Figure 1.12 Entering, Translating, and Running a High-Level Language Program

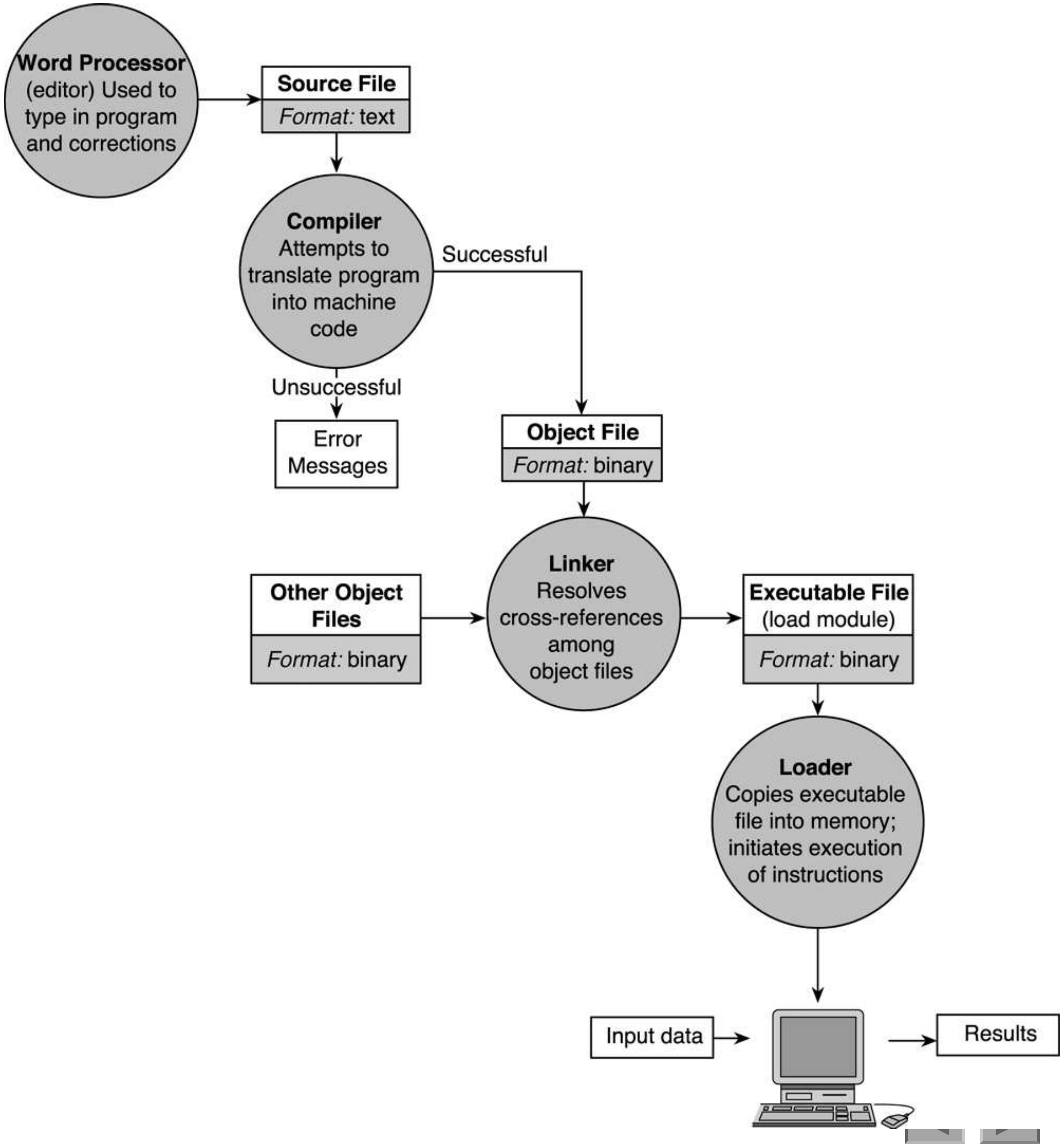
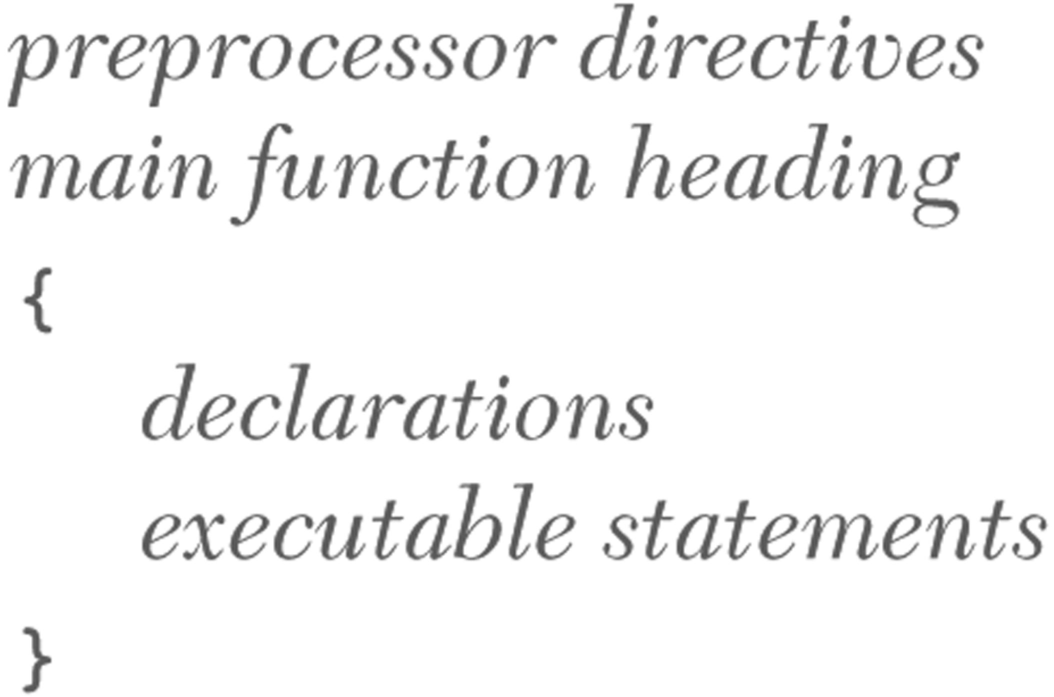
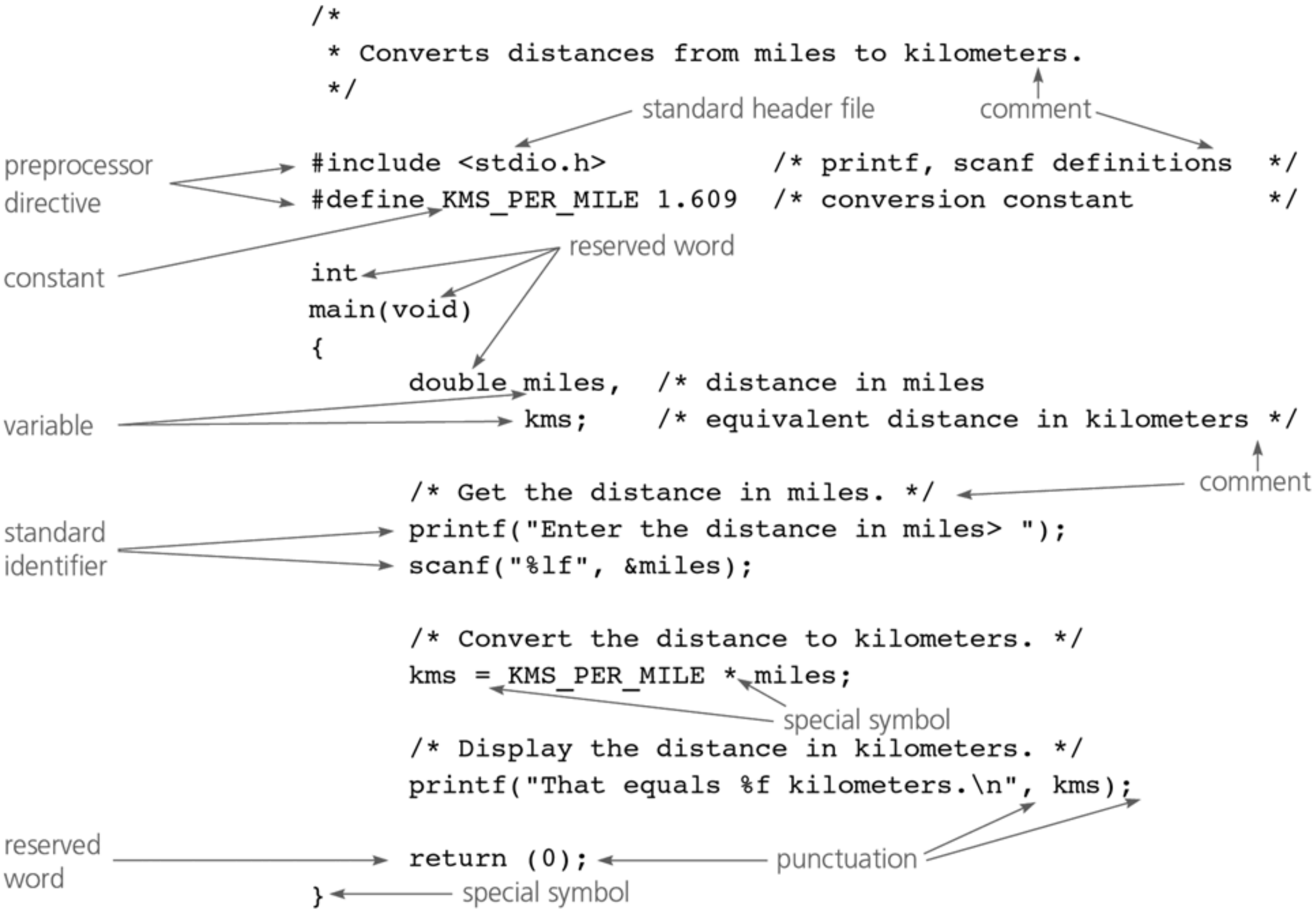

Word Processor (editor) Used to type in program and corrections
Source File
Format: text
Compiler Attempts to translate program into machine code
Successful
Unsuccessful
Error Messages
Object File
Format: binary
Other Object Files
Format: binary
Linker Resolves cross-references among object files
Executable File (load module)
Format: binary
Loader Copies executable file into memory; initiates execution of instructions
Input data
Results

