A First Book of ANSI C

Fourth Edition

Chapter 3

Processing and Interactive Input

Objectives

- Assignment
- Mathematical Library Functions
- Interactive Input
- Formatted Output

Objectives (continued)

- Symbolic Constants
- Case Study: Interactive Input
- Common Programming and Compiler Errors

Assignment

- The general syntax for an assignment statement is

```
variable = operand;
```

  - The operand to the right of the assignment operator (=) can be a constant, a variable, or an expression
- The equal sign in C does not have the same meaning as an equal sign in algebra
  - `length=25;` is read “length is assigned the value 25”
- Subsequent assignment statements can be used to change the value assigned to a variable

```
length = 3.7;
length = 6.28;
```

Assignment (continued)

- The operand to the right of the equal sign in an assignment statement can be a variable or any valid C expression

```
sum = 3 + 7;
product = .05 * 14.6;
```

  - The value of the expression to the right of = is computed first and then the calculated value is stored in the variable to the left of =
- Variables used in the expression to the right of the = must be initialized if the result is to make sense
- `amount + 1892 = 1000 + 10 * 5` is invalid!

Assignment (continued)

Program 3.1

```
1   #include <stdio.h>
2   int main()
3   {
4     float length, width, area;
5
6     length = 27.2;
7     width = 13.8;
8     area = length * width;
9     printf("The length of the rectangle is %f", length);
10    printf("\nThe width of the rectangle is %f", width);
11    printf("\nThe area of the rectangle is %f", area);
12
13    return 0;
14  }
```

If `width` was not initialized, the computer uses the value that happens to occupy that memory space previously (compiler would probably issue a warning)

Assignment (continued)

- = has the lowest precedence of all the binary and unary arithmetic operators introduced in Section 2.4
- Multiple assignments are possible in the same statement

```
a = b = c = 25;
```

  - All = operators have the same precedence
  - Operator has right-to-left associativity

```
c = 25;
b = c;
a = b;
```

Implicit Type Conversions

- Data type conversions take place across assignment operators

```
double result;
result = 4; //integer 4 is converted to 4.0
```

- The automatic conversion across an assignment operator is called an **implicit type conversion**

```
int answer;
answer = 2.764; //2.764 is converted to 2
```

  - Here the implicit conversion is from a higher precision to a lower precision data type; the compiler will issue a warning

Explicit Type Conversions (Casts)

- The operator used to force the conversion of a value to another type is the **cast** operator

  (dataType) expression

  - where *dataType* is the desired data type of the expression following the cast
- Example:
  - If sum is declared as `double sum;`, `(int) sum` is the integer value determined by truncating `sum`'s fractional part

Assignment Variations

Program 3.2

`sum = sum + 10` is not an equation—it is an expression that is evaluated in two major steps

```
1   #include <stdio.h>
2   int main()
3   {
4     int sum;
5
6     sum = 25;
7     printf("\nThe number stored in sum is %d.",sum);
8     sum = sum + 10;
9     printf("\nThe number now stored in sum is %d.\n",sum);
10
11    return 0;
12  }
```

Assignment Variations (continued)

Figure 3.1 The integer 25 is stored in `sum`

Assignment Variations (continued)

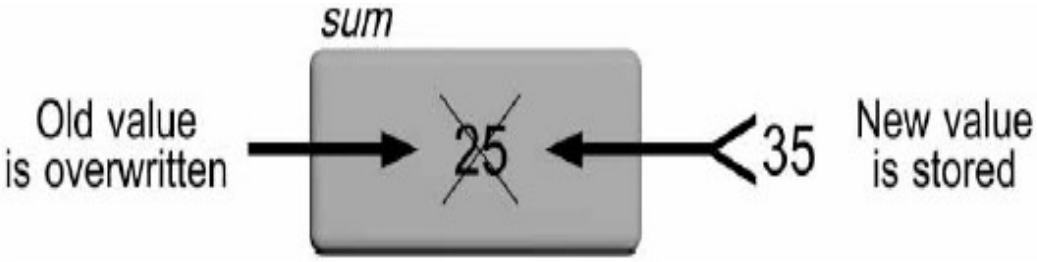

Figure 3.2 `sum = sum + 10;` causes a new value to be stored in `sum`

Assignment Variations (continued)

- Assignment expressions like `sum = sum + 25` can be written using the following operators:
  - `+= -= *= /= %=`
- `sum = sum + 10` can be written as `sum += 10`
- `price *= rate` is equivalent to `price = price * rate`
- `price *= rate + 1` is equivalent to `price = price * (rate + 1)`

Accumulating

- The first statement initializes sum to 0
  - This removes any previously stored value in sum that would invalidate the final total
  - A previously stored number, if it has not been initialized to a specific and known value, is frequently called a **garbage value**

Accumulating (continued)

