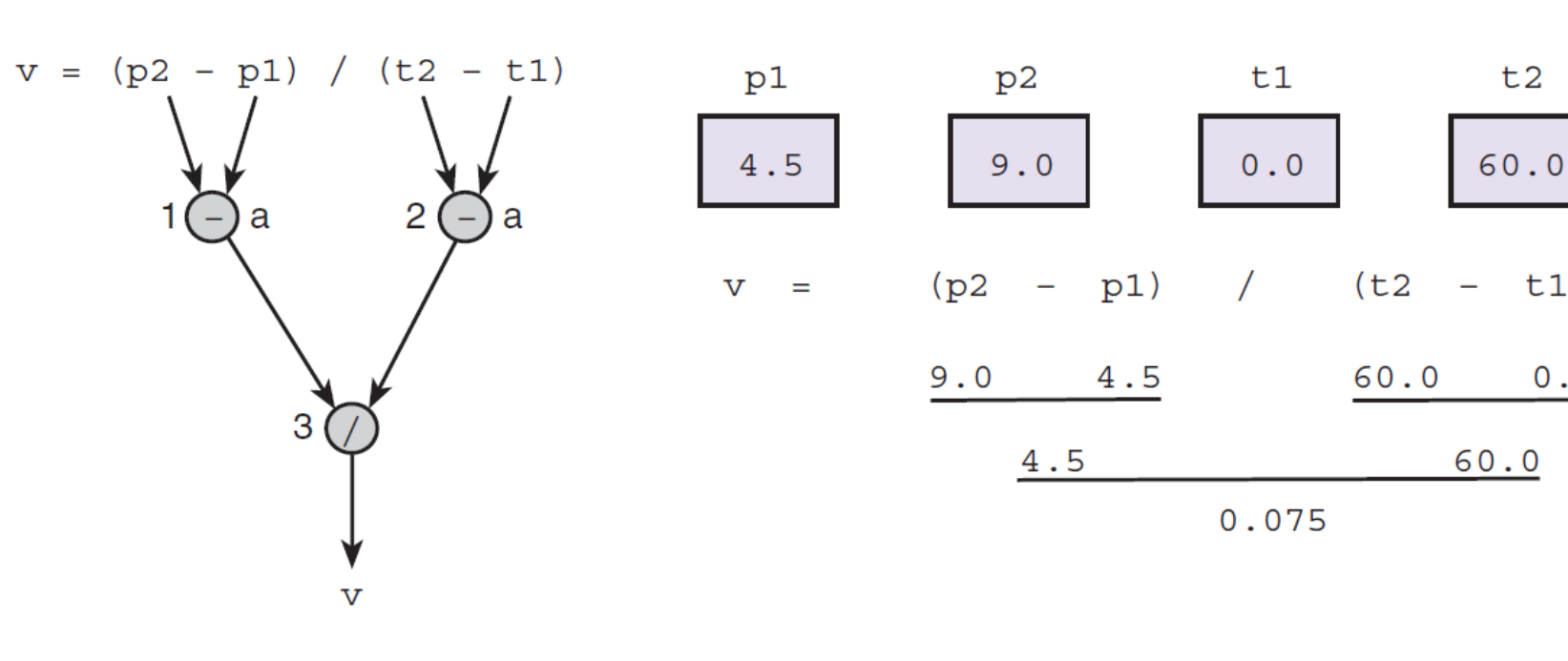
Rules for Evaluating Expressions

- **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 subexpression and at the same precedence level are **evaluated right to left**
  - **Binary operators** in the same subexpression and at the same precedence level are **evaluated left to right**

Step-By-Step Expression Evaluation







Key Points for Python Programmers

- Arithmetic expression in C are similar to Python:
  - +,-,*,/,%
  - +=, -=,*=,/=

Selection Structure

- In C, selection structures are similar to those in Python, but
  - parentheses are required around the expression,
  - semi-colon is required, if the body of the `if` is a single statement, and
  - indentation is not required, but highly recommended.

```c
if ( <condition> )
    statement;
```

```c
if ( <condition> )  {
    statement1;
    statement2;
    ...
}
```

- For example,

```c
if (x > 4)
        y = 7;
```

Key Points for Python Programmers

- Like Python, C allows an "else" clause:

```c
if (x <= 5)
        y = 8;
    else
        z = 10;
```