Figure 2.7
General Form of a C Program



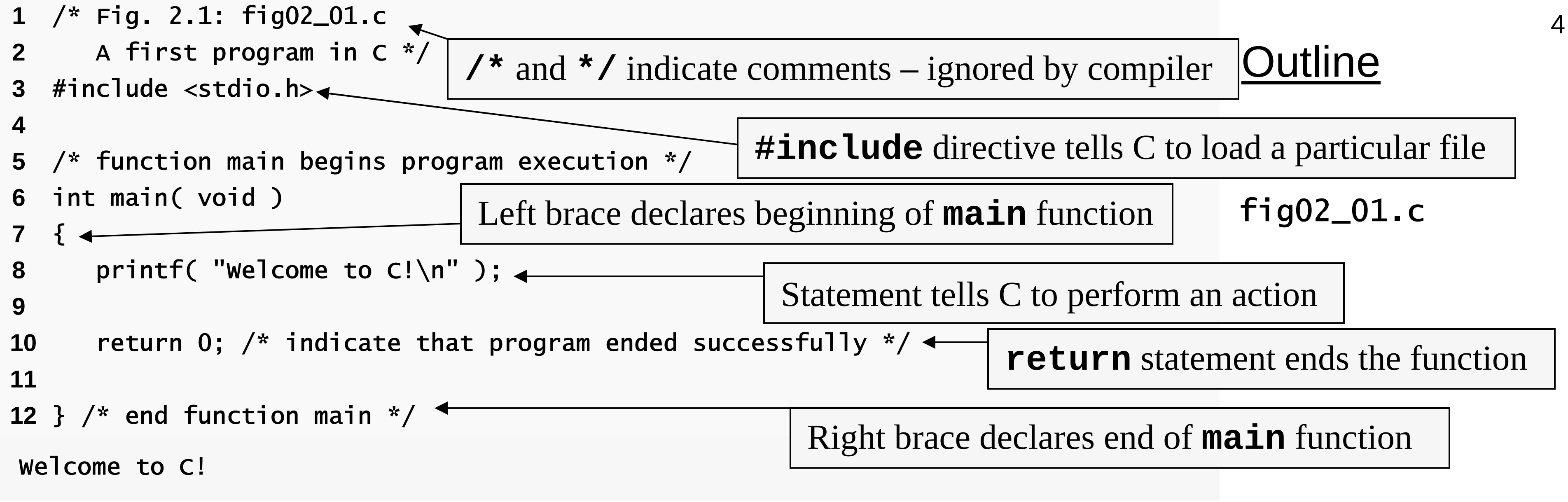
```
preprocessor directives
main function heading
{
    declarations
    executable statements
}
```

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



```
/*
 * Converts distances from miles to kilometers.
 */

#include <stdio.h>          /* printf, scanf definitions   */
#define KMS_PER_MILE 1.609  /* conversion constant         */

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

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

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

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

      return (0);
}
```

fig02_01.c



```
/* Fig. 2.1: fig02_01.c
   A first program in C */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   printf( "Welcome to C!\n" );

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

} /* end function main */
```

```
Welcome to C!
```

`/*` and `*/` indicate comments – ignored by compiler

`#include` directive tells C to load a particular file

Left brace declares beginning of `main` function

Statement tells C to perform an action

`return` statement ends the function

Right brace declares end of `main` function

2.2 A Simple C Program: Printing a Line of Text

Comments

- **Text surrounded by `/*` and `*/` is ignored by computer**
- **Used to describe program**

▪ `#include <stdio.h>`

- **Preprocessor directive**
  - **Tells computer to load contents of a certain file**
- **`<stdio.h>` allows standard input/output operations**

Common Programming Error 2.1

Forgetting to terminate a comment with `*/`.

Common Programming Error 2.2

Starting a comment with the characters `*/` or ending a comment with the characters `/*`.

2.2 A Simple C Program: Printing a Line of Text

- `int main()`
  - C programs contain one or more functions, exactly one of which must be `main`
  - Parenthesis used to indicate a function
  - `int` means that `main` "returns" an integer value
  - Braces (`{` and `}`) indicate a block
    - The bodies of all functions must be contained in braces

Good Programming Practice 2.1

Every function should be preceded by a comment describing the purpose of the function.

2.2 A Simple C Program: Printing a Line of Text

- `printf( "Welcome to C!\n" );`
  - Instructs computer to perform an action
    - Specifically, prints the string of characters within quotes (" ")
  - Entire line called a statement
    - All statements must end with a semicolon (`;`)
  - Escape character (\)
    - Indicates that printf should do something out of the ordinary
    - `\n` is the newline character

Escape sequence	Description
`\n`	Newline. Position the cursor at the beginning of the next line.
`\t`	Horizontal tab. Move the cursor to the next tab stop.
`\a`	Alert. Sound the system bell.
`\\`	Backslash. Insert a backslash character in a string.
`\"`	Double quote. Insert a double-quote character in a string.

Fig. 2.2 | Some common escape sequences.

Common Programming Error 2.3

Typing the name of the output function `printf` as `print` in a program.

2.2 A Simple C Program: Printing a Line of Text

- `return 0;`
  - A way to exit a function
  - `return 0`, in this case, means that the program terminated normally
- Right brace `}`
  - Indicates end of `main` has been reached
- Linker
  - When a function is called, linker locates it in the library
  - Inserts it into object program
  - If function name is misspelled, the linker will produce an error because it will not be able to find function in the library

Good Programming Practice 2.2

Add a comment to the line containing the right brace, `}`, that closes every function, including `main`.

◀ ▶

Good Programming Practice 2.3

The last character printed by a function that displays output should be a newline (`\n`). This ensures that the function will leave the screen cursor positioned at the beginning of a new line. Conventions of this nature encourage software reusability—a key goal in software development environments.

Good Programming Practice 2.4

Indent the entire body of each function one level of indentation (we recommend three spaces) within the braces that define the body of the function. This indentation emphasizes the functional structure of programs and helps make programs easier to read.

Good Programming Practice 2.5

Set a convention for the size of indent you prefer and then uniformly apply that convention. The tab key may be used to create indents, but tab stops may vary. We recommend using three spaces per level of indent.

fig02_03.c

```
/* Fig. 2.3: fig02_03.c
   Printing on one line with two printf statements */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   printf( "Welcome " );
   printf( "to C!\n" );

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

} /* end function main */
```

printf statement starts printing from where the last statement ended, so the text is printed on one line.

```
Welcome to C!
```

```c
/* Fig. 2.4: fig02_04.c
   Printing multiple lines with a single printf */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   printf( "Welcome\nto\nC!\n" );

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

} /* end function main */
```