Table 3.1 Statements and Resulting Value when Adding 96, 70, 85, and 60

Statement	Value in sum
`sum = 0;`	0
`sum = sum + 96;`	96
`sum = sum + 70;`	166
`sum = sum + 85;`	251
`sum = sum + 60;`	311

Accumulating (continued)

Program 3.3

```
 1  #include <stdio.h>
 2  int main()
 3  {
 4    int sum;
 5
 6    sum = 0;
 7    printf("\nThe value of sum is initially set to %d.", sum);
 8    sum = sum + 96;
 9    printf("\n sum is now %d.", sum);
10    sum = sum + 70;
11    printf("\n sum is now %d.", sum);
12    sum = sum + 85;
13    printf("\n sum is now %d.", sum);
14    sum = sum + 60;
15    printf("\n The final sum is %d.\n", sum);
16
17    return 0;
18  }
```

Counting

- A **counting statement** is very similar to the accumulating statement

  ```
  variable = variable + fixedNumber;
  ```

- Examples: `i = i + 1;` and `m = m + 2;`
- **Increment operator** (++): `variable = variable + 1` can be replaced by `variable++` or `++variable`

Counting (continued)

Table 3.2 Examples of the Increment Operator

Expression	Alternative
`i = i + 1`	`i++ and ++i`
`n = n + 1`	`n++ and ++n`
`count = count + 1`	`count++ and ++count`

Counting (continued)

Program 3.4

```
 1  #include <stdio.h>
 2  int main()
 3  {
 4    int count;
 5
 6    count = 0;
 7    printf("\nThe initial value of count is %d.", count);
 8    count++;
 9    printf("\n count is now %d.", count);
10    count++;
11    printf("\n count is now %d.", count);
12    count++;
13    printf("\n count is now %d.", count);
14    count++;
15    printf("\n count is now %d.\n", count);
16
17    return 0;
18  }
```

Counting (continued)

- When the ++ operator appears before a variable, it is called a **prefix increment operator**; when it appears after a variable, it is called **postfix increment operator**
  - `k = ++n;` is equivalent to
    - `n = n + 1; // increment n first`
    - `k = n; // assign n's value to k`
  - `k = n++;` is equivalent to
    - `k = n; // assign n's value to k`
    - `n = n + 1; // and then increment n`

Counting (continued)

- **Prefix decrement operator:** the expression `k = --n` first decrements the value of `n` by 1 before assigning the value of `n` to `k`
- **Postfix decrement operator:** the expression `k = n--` first assigns the current value of `n` to `n` and then reduces the value of `n` by 1

Counting (continued)

Table 3.3 Examples of the Decrement Operator

Expression	Alternative
`i = i - 1`	`i-- and --i`
`n = n - 1`	`n-- and --n`
`count = count - 1`	`count-- and --count`

Mathematical Library Functions

Table 3.4 Commonly Used Mathematical Functions (all functions require the math.h header file)

Function	Description	Example	Returned Value	Comments
sqrt(x)	Square root of x	sqrt(16.00)	4.000000	an integer value of x results in a compiler error
pow(x,y)	x raised to the y power (x^y)	pow(2, 3) pow(81, .5)	8.000000 9.000000	integer values of x and y are permitted
exp(x)	e raised to the x power (e^x)	exp(-3.2)	0.040762	an integer value of x results in a compiler error
log(x)	Natural log of x (base e)	log(18.697)	2.928363	an integer value of x results in a compiler error
log10(x)	Common log of x (base 10)	log10(18.697)	1.271772	an integer value of x results in a compiler error
fabs(x)	Absolute value of x	fabs(-3.5)	3.5000000	an integer value of x results in a compiler error
abs(x)	Absolute value of x	abs(-2)	2	a floating-point value of x returns a Value of 0

Mathematical Library Functions (continued)

- The argument to `sqrt` must be floating-point value; passing an integer value results in a compiler error
  - Return value is double-precision
- Must include `#include <math.h>`

Mathematical Library Functions (continued)

Table 3.5 Examples Using `sqrt()`

Expression	Returned Value
`sqrt(4.0)`	`2.000000`
`sqrt(17.0)`	`4.123106`
`sqrt(25.0)`	`5.000000`
`sqrt(1043.29)`	`32.300000`
`sqrt(6.4516)`	`2.540000`

Mathematical Library Functions (continued)