Key Points for Python Programmers

- C does not have the "elif" keyword. Instead, you must use "else if".

```c
if (y == 7)
        x = 2;
    else if (x != 2)
            y = 5;
     else
            z = 10;
```

Key Points for Python Programmers

- Python uses indentation to indicate a block of statement(s).
- C uses curly braces, { and } to delimit blocks of statements.
- Indentation is highly recommended.
- For example:

```
if ((x <= 5) && (y != 7)) {
        z = 12;
        w = 13;
    }
```

Key Points for Python Programmers

- **Relational Operators**

C Operator	Relational Operation
`<=`	Less than or equal
`<`	Less than
`>=`	Greater than or equal
`>`	Greater than
`==`	Equals
`!=`	Not Equal to

- **Logical Operators**

C Operator	Logical Operation
`!`	Logical Not
`&&`	Logical And
`||`	Logical Or

Logical Expression

- Condition that uses one or more logical operators

salary	children	temperature	humidity	n
1050	6	38.2	0.85	101

Logical Expression	Value
salary < 1000 || children > 4	
temperature > 35.0 && humidity > 0.90	
n >= 0 && n <= 100	
!(n >= 0 && n <= 100)	

Logical Expression

- Condition that uses one or more logical operators

salary	children	temperature	humidity	n
1050	6	38.2	0.85	101

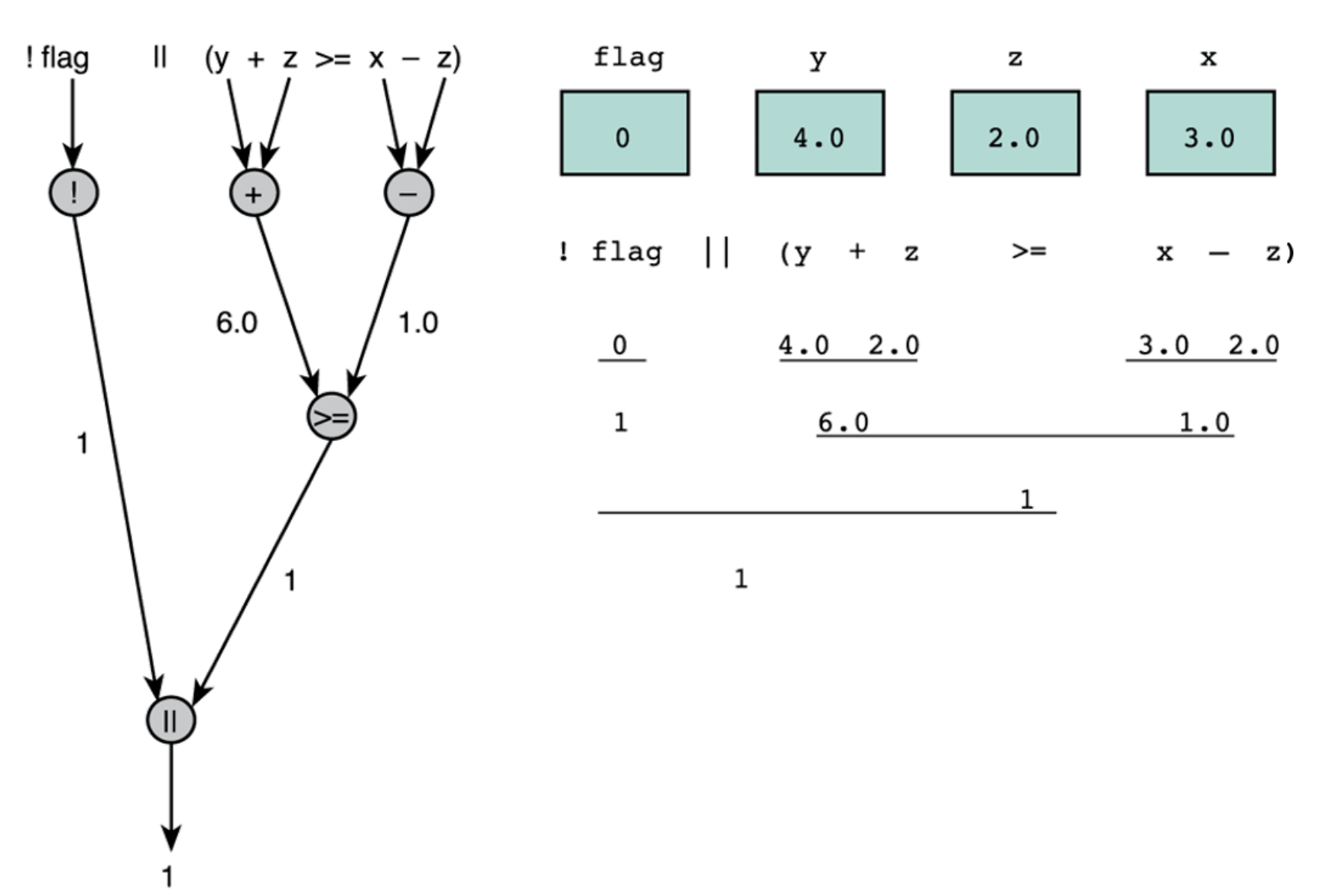
Logical Expression	Value
salary < 1000 || children > 4	true (1)
temperature > 35.0 && humidity > 0.90	false (0)
n >= 0 && n <= 100	false (0)
!(n >= 0 && n <= 100)	true (1)