Newline characters move the cursor to the next line

```
Welcome
to
C!
```

fig02_04.c

Outline

fig02_05.c

```
/* Fig. 2.5: fig02_05.c
   Addition program */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int integer1; /* first number to be input by user  */
   int integer2; /* second number to be input by user */
   int sum;      /* variable in which sum will be stored */

   printf( "Enter first integer\n" ); /* prompt */
   scanf( "%d", &integer1 );          /* read an integer */

   printf( "Enter second integer\n" ); /* prompt */
   scanf( "%d", &integer2 );           /* read an integer */

   sum = integer1 + integer2; /* assign total to sum */

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

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

} /* end function main */
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Definitions of variables

`scanf` obtains a value from the user and assigns it to **`integer1`**

`scanf` obtains a value from the user and assigns it to **`integer2`**

Assigns a value to **`sum`**

2.3 Another Simple C Program: Adding Two Integers

- **As before**
  - **Comments,** `#include <stdio.h>` **and** `main`
- `int integer1, integer2, sum;`
  - **Definition of variables**
    - **Variables: locations in memory where a value can be stored**
  - `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) and underscores(_)**

      Case sensitive
  - **Definitions appear before executable statements**
    - **If an executable statement references and undeclared variable it will produce a syntax (compiler) error**

Common Programming Error 2.4

Using a capital letter where a lowercase letter should be used (for example, typing `Main` instead of `main`).

Portability Tip 2.1

Use identifiers of 31 or fewer characters. This helps ensure portability and can avoid some subtle programming errors.

Good Programming Practice 2.6

Choosing meaningful variable names helps make a program self-documenting, i.e., fewer comments are needed.

◀ ▶

Good Programming Practice 2.7

The first letter of an identifier used as a simple variable name should be a lowercase letter. Later in the text we will assign special significance to identifiers that begin with a capital letter and to identifiers that use all capital letters.

Good Programming Practice 2.8

Multiple-word variable names can help make a program more readable. Avoid run-ning the separate words together as in `totalcommissions`. Rather, separate the words with underscores as in `total_commissions`, or, if you do wish to run the words together, begin each word after the first with a capital letter as in `totalCommissions`. The latter style is preferred.

◀ ▶

Common Programming Error 2.5

Placing variable definitions among executable statements causes syntax errors.

Good Programming Practice 2.9

Separate the definitions and executable statements in a function with one blank line to emphasize where the definitions end and the executable statements begin.

2.3 Another 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
    - & is confusing in beginning – for now, just remember to include it with the variable name in `scanf` statements
  - When executing the program the user responds to the `scanf` statement by typing in a number, then pressing the *enter* (return) key

Good Programming Practice 2.10