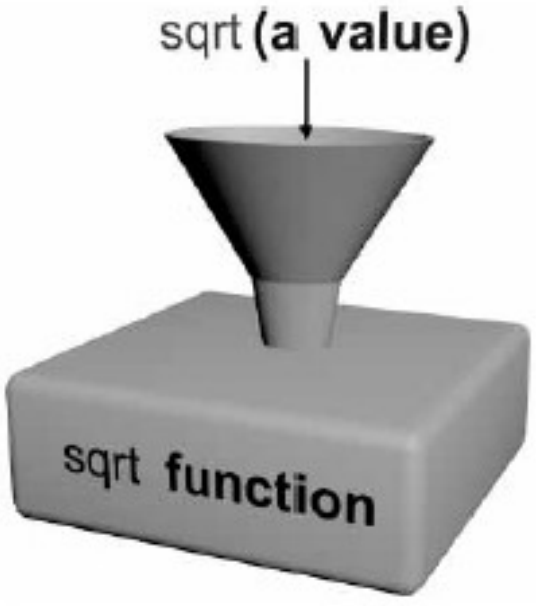

Figure 3.3 Passing data to the `sqrt()` function

Mathematical Library Functions (continued)

Program 3.5

```
 1  #include <stdio.h>
 2  #include <math.h>
 3  int main()
 4  {
 5    double result;
 6
 7    printf("The square root of 6.456 is %f\n", sqrt(6.456));
 8    printf("7.6 raised to the 3rd power is %f\n", pow(7.6, 3));
 9
10    result = fabs(-8.24);
11    printf("The absolute value of -8.24 is %f\n", result);
12
13    return 0;
14  }
```

Argument need not be a single constant

Mathematical Library Functions (continued)

- The step-by-step evaluation of the expression

  ```
  3.0 * sqrt(5 * 33 - 13.91) / 5
  ```

  is (see next slide)

Mathematical Library Functions (continued)

Step	Result
1. Perform multiplication in argument	`3.0 * sqrt(165 - 13.91) / 5`
2. Complete argument calculation	`3.0 * sqrt(151.090000) / 5`
3. Return a function value	`3.0 * 12.2918672 / 5`
4. Perform the multiplication	`36.8756017 / 5`
5. Perform the division	`7.3751203`

Mathematical Library Functions (continued)

- Determine the time it takes a ball to hit the ground after it has been dropped from an 800-foot tower
  - *time = sqrt(2 * distance/g),* where *g* = 32.2 ft/sec2

Mathematical Library Functions (continued)

Program 3.6

```
 1  #include <stdio.h>  /* this line may be placed second instead of first */
 2  #include <math.h>   /* this line may be placed first instead of second */
 3  int main()
 4  {
 5    int height;
 6    double time;
 7
 8    height = 800.0;
 9    time = sqrt(2.0 * height / 32.2);
10    printf("It will take %f seconds", time);
11    printf(" to fall %d feet.\n", height);
12
13    return 0;
14  }
```

Interactive Input

Program 3.8

```
1  #include <stdio.h>
2  int main()
3  {
4    printf("%f times %f is %f\n", 300.0, .05, 300.0*.05);
5
6    return 0;
7  }
```

Interactive Input (continued)

- This program must be rewritten to multiply different numbers
- `scanf()` is used to enter data into a program while it is executing; the value is stored in a variable
  - It requires a control string as the first argument inside the function name parentheses

Interactive Input (continued)

- The control string passed to `scanf()` typically consists of conversion control sequences only
- `scanf()` requires that a list of variable addresses follow the control string
  - `scanf("%d", &num1);`

Interactive Input (continued)

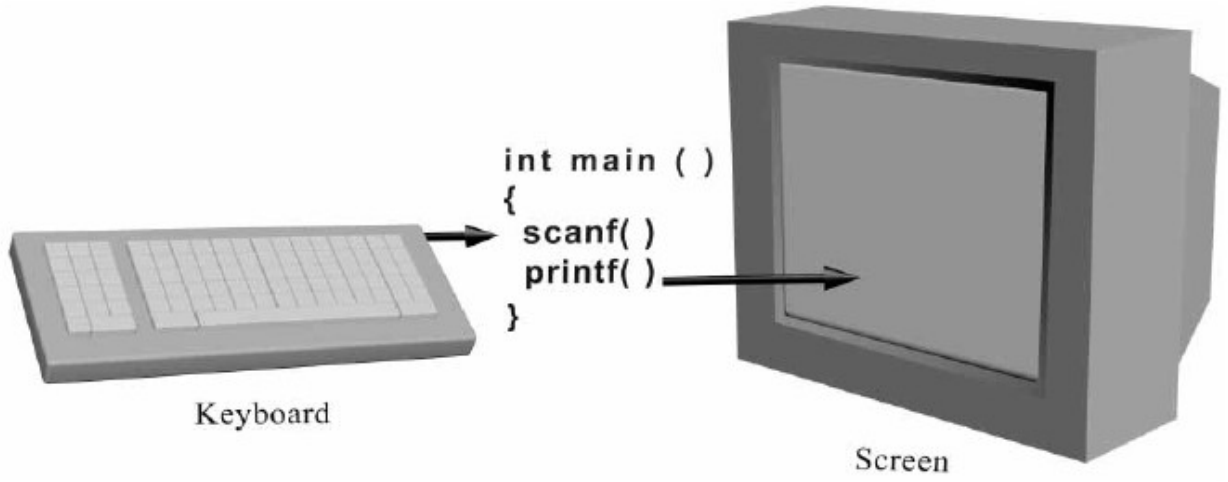

Figure 3.5 `scanf()` used to enter data; `printf()` used to display data

Interactive Input (continued)