Note: In the C Language,

- There is no **Boolean** type. Results of logical expressions are either `zero` (`false`) or `one` (`true`).
- Any nonzero value is considered to be `true`. For example:

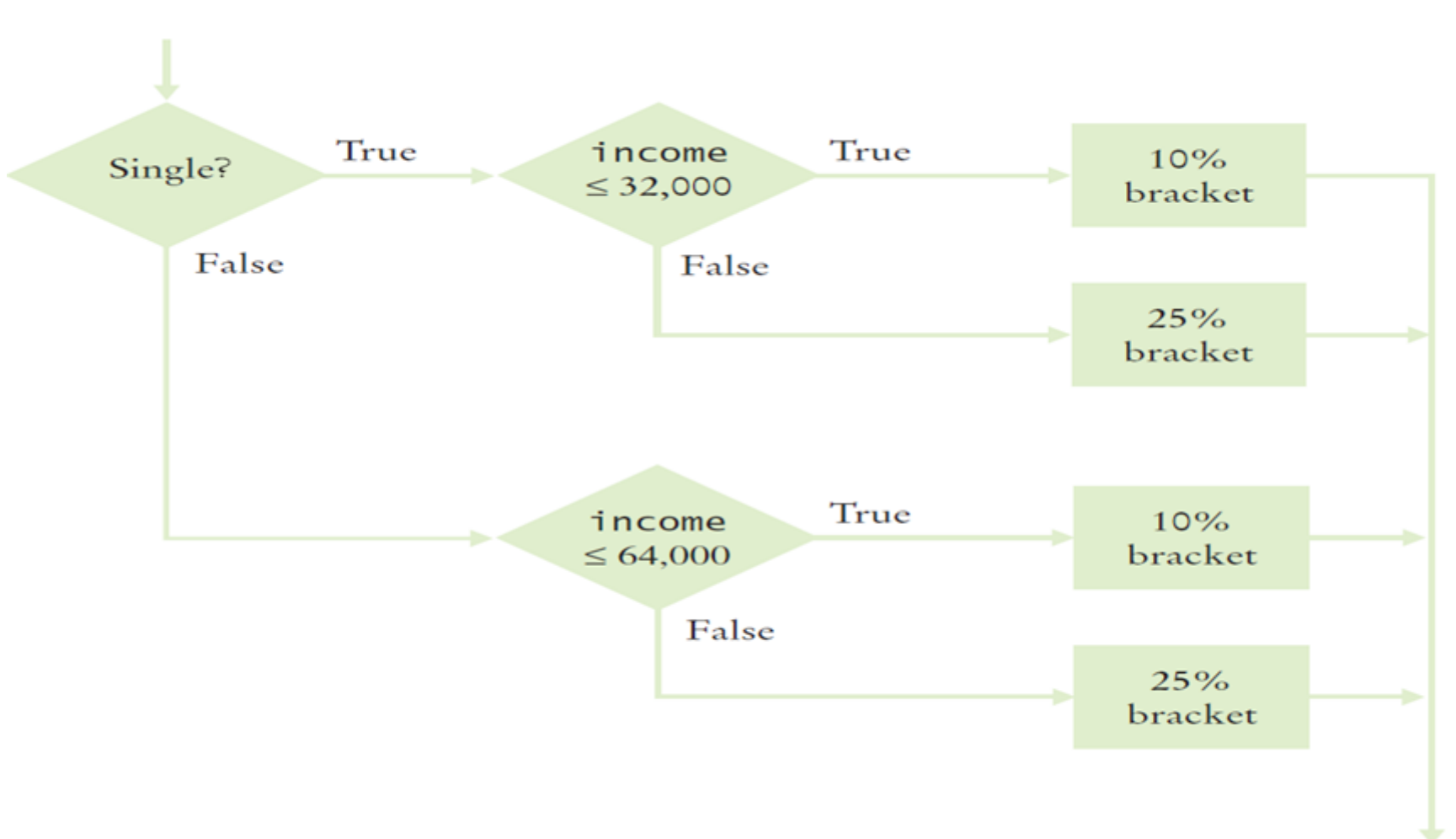
```
/* true vs. false */
#include <stdio.h>
int main(void)
{
  if (1+2)
     printf("1 + 2 is true. The value of the condition 1 + 2 == 3 is %d", 1+2 == 3);
  if (1-1)
     printf("I should not see this printf statement, since the value is 0");
  return 0;
}
```