Place a space after each comma (,) to make programs more readable.

2.3 Another 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;
    ```

  - Variable receiving value on left
- `printf( "Sum is %d\n", sum );`
  - Similar to `scanf`
    - `%d` means decimal integer will be printed
    - `sum` specifies what integer will be printed
  - Calculations can be performed inside `printf` statements

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

Good Programming Practice 2.11

Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more readable.

Common Programming Error 2.6

A calculation in an assignment statement must be on the right side of the = operator. It is a syntax error to place a calculation on the left side of an assignment operator.

Common Programming Error 2.7

Forgetting one or both of the double quotes surrounding the format control string in a `printf` or `scanf`.

Common Programming Error 2.8

Forgetting the `%` in a conversion specification in the format control string of a `printf` or `scanf`.

Common Programming Error 2.9

Placing an escape sequence such as `\n` outside the format control string of a `printf` or `scanf`.

Common Programming Error 2.10

Forgetting to include the expressions whose values are to be printed in a `printf` containing conversion specifiers.

Common Programming Error 2.11

Not providing a conversion specifier when one is needed in a `printf` format control string to print the value of an expression.

Common Programming Error 2.12

Placing inside the format control string the comma that is supposed to separate the format control string from the expressions to be printed.

Common Programming Error 2.13

Forgetting to precede a variable in a `scanf` statement with an ampersand when that variable should, in fact, be preceded by an ampersand.

Common Programming Error 2.14

Preceding a variable included in a `printf` statement with an ampersand when, in fact, that variable should not be preceded by an ampersand.

2.4 Memory Concepts

- **Variables**
  - **Variable names correspond to locations in the computer's memory**
  - **Every variable has a name, a type, a size and a value**
  - **Whenever a new value is placed into a variable (through `scanf`, for example), it replaces (and destroys) the previous value**
  - **Reading variables from memory does not change them**

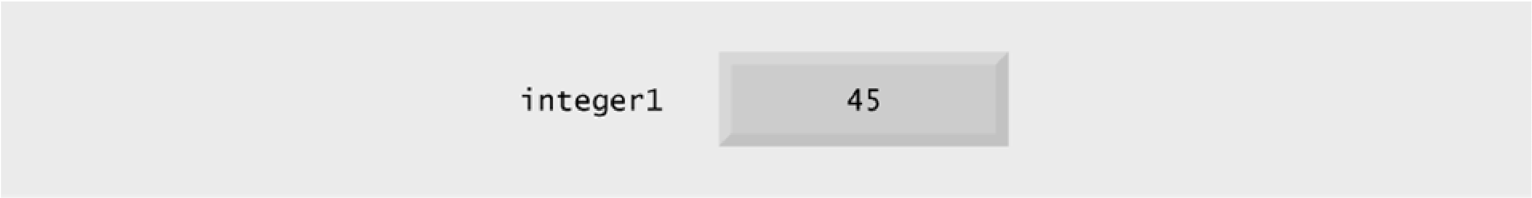


Fig. 2.6 | Memory location showing the name and value of a variable.

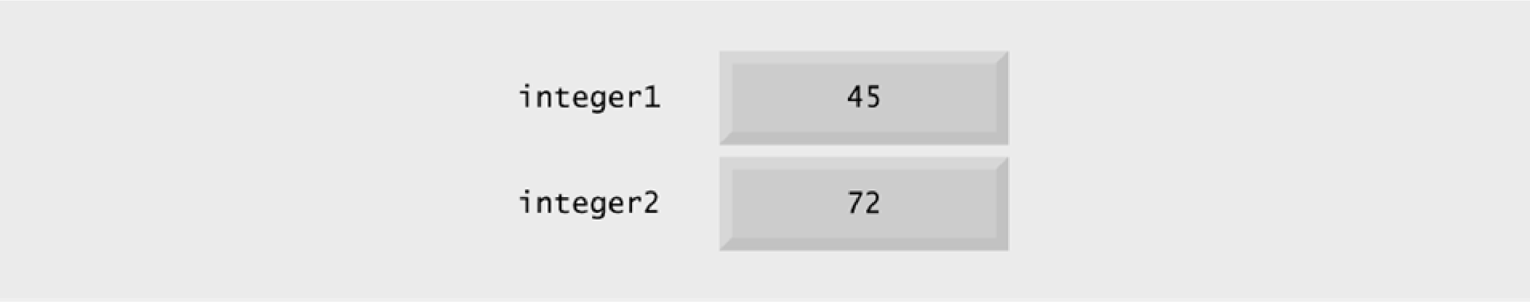
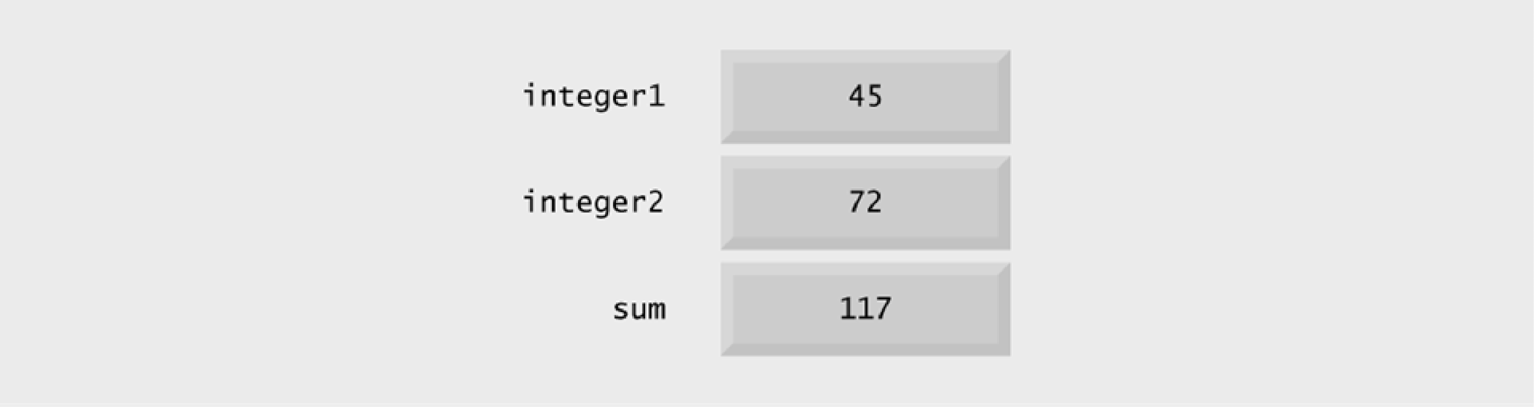


Fig. 2.7 | Memory locations after both variables are input.



integer1 45

integer2 72

sum 117

Fig. 2.8 | Memory locations after a calculation.

C opetration	Arithmetic operator	Algebraic expression	C expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm	`b * m`
Division	`/`	x/y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	`%`	$r \bmod s$	`r % s`

Fig. 2.9 | Arithmetic operators.

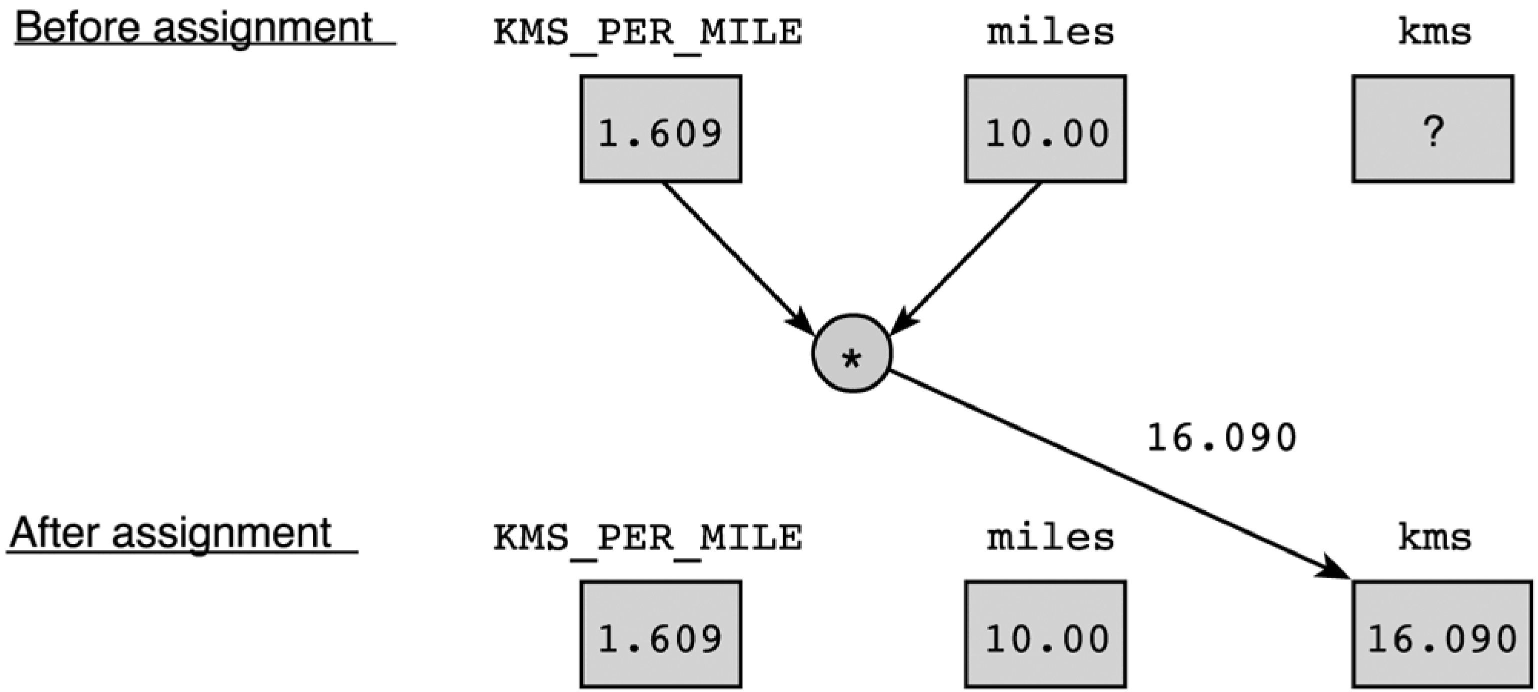
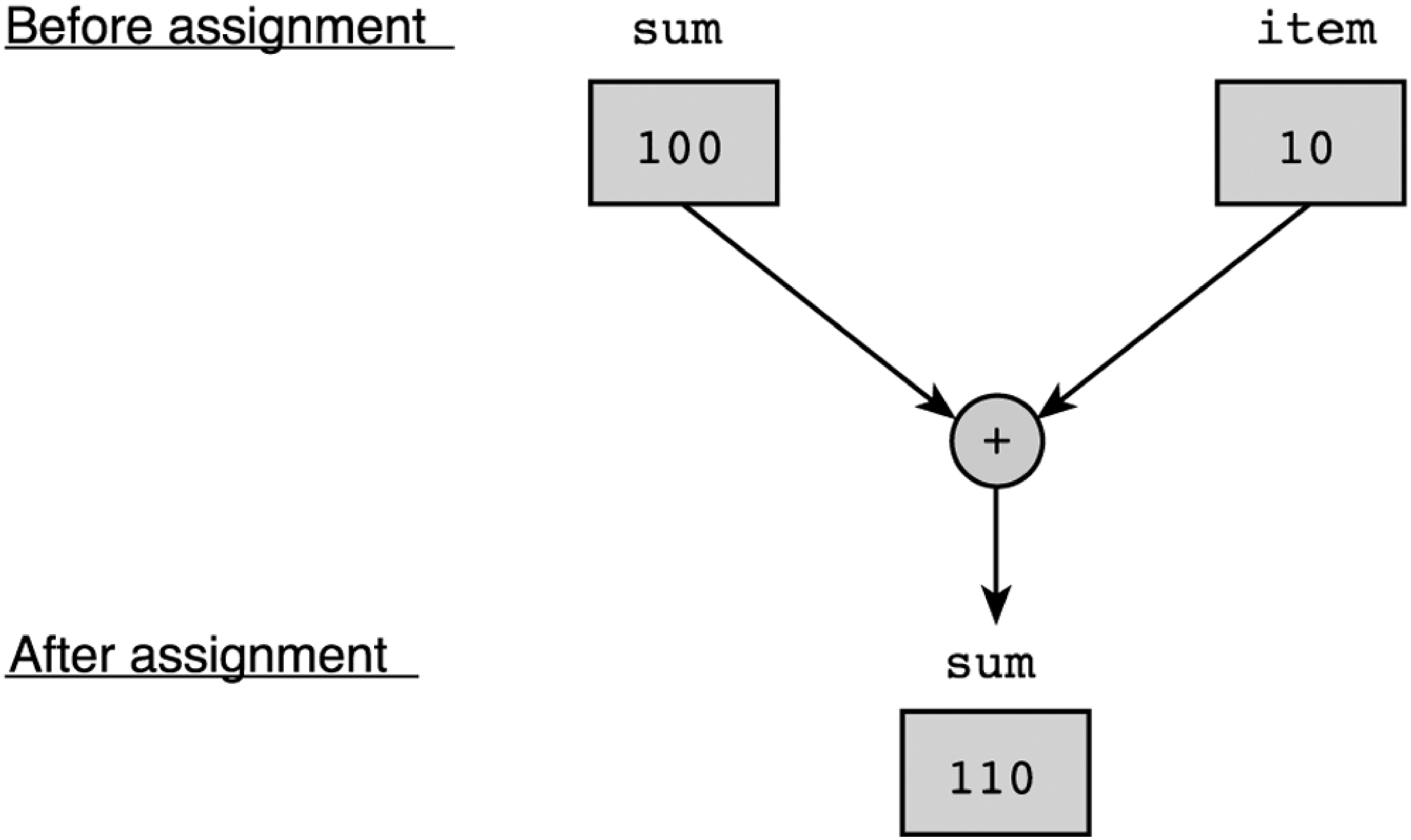


Figure 2.3 Effect of kms = KMS_PER_MILE * miles;



Before assignment

sum

100

item

10

+

After assignment

sum

110

Figure 2.4 Effect of sum = sum + item;

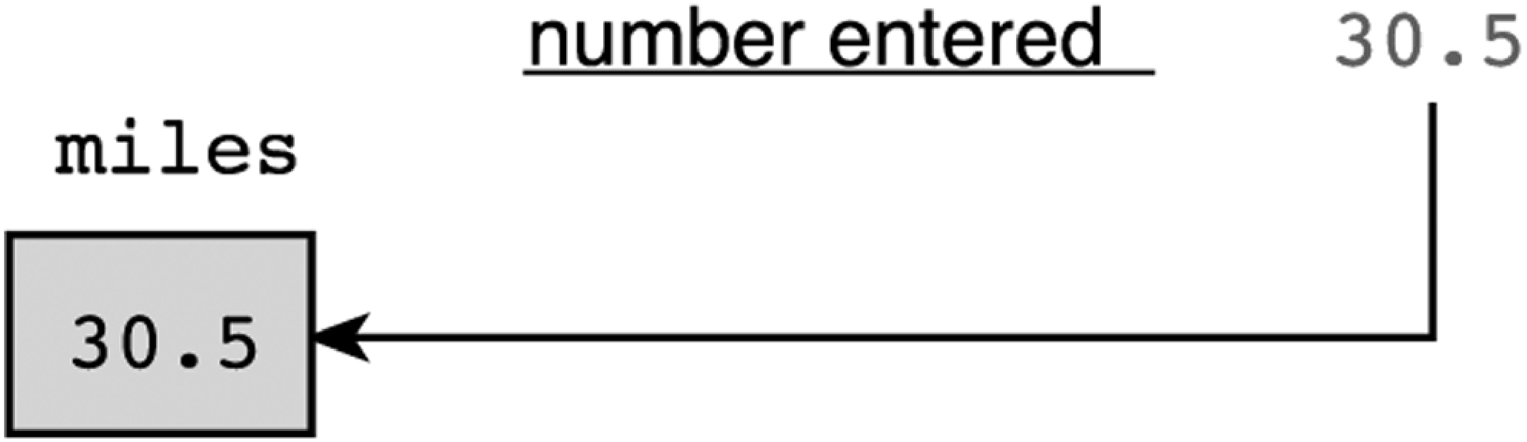


Figure 2.5 Effect of scanf("%lf", &miles);

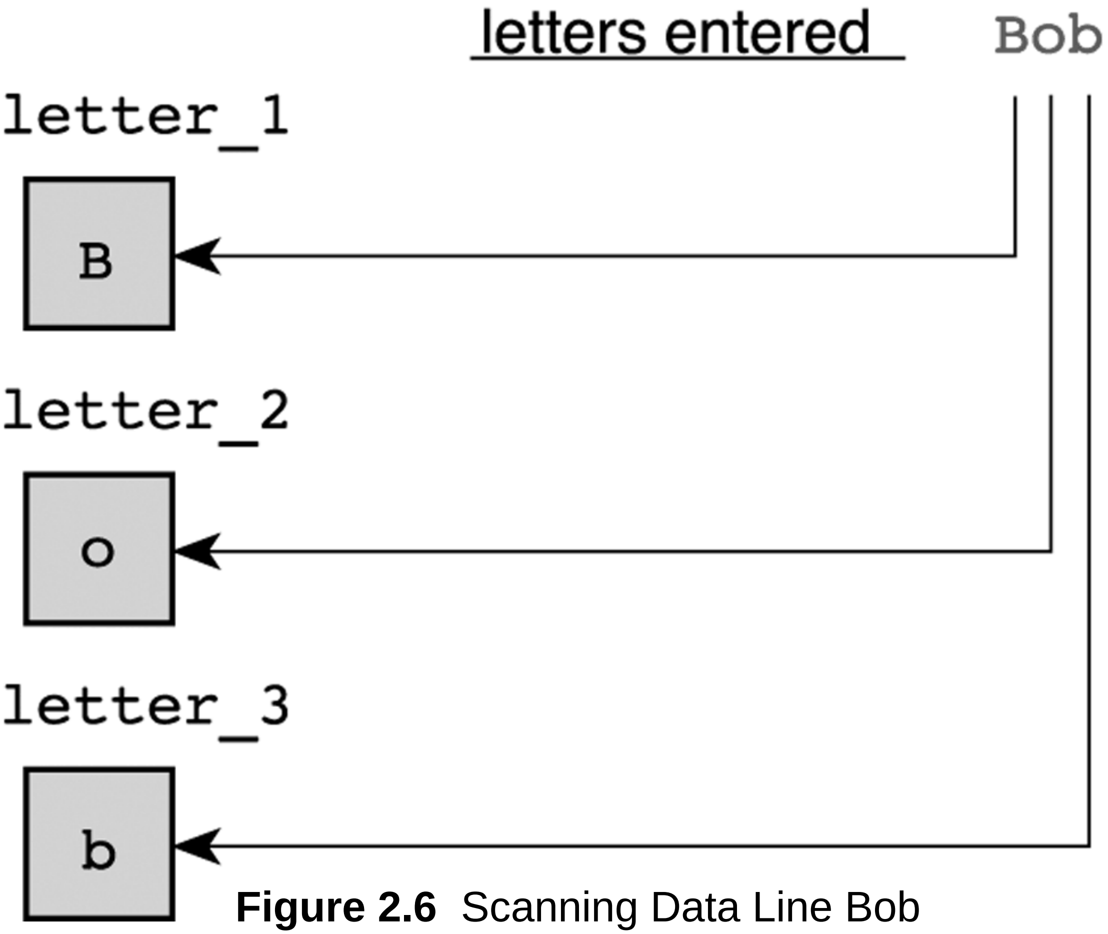