Program 3.9

```
 1  #include <stdio.h>
 2  int main()
 3  {
 4    float num1, num2, product;
 5
 6    printf("Please type in a number: ");
 7    scanf("%f",&num1);
 8    printf("Please type in another number: ");
 9    scanf("%f",&num2);
10    product = num1 * num2;
11    printf("%f times %f is %f\n", num1, num2, product);
12
13    return 0;
14  }
```

This statement produces a **prompt**

Address operator (&)

Interactive Input (continued)

- `scanf()` can be used to enter many values

  `scanf("%f %f",&num1,&num2); //"%f%f"` is the same
- A space can affect what the value being entered is when `scanf()` is expecting a character data type
  - `scanf("%c%c%c",&ch1,&ch2,&ch3);` stores the next three characters typed in the variables `ch1`, `ch2`, and `ch3`; if you type `x y z`, then `x` is stored in `ch1`, a blank is stored in `ch2`, and `y` is stored in `ch3`
  - `scanf("%c %c %c",&ch1,&ch2,&ch3);` causes `scanf()` to look for three characters, each character separated by exactly one space

Interactive Input (continued)

- In printing a double-precision number using `printf()`, the conversion control sequence for a single-precision variable, `%f`, can be used
- When using `scanf()`, if a double-precision number is to be entered, you must use the `%lf` conversion control sequence
- `scanf()` does not test the data type of the values being entered
- In `scanf("%d %f", &num1, &num2)`, if user enters `22.87`, `22` is stored in `num1` and `.87` in `num2`

Caution: The Phantom Newline Character

Program 3.10

```
 1  #include <stdio.h>
 2  int main()
 3  {
 4    char fkey, skey;
 5
 6    printf("Type in a character: ");
 7    scanf("%c", &fkey);
 8    printf("The keystroke just accepted is %d", fkey);
 9    printf("\nType in another character: ");
10    scanf("%c", &skey);
11    printf("The keystroke just accepted is %d\n", skey);
12
13    return 0;
14  }
```

Caution: The Phantom Newline Character (continued)

- The following is a sample run for Program 3.10:

```
Type in a character: m
The keystroke just accepted is 109
Type in another character: The keystroke just
  accepted is 10
```

Caution: The Phantom Newline Character (continued)

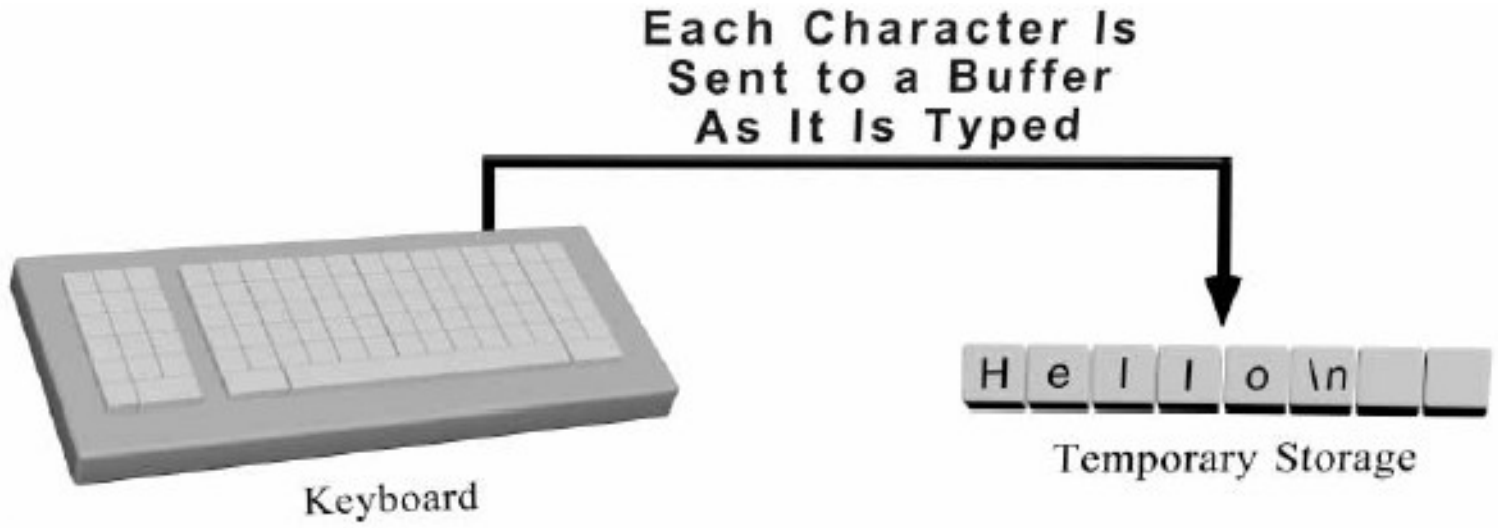

Figure 3.6 Typed keyboard characters are first stored in a buffer

Caution: The Phantom Newline Character (continued)