Operator	Precedence
function calls	highest
! + - & (unary operators)	
* / %	
+ -	
< <= >= >	
== !=	
&& (logical AND)	
|| (logical OR)	
= (assignment operator)	lowest



Student Activity

Do You Remember This Example?



- Develop its implementation in C.

```
#include <stdio.h>
#define RATE1 0.10
#define RATE2 0.25
#define RATE1_SINGLE_LIMIT 32000.0
#define RATE1_MARRIED_LIMIT 64000.0
int main() {
    double income;
    char maritalStatus;

    double tax1, tax2;
    double totalTax;

    /* read income and marital status */
    printf("Please enter your income: ");
    scanf("%lf", &income);
    printf("Please enter s for single, m for married: ");
    scanf(" %c", &maritalStatus);
    /* The space before %c is to consume any whitespace before reading the character */

    /* Compute taxes due */
    tax1 = 0;
    tax2 = 0;

    /* Enter the if statement[s] here */

    totalTax = tax1 + tax2;

    printf("The tax is $%.2f\n",totalTax);
 return 0;
 }
```

Loops

While loop

- The C `while` loop will be familiar to you, and shares the same syntax rules with the C `if` statement:
  - Parentheses are required for the expression.
  - Curly braces delimit a block (required for compound statements)
- For example,

```
while (x < 5) {
        y = z + x;
        x = x + 1;
    }
```

Example

```
#include <stdio.h>


int
main(void)
{
    double total_pay; /* company payroll */
    int count_emp; /* current employee */
    int number_emp; /* number of employees */
    double hours; /* hours worked */
    double rate; /* hourly rate */
    double pay; /* pay for this period */

    /* Get number of employees. */
    printf("Enter number of employees> ");
    scanf("%d", &number_emp);

    /* Compute each employee's pay and add it to the payroll. */
    total_pay = 0.0;
    count_emp = 0;
    while (count_emp < number_emp) {
        printf("Hours> ");
        scanf("%lf", &hours);
        printf("Rate > $");
        scanf("%lf", &rate);
        pay = hours * rate;
        printf("Pay is $%6.2f\n\n", pay);
        total_pay = total_pay + pay; /* Add next pay. */
        count_emp = count_emp + 1;
    }

    printf("All employees processed\n");
    printf("Total payroll is $%8.2f\n", total_pay);
    return (0);
}
```

For Loop

- The "for" loop in C is **quite different** from Python.

In [ ]:
```
for x in range(5):
    print("Inside the loop, x =",x)
print("Outside the loop, x =",x)
```

```
#include <stdio.h>
int main() {
    int x;
    for (x=0; x <= 4; x++)
        printf("Inside the for loop, x = %d\n", x);
    printf("Outside the for loop, x = %d\n", x);
    return 0;
}
```

- A `for` loop in C has three control components in addition to the loop body:
  - initialization of the loop control variable,
  - testing of the loop repetition condition, and
  - changing (updating) of the loop control variable.
- The `for` statement in C supplies a designated place for each of these three components.