Figure 2.6 Scanning Data Line Bob

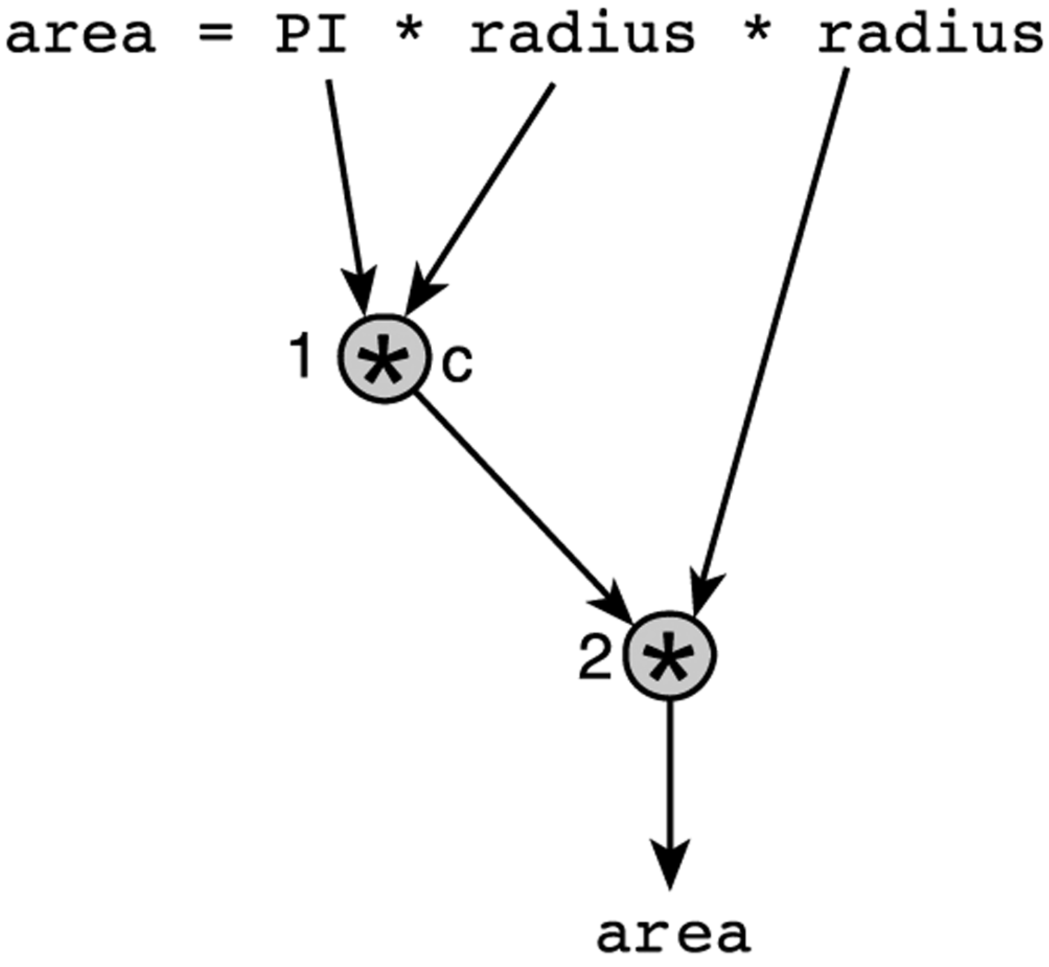
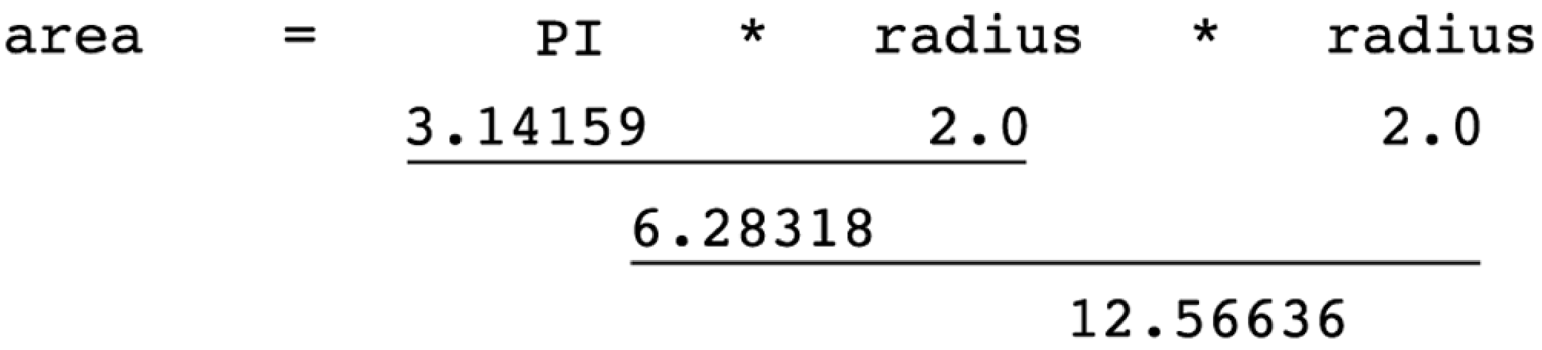


Figure 2.8 Evaluation Tree for
area = PI * radius * radius;



area = PI * radius * radius

3.14159 2.0 2.0

6.28318

12.56636

Figure 2.9 Step-by-Step Expression Evaluation

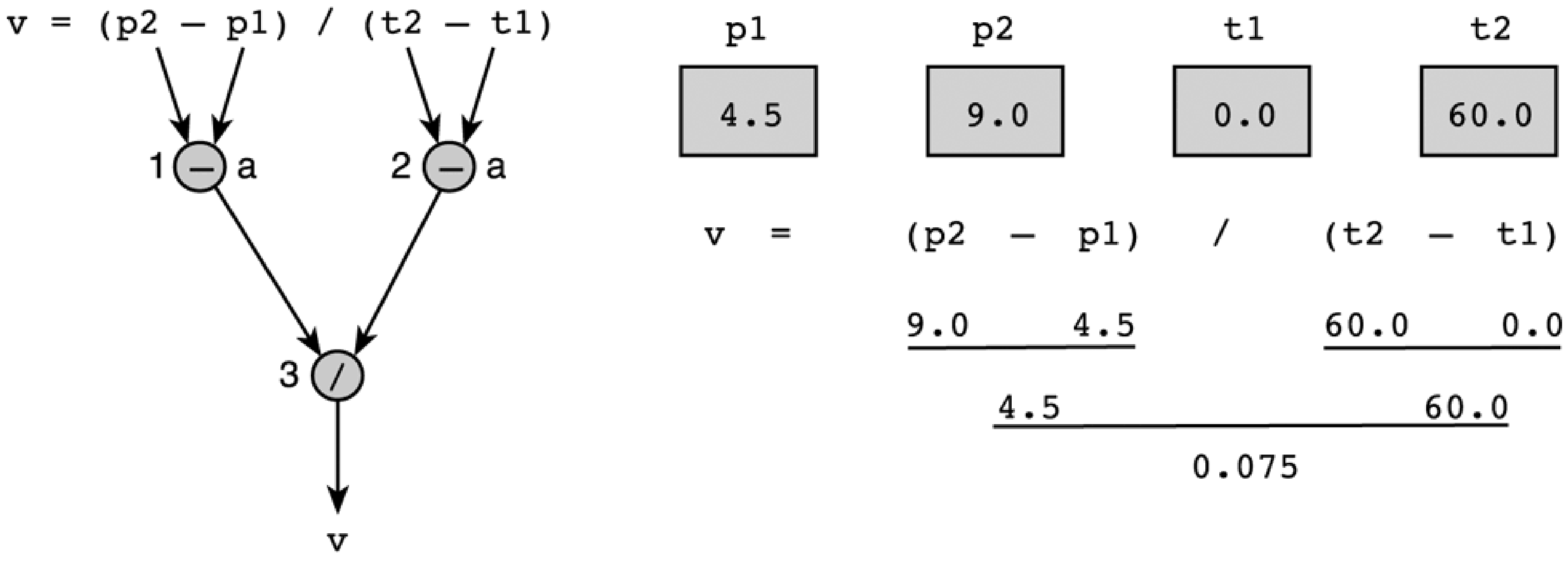


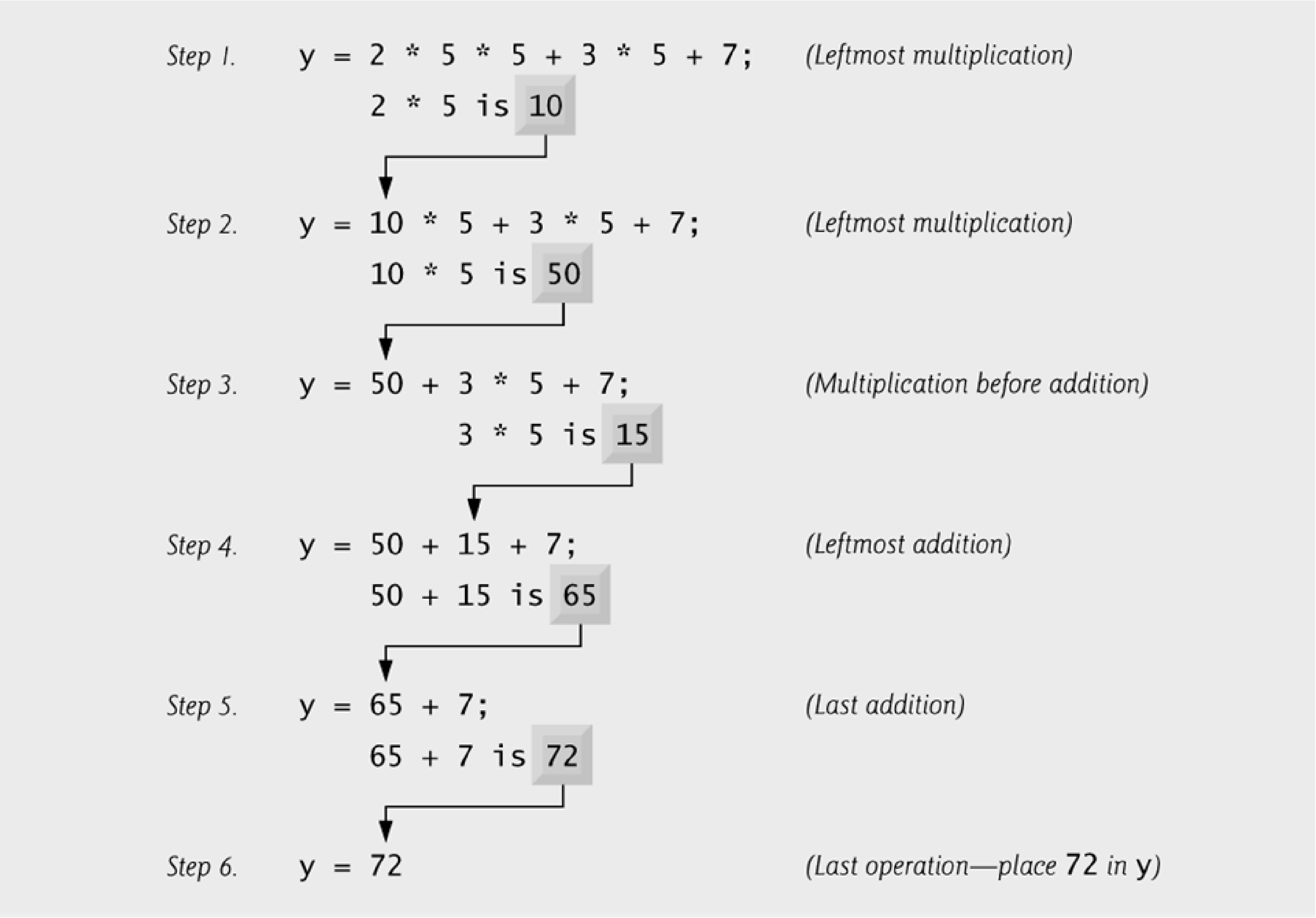
Figure 2.10 Evaluation Tree and Evaluation for v = (p2 - p1) / (t2 - t1);

2.5 Arithmetic

- **Arithmetic calculations**
  - **Use `*` for multiplication and `/` for division**
  - **Integer division truncates remainder**
    - **`7 / 5` evaluates to `1`**
  - **Modulus operator(%) returns the remainder**
    - **`7 % 5` evaluates to `2`**
- **Operator precedence**
  - **Some arithmetic operators act before others (i.e., multiplication before addition)**
    - **Use parenthesis when needed**
  - **Example: Find the average of three variables `a`, `b` and `c`**
    - **Do not use: `a + b + c / 3`**
    - **Use: `(a + b + c ) / 3`**

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

Fig. 2.10 | Precedence of arithmetic operators.



Step 1. `y = 2 * 5 * 5 + 3 * 5 + 7;` *(Leftmost multiplication)*

`2 * 5 is 10`

Step 2. `y = 10 * 5 + 3 * 5 + 7;` *(Leftmost multiplication)*

`10 * 5 is 50`

Step 3. `y = 50 + 3 * 5 + 7;` *(Multiplication before addition)*

`3 * 5 is 15`

Step 4. `y = 50 + 15 + 7;` *(Leftmost addition)*

`50 + 15 is 65`

Step 5. `y = 65 + 7;` *(Last addition)*

`65 + 7 is 72`

Step 6. `y = 72` *(Last operation—place* `72` *in* `y`*)*

Fig. 2.11 | Order in which a second-degree polynomial is evaluated.

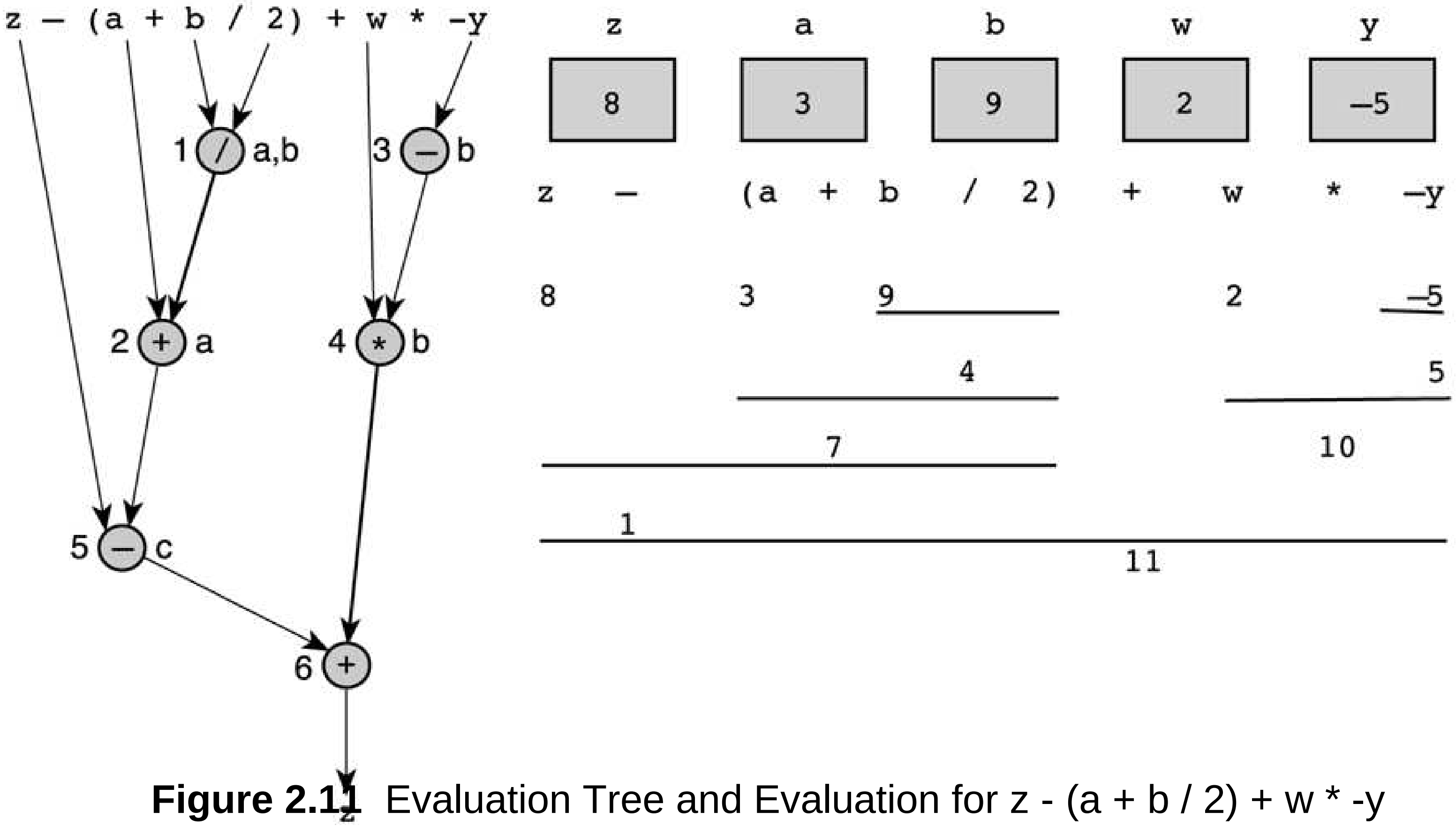


Figure 2.11 Evaluation Tree and Evaluation for z - (a + b / 2) + w * -y

Good Programming Practice 2.12

Using redundant parentheses in complex arithmetic expressions can make the expressions clearer.

2.6 Decision Making: Equality and Relational Operators

- Executable statements
  - Perform actions (calculations, input/output of data)
  - Perform decisions
    - May want to print `"pass"` or `"fail"` given the value of a test grade
- `if` control statement
  - Simple version in this section, more detail later
  - If a condition is `true`, then the body of the `if` statement executed
    - `0` is `false`, non-zero is `true`
  - Control always resumes after the `if` structure
- Keywords
  - Special words reserved for C
  - Cannot be used as identifiers or variable names

Standard algebraic equality operator or relational operator	C equality or relational operator	Example of C condition	Meaning of C condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Fig. 2.12 | Equality and relational operators.

Common Programming Error 2.16

A syntax error will occur if the two symbols in any of the operators ==, != , >= and <= are separated by spaces.

Common Programming Error 2.17

A syntax error will occur if the two symbols in any of the operators !=, >= and <= are reversed as in =!, => and =<, respectively.

Common Programming Error 2.18

Confusing the equality operator == with the assignment operator =.

Common Programming Error 2.19

Placing a semicolon immediately to the right of the right parenthesis after the condition in an `if` statement.

Good Programming Practice 2.13

Indent the statement(s) in the body of an `if` statement.

Outline

fig02_13.c

(1 of 3)