Program 3.11

```
 1  #include <stdio.h>
 2  int main()
 3  {
 4    char fkey, skey;
 5
 6    printf("Type in a character: ");
 7    scanf("%c%c", &fkey, &skey); /* the enter code goes to skey */
 8    printf("The keystroke just accepted is %d", fkey);
 9    printf("\nType in another character: ");
10    scanf("%c", &skey); /* accept another code */
11    printf("The keystroke just accepted is %d\n", skey);
12
13    return 0;
14  }
```

A First Look at User-Input Validation

Program 3.12

```
 1  #include <stdio.h>
 2  int main()
 3  {
 4    int num1, num2, num3;
 5    double average;
 6
 7    /* get the input data */
 8    printf("Enter three integer numbers: ");
 9    scanf("%d %d %d", &num1, &num2, &num3);
10
11    /* calculate the average*/
12    average = (num1 + num2 + num3) / 3.0;
13
14    /* display the result */
15    printf("\nThe avearge of %d, %d, and %d is %f\n",
16                            num1, num2, num3, average);
17
18
19    return 0;
20  }
```

A First Look at User-Input Validation (continued)

- As written, Program 3.12 is not **robust**
- The problem becomes evident when a user enters a non-integer value

```
Enter three integer numbers: 10 20.68 20
The average of 10, 20, and -858993460 is
  -286331143.333333
```

- Handling invalid data input is called **user-input validation**
  - Validating the entered data either during or immediately after the data have been entered
  - Providing the user with a way of reentering any invalid data

Formatted Output

Program 3.13

```
 1  #include <stdio.h>
 2  int main()
 3  {
 4    printf("\n%d", 6);
 5    printf("\n%d", 18);
 6    printf("\n%d", 124);
 7    printf("\n---");
 8    printf("\n%d\n", 6+18+124);
 9
10    return 0;
11  }
```

```
6
18
124
---
148
```

Output is not aligned

Formatted Output (continued)

Program 3.14

```
 1  #include <stdio.h>
 2  int main()
 3  {
 4    printf("\n%3d", 6);
 5    printf("\n%3d", 18);
 6    printf("\n%3d", 124);
 7    printf("\n---");
 8    printf("\n%3d\n", 6+18+124);
 9
10    return 0;
11  }
```

Field width specifier

```
  6
 18
124
---
148
```

Formatted Output (continued)

Table 3.6 Effect of Field Width Specifiers

Specifier	Number	Display	Comments
`%2d`	3	`␣3`	Number fits in field
`%2d`	43	`43`	Number fits in field
`%2d`	143	`143`	Field width ignored
`%2d`	2.3	Compiler dependent	Floating-point number in an integer field
`%5.2f`	2.366	`␣2.37`	Field of 5 with 2 decimal digits
`%5.2f`	42.3	`42.30`	Number fits in field
`%5.2f`	142.364	`142.36`	Field width ignored but fractional specifier is used
`%5.2f`	142	Compiler dependent	Integer in a floating-point field

Format Modifiers

- **Left justification:** `printf("%-10d",59);` produces the display `59ʌʌʌʌʌʌʌʌ`
- **Explicit sign display:** `printf("%+10d",59);` produces the display `ʌʌʌʌʌʌʌʌ+59`
- Format modifiers may be combined
  - `%-+10d` would cause an integer number to both display its sign and be left-justified in a field width of 10 spaces
    - The order of the format modifiers is not critical

      `%+-10d` is the same

Other Number Bases [Optional]

Program 3.15

```
1  #include <stdio.h>
2  int main() /* a program to illustrate output conversions */
3  {
4    printf("The decimal (base 10) value of 15 is %d.", 15);
5    printf("\nThe octal (base 8) value of 15 is %o.", 15);
6    printf("\nThe hexadecimal (base 16) value of 15 is %x\n.", 15);
7
8    return 0;
9  }
```

```
The decimal (base 10) value of 15 is 15.
The octal (base 8) value of 15 is 17.
The hexadecimal (base 16) value of 15 is f.
```

Other Number Bases (continued)