Program Style - Formatting the for Statement

- For clarity, we usually place each expression of the `for` heading on a separate line.
- If all three expressions are very short, we may place them together on one line.
- For example,

```
/* Compute the payroll for a company */
#include <stdio.h>
int
main(void)
{
    double total_pay; /* company payroll */
    int count_emp; /* current employee */
    int number_emp; /* number of employees */
    double hours; /* hours worked */
    double rate; /* hourly rate */
    double pay; /* pay for this period */

    /* Get number of employees. */
    printf("Enter number of employees> ");
    scanf("%d", &number_emp);

    /* Compute each employee's pay and add it to the payroll. */
    total_pay = 0.0;
    for (count_emp = 0;  /* initialization */
         count_emp < number_emp;  /* loop repetition condition */
         count_emp += 1) {  /* update */
        printf("Hours> ");
        scanf("%lf", &hours);
        printf("Rate > $");
        scanf("%lf", &rate);
        pay = hours * rate;
        printf("Pay is $%6.2f\n\n", pay);
        total_pay = total_pay + pay; /* Add next pay. */
    }
    printf("All employees processed\n");
    printf("Total payroll is $%8.2f\n", total_pay);
return (0);
}
```

Increment and Decrement Operators

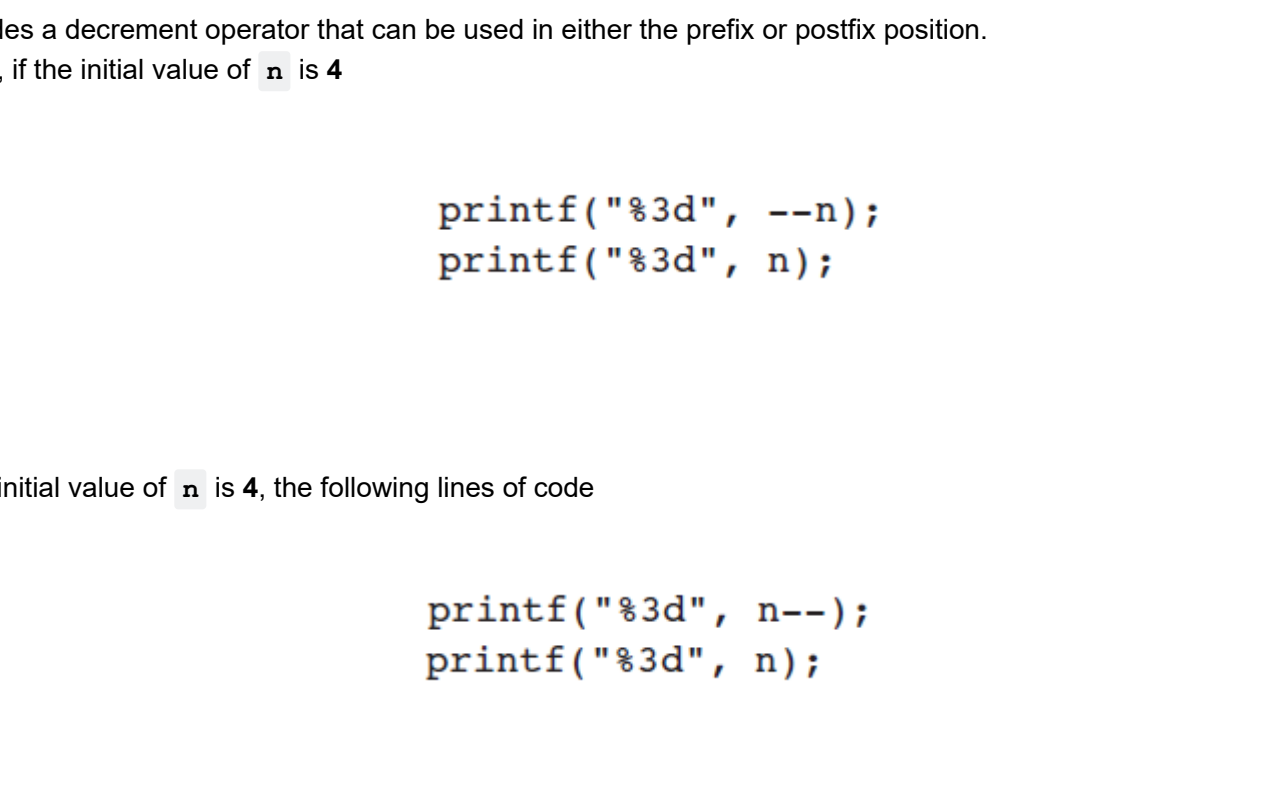
- The for loops examples we have seen all included assignment expression of the form

```
counter = counter + 1;
    // OR
    counter += 1;
    // OR
    counter ++;
```

- The increment operator **++** takes a single variable as its operand.
- The **side effect** of applying the **++** operator is that the value of its operand is incremented by one.
- Frequently, **++** is used just for this side effect, as in the following loop in which the variable `counter` is to run from **0** up to `limit`:

```
for (counter = 0; counter < limit; counter++) {
        ...
    }
```

- The value of the expression in which the **++** operator is used depends on the **position** of the operator.
- When the **++** is placed immediately **in front of** its operand (prefix increment), the value of the expression is the variable's value **after** incrementing.
- When the **++** comes immediately **after** the operand (postfix increment), the expression's value is the value of the variable **before** it is incremented.
- Compare the action of the two code segments in following figure, given an initial value of **2** in `i`.