```c
/* Fig. 2.13: fig02_13.c
   Using if statements, relational
   operators, and equality operators */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int num1; /* first number to be read from user  */
   int num2; /* second number to be read from user */

   printf( "Enter two integers, and I will tell you\n" );
   printf( "the relationships they satisfy: " );

   scanf( "%d%d", &num1, &num2 ); /* read two integers */

   if ( num1 == num2 ) {
      printf( "%d is equal to %d\n", num1, num2 );
   } /* end if */

   if ( num1 != num2 ) {
      printf( "%d is not equal to %d\n", num1, num2 );
   } /* end if */

   if ( num1 < num2 ) {
      printf( "%d is less than %d\n", num1, num2 );
   } /* end if */

```

Checks if **num1** is equal to **num2**

Checks if **num1** is not equal to **num2**

Checks if **num1** is less than **num2**

```
   if ( num1 > num2 ) {
      printf( "%d is greater than %d\n", num1, num2 );
   } /* end if */

   if ( num1 <= num2 ) {
      printf( "%d is less than or equal to %d\n", num1, num2 );
   } /* end if */

   if ( num1 >= num2 ) {
      printf( "%d is greater than or equal to %d\n", num1, num2 );
   } /* end if */

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

} /* end function main */
} /* end function main */
```

Checks if **num1** is greater than **num2**

Checks if **num1** is less than or equal to **num2**

Checks if **num1** is greater than equal to **num2**

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

(continued on next slide…)

Outline

fig02_13.c

(2 of 3)

(continued from previous slide…)

```
Enter two integers, and I will tell you
the relationships they satisfy:
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy:
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

`fig02_13.c`

(3 of 3)

Good Programming Practice 2.16

A lengthy statement may be spread over several lines. If a statement must be split across lines, choose breaking points that make sense (such as after a comma in a comma-separated list). If a statement is split across two or more lines, indent all subsequent lines.

Good Programming Practice 2.17

Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are applied in the proper order. If you are uncertain about the order of evaluation in a complex expression, use parentheses to group expressions or break the statement into several simpler statements. Be sure to observe that some of C’s operators such as the assignment operator (=) associate from right to left rather than from left to right.

◀ ▶

Operators	Associativity
( )	left to right
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
=	right to left

Fig. 2.14 | Precedence and associativity of the operators discussed so far.