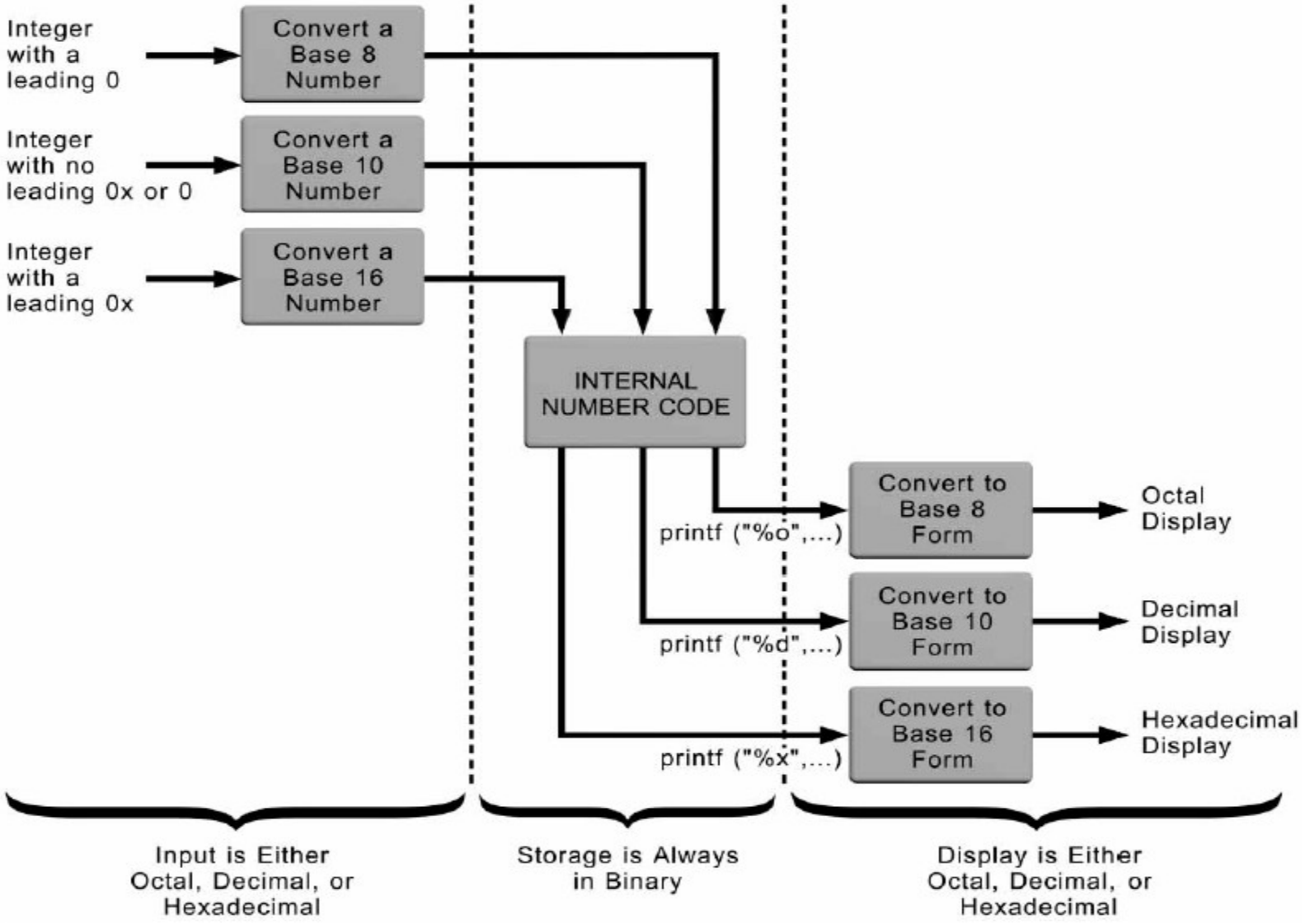

Figure 3.7 Input, storage, and display of integers

Other Number Bases (continued)

Program 3.17

```
1  #include <stdio.h>
2  int main()
3  {
4    printf("The decimal value of the letter %c is %d.", 'a', 'a');
5    printf("\nThe octal value of the letter %c is %o.", 'a', 'a');
6    printf("\nThe hex value of the letter %c is %x.\n", 'a', 'a');
7
8    return 0;
9  }
```

```
The decimal value of the letter a is 97.
The octal value of the letter a is 141.
The hex value of the letter a is 61.
```

Other Number Bases (continued)