- C also provides a decrement operator that can be used in either the prefix or postfix position.
- For example, if the initial value of `n` is **4**

```
printf("%3d", --n);
printf("%3d", n);
```

print

```
3  3
```

- Again, if the initial value of `n` is **4**, the following lines of code

```
printf("%3d", n--);
printf("%3d", n);
```

print

```
4  3
```

Example

- Function factorial computes the factorial of an integer represented by the formal parameter `n`.
- The loop body executes for decreasing values of `i` from `n` down to **2**, and each value of `i` is incorporated in the accumulating product.
- The loop exit occurs when `i` is **1**.

FIGURE 5.7 Function to Compute Factorial

```
/*
 * Computes n!
 * Pre: n is greater than or equal to zero
 */
int
factorial(int n)
{
    int i,          /* local variables */
        product;    /* accumulator for product computation */

    product = 1;
    /* Computes the product n x (n-1) x (n-2) x . . . x 2 x 1 */
    for (i = n; i > 1; --i) {
         product = product * i;
    }

    /* Returns function result */
    return (product);
}
```

```
#include <stdio.h>


int factorial (int n) ; /* function prototype (declaration) */


int
main(void)
{
    int x, fact_x;
    printf("Enter the value of x: ");
    scanf("%d", &x);
    fact_x = factorial(x);
    printf("The factorial of %d = %d\n", x, fact_x);
    return (0);
}

/*
 * Computes n!
 * Pre: n is greater than or equal to zero
 */
int
factorial(int n)
{
    int i, /* local variables */
        product; /* accumulator for product computation */


    product = 1;


    /* Computes the product n x (n-1) x (n-2) x . . . x 2 x 1 */
    for (i = n; i > 1; --i) {
        product = product * i;
    }
    /* Returns function result */
    return (product);
}
```

Nested Loops

- Consist of an outer loop with one or more inner loops.
- Each time the outer loop is repeated, the inner loop is reentered and executed.
- Example

```
void stars(int n) {
  int i, j;
  for (i=1; i<=n; i++) {
    for (j=1; j<=i; j++) {
      printf("*");
    }
    printf("\n");
  }
}
```

outer loop

inner loop

```
stars(5);

*
**
***
****
*****
```

```
#include <stdio.h>

void stars (int n); /* Function prototype (declaration) */

int
main(void)
{
    int x;
    printf("Enter the maximum number of stars: ");
    scanf("%d", &x);
    printf("\nThe result of stars(%d) is:\n", x);
    stars(x);
    return (0);
}

void stars (int n) {
    int i, j;
    for (i=1; i<=n; i++) {
        for (j=1; j<=i; j++) {
            printf("*");
        }
        printf("\n");
    }
}
```

The do-while Statement

- The for and while statements evaluate the loop condition before the execution of the loop body.
- The do-while statement evaluates the loop condition after the execution of the loop body
- Syntax:

do-while Statement

SYNTAX:
```
do
        statement;
while (loop repetition condition);
```

EXAMPLE:
```
/* Find first even number input */
do
        status = scanf("%d", &num);
while (status > 0   &&   (num % 2) != 0);
```

INTERPRETATION: First, the *statement* is executed. Then, the *loop repetition condition* is tested, and if it is true, the *statement* is repeated and the *condition* retested. When this condition is tested and found to be false, the loop is exited and the next statement after the `do-while` is executed.

Note: If the loop body contains more than one statement, the group of statements must be surrounded by braces.

Using do-while to Repeat Program

```
#include <stdio.h>

int
main(void)
{
    char ch;
    ch = 'y';            /* User response [y/n] */
    do {
        printf("%c\n", ch); /* display character */
        printf("Repeat again [y/n]? ");
        scanf(" %c",&ch); /* read character */
    } while (ch=='y' || ch == 'Y');
    return (0);
}
```