Figure 3.10 Structure Chart for Drawing a Stick Figure

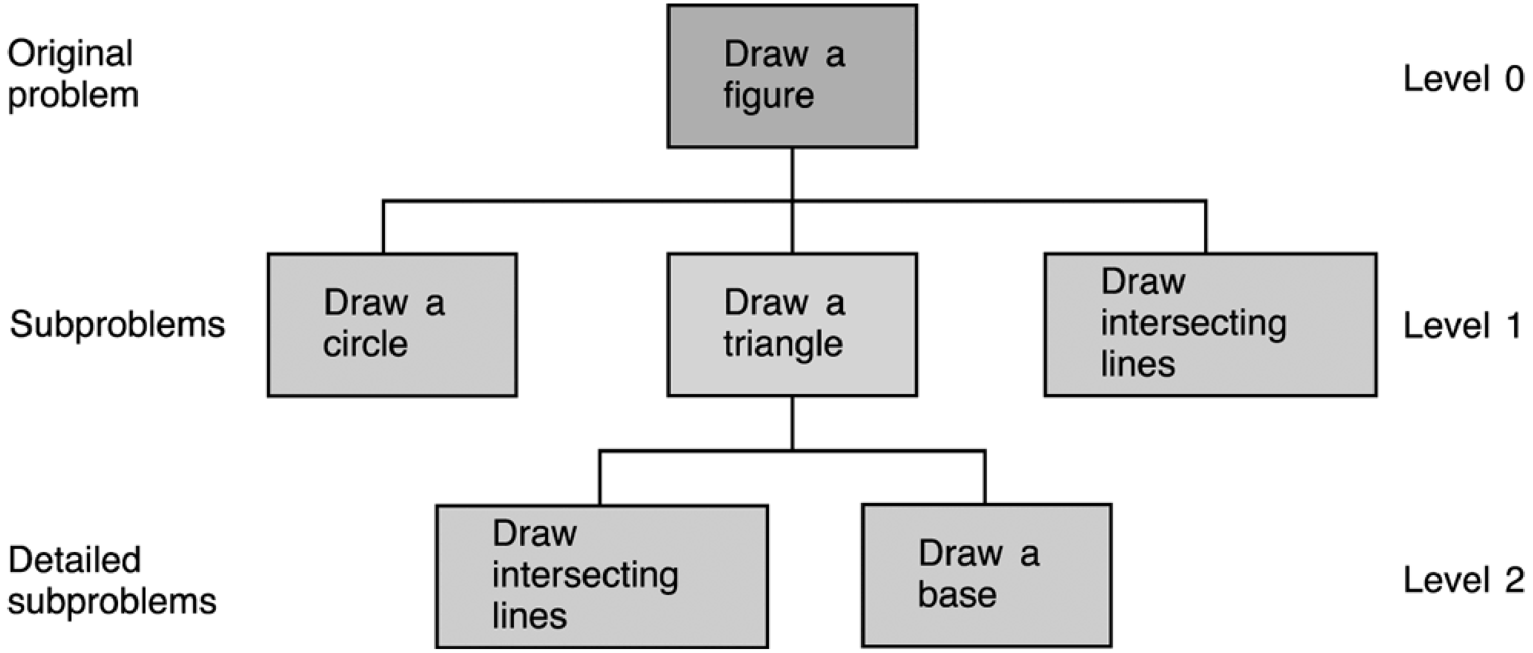
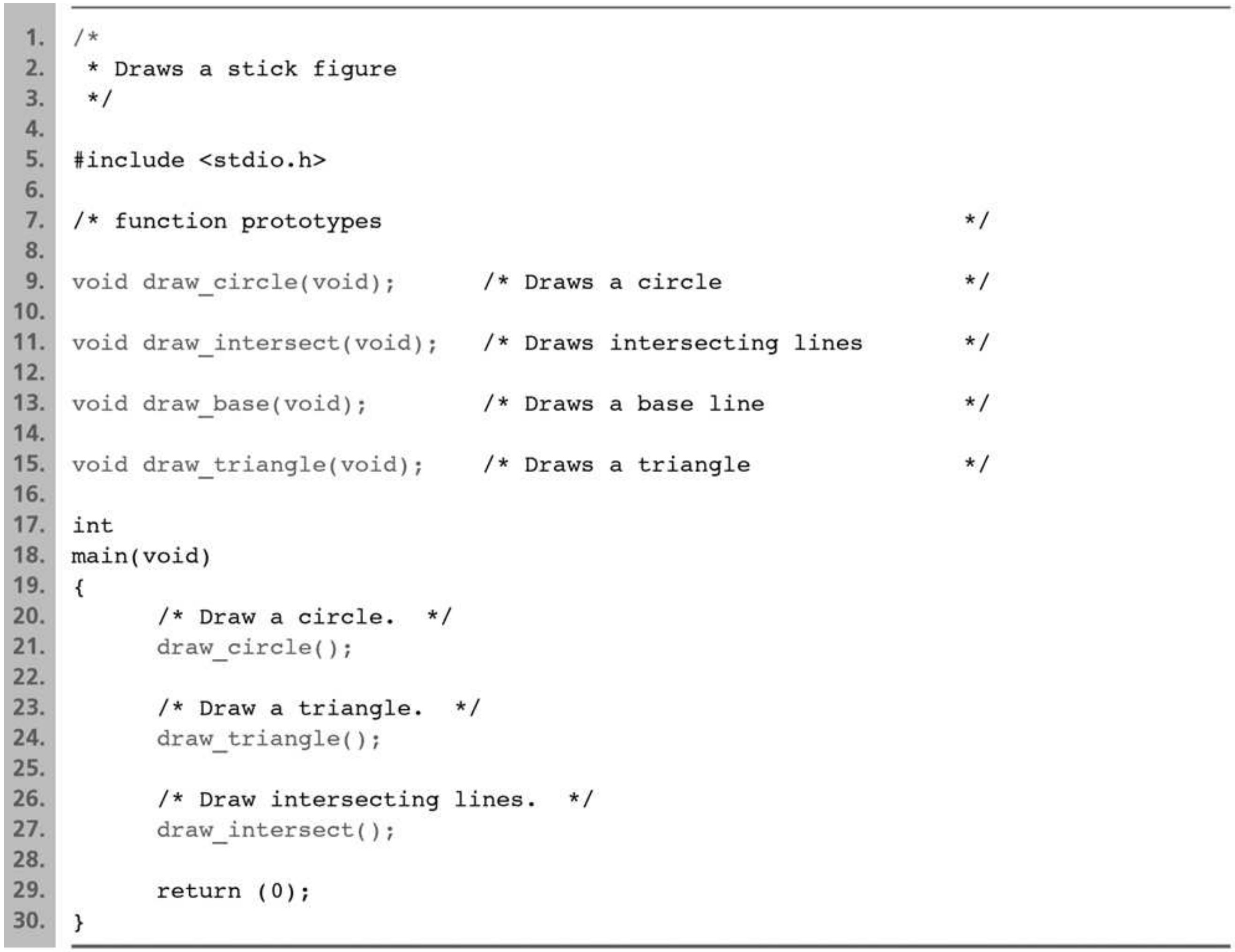
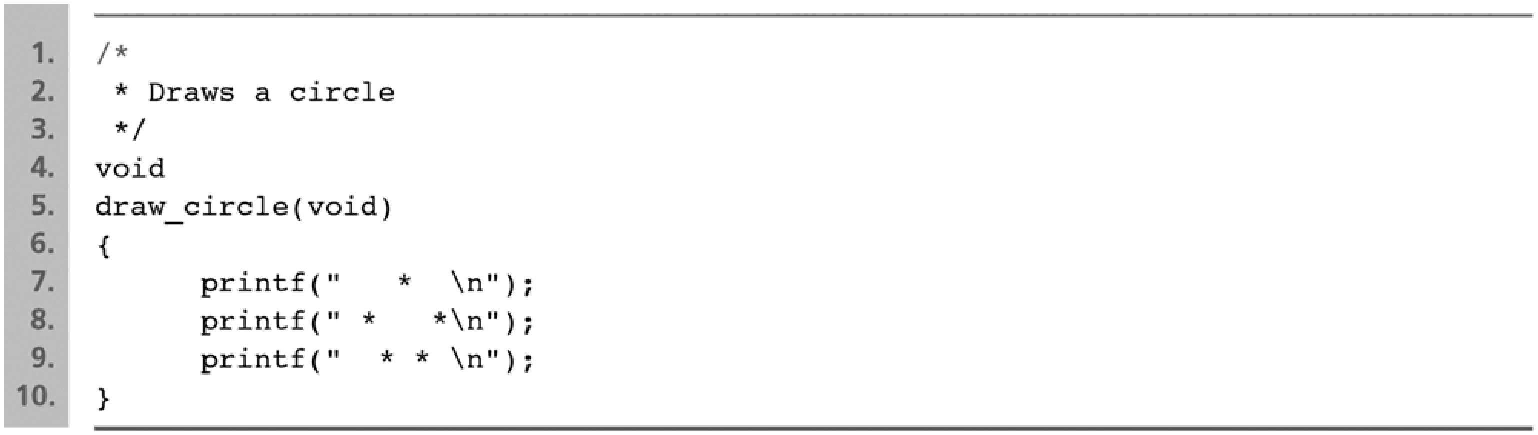


Figure 3.11 Function Prototypes and Main Function for Stick Figure



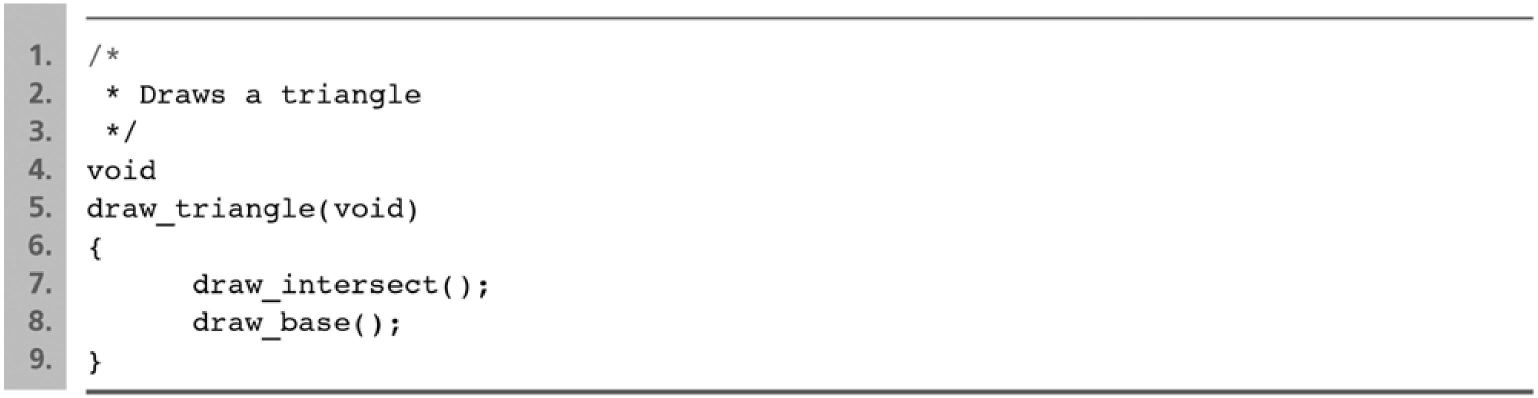
```c
/*
 * Draws a stick figure
 */

#include <stdio.h>

/* function prototypes                                         */

void draw_circle(void);      /* Draws a circle                 */

void draw_intersect(void);   /* Draws intersecting lines       */

void draw_base(void);        /* Draws a base line              */

void draw_triangle(void);    /* Draws a triangle               */

int
main(void)
{
      /* Draw a circle.  */
      draw_circle();

      /* Draw a triangle.  */
      draw_triangle();

      /* Draw intersecting lines.  */
      draw_intersect();

      return (0);
}
```