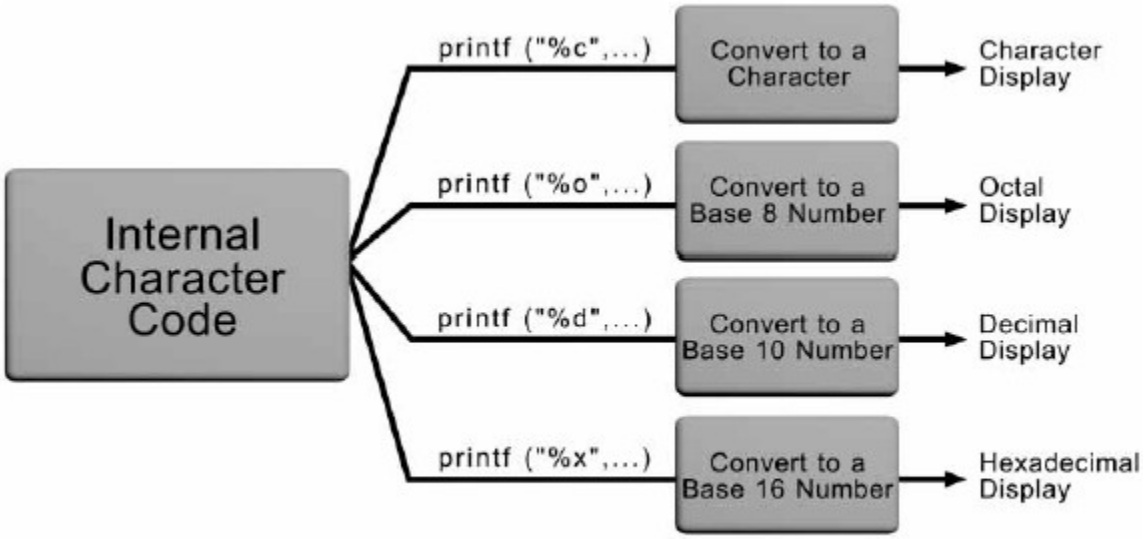

Figure 3.8 Character display options

Symbolic Constants

- **Literal data** refers to any data within a program that explicitly identifies itself
- Literal values that appear many times in the same program are called **magic numbers**
- C allows you to define the value once by equating the number to a **symbolic name**
  - `#define SALESTAX 0.05`
  - `#define PI 3.1416`
  - Also called **symbolic constants** and **named constants**

Symbolic Constants (continued)

Program 3.18

sign is a signal to a C preprocessor

```
1  #include <stdio.h>
2  #define SALESTAX 0.05
3  int main()
4  {
5    float amount, taxes, total;
6
7    printf("\nEnter the amount purchased: ");
8    scanf("%f", &amount);
9    taxes = SALESTAX * amount;
10   total = amount + taxes;
11   printf("The sales tax is $%4.2f",taxes);
12   printf("\nThe total bill is $%5.2f\n",total);
13
14   return 0;
15 }
```

Case Study: Interactive Input

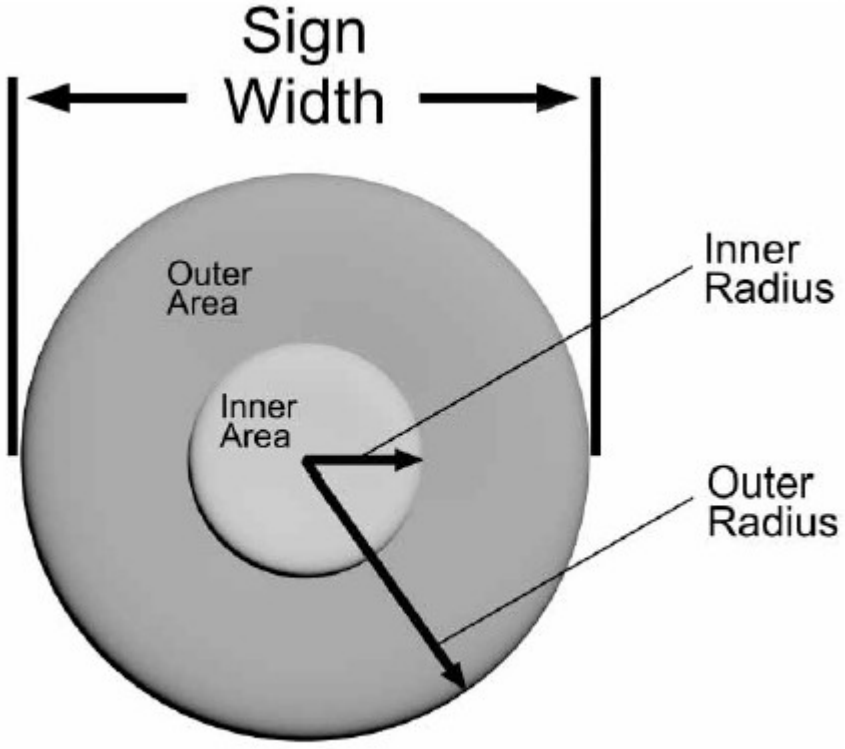

Figure 3.9 The Hit-The-Mark display

Case Study: Interactive Input (continued)

Program 3.19

```
 1  #include <stdio.h>
 2  #include <math.h>
 3  #define SQFTPERQUART 200.0
 4  #define PI 3.1416
 5
 6  int main()
 7  {
 8    float width, outerRadius, innerRadius;
 9    float totalArea, innerArea, outerRimArea;
10    float blue, red;
11
12    /* get the input data */
13    printf("Enter the width of the display (in feet): ");
14    scanf("%f", &width);
15
16    /* determine the two radii */
17    outerRadius = width/2.0;
18    innerRadius = 0.25 * outerRadius;
19
20    /* determine the two areas */
21    totalArea = PI * pow(outerRadius, 2);
22    innerArea = PI * pow(innerRadius, 2);
23    outerRimArea = totalArea - innerArea;
24
25    /* determine the gallons of paint needed */
26    red =  innerArea / SQFTPERQUART;
27    blue = outerRimArea / SQFTPERQUART;
28
29    /* provide the required outputs */
30    printf("\nThe inner area is %5.2f sq. feet", innerArea);
31    printf("\nThe outer rim area is %5.2f sq feet", outerRimArea);
32    printf("\n\nRed paint required is %6.3f quarts", red);
33    printf("\nBlue paint required is %6.3f quarts\n", blue);
34
35    return 0;
36  }
```

Common Programming Errors

- Forgetting to assign initial values to all variables before the variables are used in an expression
- Calling `sqrt()` with an integer argument
- Forgetting to use the address operator, &, in front of variable names in a `scanf()` function call
- Not including the correct control sequences in `scanf()` function calls for the data values that must be entered
- Including a message within the control string passed to `scanf()`

Common Programming Errors (continued)

- Terminating a `#define` command to the preprocessor with a semicolon
- Placing an equal sign in a `#define` command when equating a symbolic constant to a value
- Using the increment and decrement operators with variables that appear more than once in the same expression
- Being unwilling to test a program in depth

Common Compiler Errors

Error	Typical Unix-based compiler error message	Typical Windows-based compiler error message
Attempting to use a mathematical function, such as `pow` without including the `math.h` header file.	`"ERROR: Undefined symbol: .pow` (You can use the `-bloadmap` or `-bnoquiet` options when compiling the program to obtain more information. Additionally, you must use the `-lm` option for correct compilation.)	`"pow identifier not found."`
Forgetting to close the control string passed to `scanf()` with double quotes.	`"(S) String literal must be ended before the end of line."` `"(S) Syntax error: possible missing ')'?"` (The first error message is attempting to tell you that the string has not been closed using a double quote. The second error message is a result of the string not being terminated, which causes an error on the line following the call to `scanf()`.)	`"newline in constant" "syntax error: missing ')' before identifier…"` (The first error message is attempting to tell you that the string has not been closed using a double quote. The second error message is a result of the string not being terminated, which causes an error on the line following the call to `scanf()`.)

Common Compiler Errors (continued)

Error	Typical Unix-based compiler error message	Typical Windows-based compiler error message
Failing to separate all arguments in `scanf()` with commas as, for example, in the call `scanf("%f%f", &count &n);`	`"(S) Operation between types "unsigned char*" and "float" is not allowed."` (Although very cryptic, this message indicates that the compiler cannot recognize the variable in which the function is trying to store a value.)	`" '&': illegal, left operand has type …"` (Although very cryptic, this message indicates that the compiler cannot recognize the variable in which the function is trying to store a value.)
Placing the parentheses in the wrong location when using the cast operator, as, for example, in the expression `(int count)`	`"(E) Identifier not allowed in cast or sizeof declarations." "(S) Syntax error."`	`"syntax error: missing ')' before count" "syntax error:')'"`
Applying the increment or decrement operators to an expression. For example, the expression `(count + n)++`	`"Operand must be a modifiable lvalue."` (This error message is indicates that the expression to the left of the ++ operator can not be modified.)	`"++ needs l-value."` (This error message indicates that the expression to the left of the ++ operator can not be modified.)

Summary

- Arithmetic calculations can be performed using assignment statements or mathematical functions
- The assignment symbol, =, is an operator
- C provides the +=, -=, *= and /= assignment operators
- The increment operator, ++, adds 1 to a variable
- The decrement operator, --, subtracts 1 from a variable
- C provides library functions for calculating square root, logarithmic, and other mathematical computations

Summary (continued)

- Mathematical functions may be included within larger expressions
- `scanf()` is a standard library function used for data input
- When a `scanf()` function is encountered, the program temporarily suspends further statement execution until sufficient data has been entered for the number of variable addresses contained in the `scanf()` function call

Summary (continued)

- It is good programming practice to display a message, prior to a `scanf()` function call, that alerts the user as to the type and number of data items to be entered
- Field width specifiers can be included with conversion control sequences to explicitly specify the format of displayed fields
- Each compiled C program is automatically passed through a preprocessor
- Expressions can be made equivalent to a single identifier using the preprocessor `#define` command