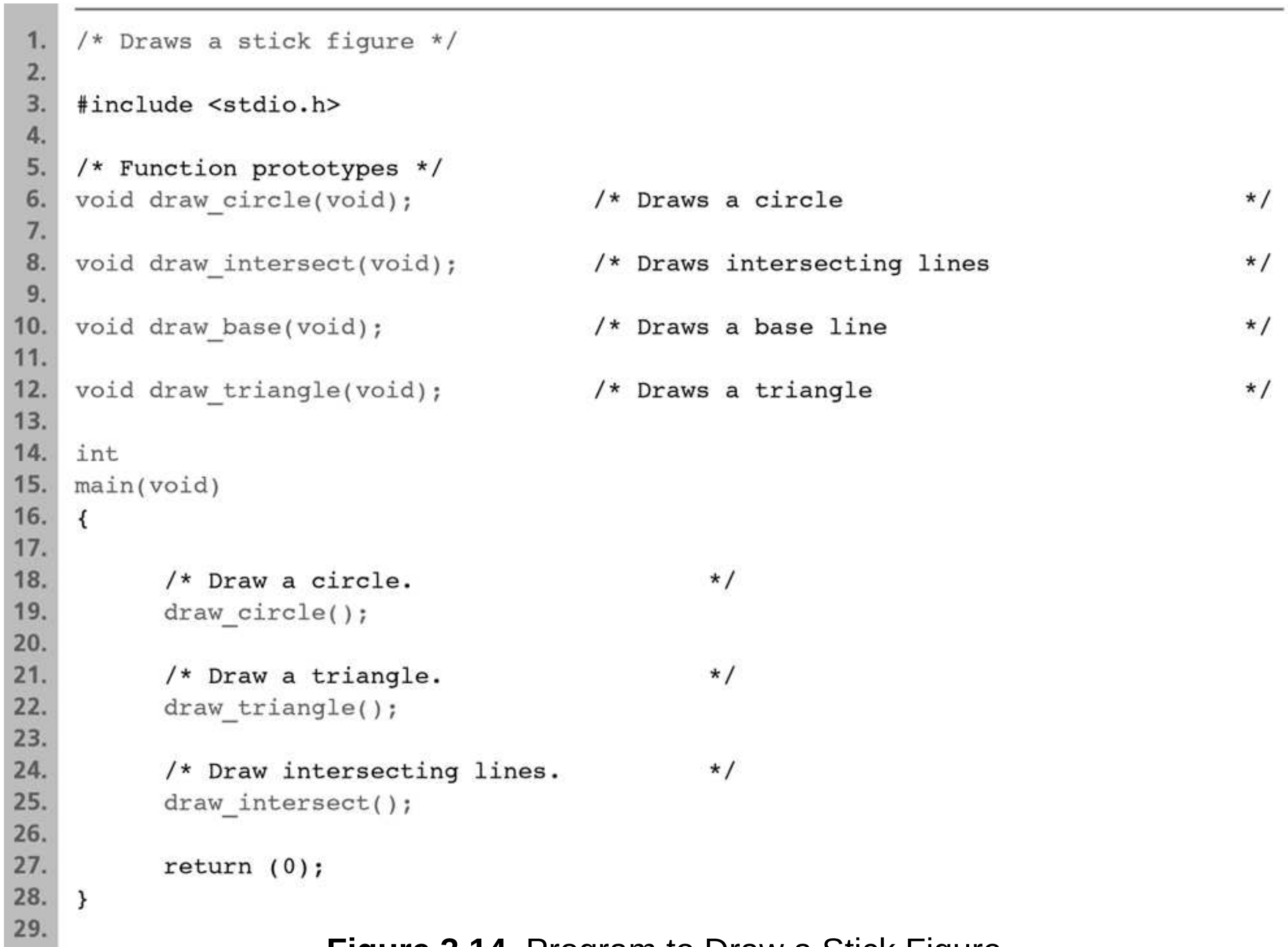
```c
/*
 * Draws a circle
 */
void
draw_circle(void)
{
      printf("   *  \n");
      printf(" *   *\n");
      printf("  * * \n");
}
```

Figure 3.12 Function draw_circle

Figure 3.13 Function draw_triangle



```
/*
 * Draws a triangle
 */
void
draw_triangle(void)
{
      draw_intersect();
      draw_base();
}
```

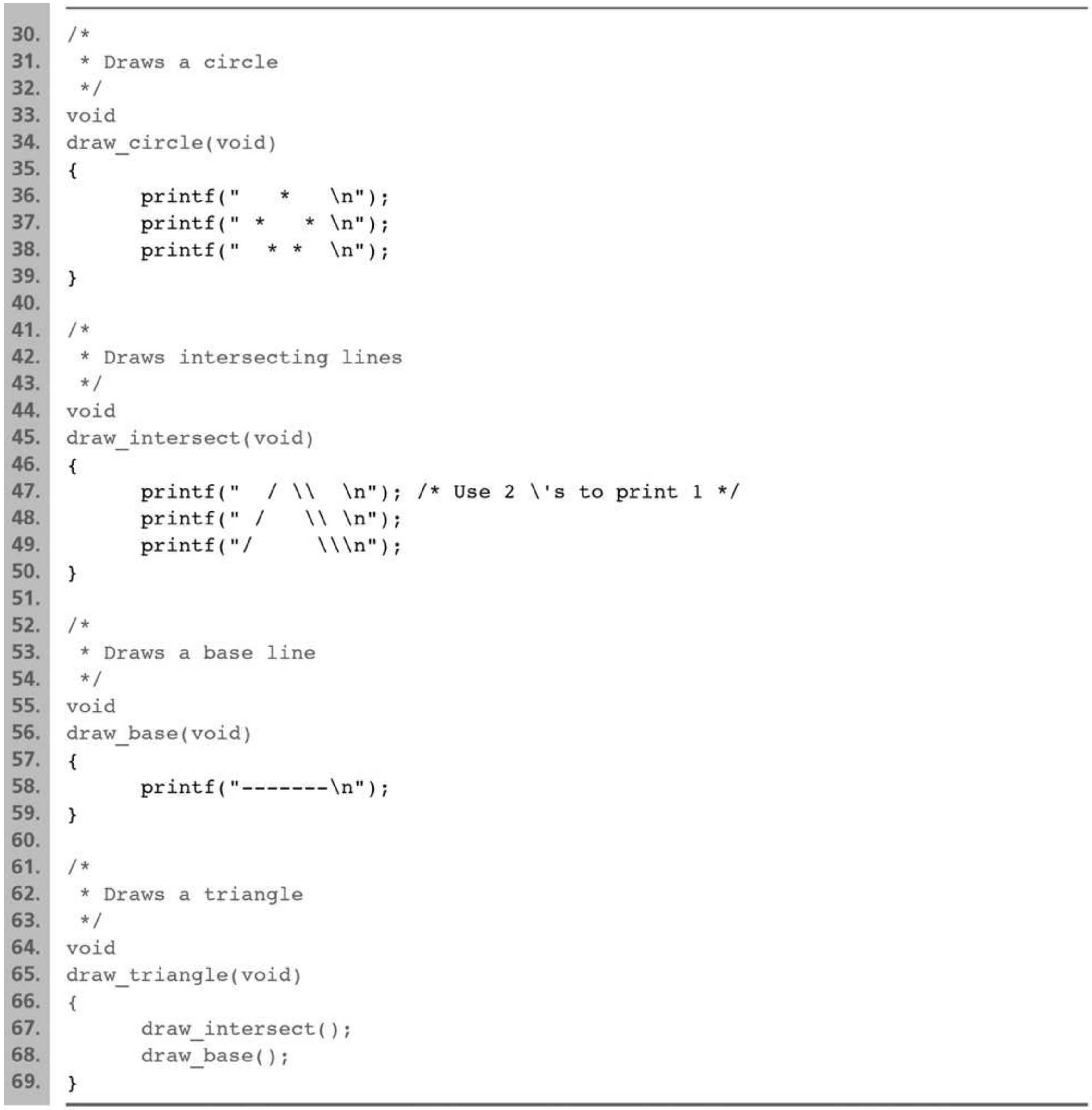


```
/* Draws a stick figure */

#include <stdio.h>

/* Function prototypes */
void draw_circle(void);          /* Draws a circle                   */

void draw_intersect(void);       /* Draws intersecting lines         */

void draw_base(void);            /* Draws a base line                */

void draw_triangle(void);        /* Draws a triangle                 */

int
main(void)
{

      /* Draw a circle.                    */
      draw_circle();

      /* Draw a triangle.                  */
      draw_triangle();

      /* Draw intersecting lines.          */
      draw_intersect();

      return (0);
}

```

Figure 3.14 Program to Draw a Stick Figure

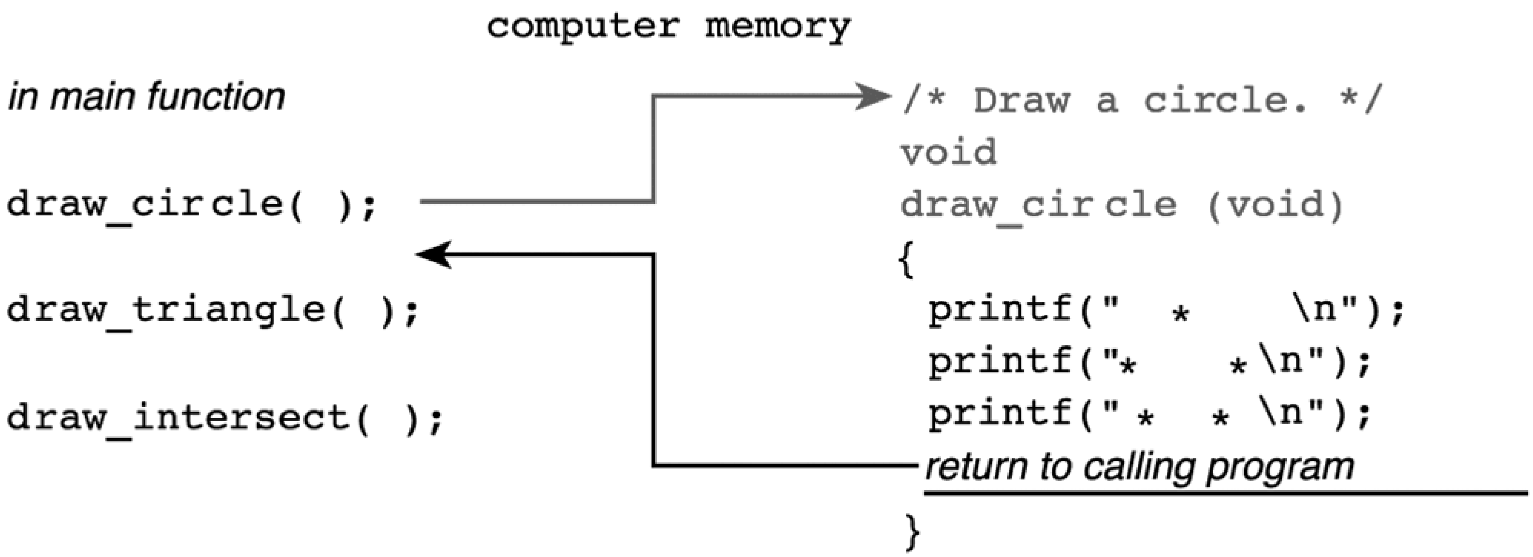
(continued)

Figure 3.14 Program to Draw a Stick Figure (cont'd)



```
/*
 * Draws a circle
 */
void
draw_circle(void)
{
      printf("   *   \n");
      printf(" *   * \n");
      printf("  * *  \n");
}

/*
 * Draws intersecting lines
 */
void
draw_intersect(void)
{
      printf("  / \\  \n"); /* Use 2 \'s to print 1 */
      printf(" /   \\ \n");
      printf("/     \\\n");
}

/*
 * Draws a base line
 */
void
draw_base(void)
{
      printf("-------\n");
}

/*
 * Draws a triangle
 */
void
draw_triangle(void)
{
      draw_intersect();
      draw_base();
}
```

Figure 3.15 Flow of Control Between the main Function and a Function Subprogram



computer memory

in main function

```
draw_circle( );

draw_triangle( );

draw_intersect( );
```

```
/* Draw a circle. */
void
draw_circle (void)
{
 printf("  *    \n");
 printf("*    *\n");
 printf(" *  * \n");
```

return to calling program

```
}
```