
Chapter Two: Fundamental Data Types

Chapter Goals

- define and initialize variables and constants
- understand the properties and limitations of integer and floating-point numbers
- write arithmetic expressions and assignment statements
- appreciate the importance of comments and good code layout
- create programs that read and process input, and display the results

Variables

- A variable
 - is used to store information: the contents of the variable:
 - can contain one piece of information at a time.
 - has an identifier: the name of the variable
- - The programmer picks a good name
 - A good name describes the contents of the variable or what the variable will be used for

Parking garages store cars.



Variables

Each parking space is identified—
like a variable's identifier



A each parking space in a garage “contains” a car
— like a variable's current contents.

Variables

and
each space can contain only *one* car



and
only cars, not buses or trucks

Variable Definitions

- When creating variables, the programmer specifies the type of information to be stored.
 - (more on types later)
- Unlike a parking space, a variable is often given an initial value.
 - *Initialization* is putting a value into a variable when the variable is created.
 - Initialization is not required.

Variable Definitions

The following statement defines a variable.

```
int cans_per_pack = 6;
```

cans_per_pack is the variable's name.

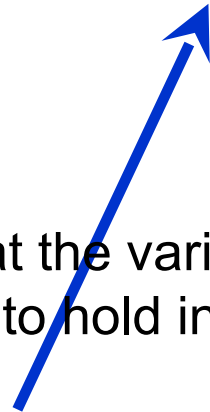


int



indicates that the variable **cans_per_pack** will be used to hold integers.

= 6



indicates that the variable **cans_per_pack** will initially contain the value 6.

Variable Definitions

SYNTAX 2.1 Variable Definition

Types introduced in this chapter are the number types `int` and `double` and the `string` type

Use a descriptive variable name.



Must obey the rules for valid names

A variable definition ends with a semicolon.

Supplying an initial value is optional, but it is usually a good idea.

```
int cans_per_pack = 6;
```



Variable Definitions

Variable Name	Comment
<code>int cans = 6;</code>	Defines an integer variable and initializes it with 6.
<code>int total = cans + bottles;</code>	The initial value need not be a constant. (Of course, <code>cans</code> and <code>bottles</code> must have been previously defined.)
 <code>int bottles = "10";</code>	Error: You cannot initialize a number with a string.
<code>int bottles;</code>	Defines an integer variable without initializing it. This can be a cause for errors—see Common Error 2.2 on page 37.
<code>int cans, bottles;</code>	Defines two integer variables in a single statement. In this book, we will define each variable in a separate statement.
 <code>bottles = 1;</code>	Caution: The type is missing. This statement is not a definition but an assignment of a new value to an existing variable—see Section 2.1.4 on page 34.

A number written by a programmer is called a *number literal*.

There are rules for writing literal values:

Number Types

Number	Type	Comment
6	int	An integer has no fractional part.
-6	int	Integers can be negative.
0	int	Zero is an integer.
0.5	double	A number with a fractional part has type double.
1.0	double	An integer with a fractional part .0 has type double.
1E6	double	A number in exponential notation: 1×10^6 or 1000000. Numbers in exponential notation always have type double.
2.96E-2	double	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
 100,000		Error: Do not use a comma as a decimal separator.
 3 1/2		Error: Do not use fractions; use decimal notation: 3.5






Variable Names

- When you define a variable, you should pick a name that explains its purpose.
- For example, it is better to use a descriptive name, such as `can_volume`, than a terse name, such as `cv`.

In C, there are a few simple rules for variable names

1. Variable names must start with a letter or the underscore (`_`) character, and the remaining characters must be letters, numbers, or underscores.
2. You cannot use other symbols such as `$` or `%`. Spaces are not permitted inside names; you can use an underscore instead, as in `can_volume`.
3. Variable names are *case-sensitive*, that is, `can_volume` and `can_Volume` are different names.
For that reason, it is a good idea to use only lowercase letters in variable names.
4. You cannot use *reserved words* such as `double` or `return` as names; these words are reserved exclusively for their special C meanings.

Variable Names

Variable Name	Comment
can_volume1	Variable names consist of letters, numbers, and the underscore character.
x	In mathematics, you use short variable names such as x or y . This is legal in C++, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 38).
 Can_volume	Caution: Variable names are case-sensitive. This variable name is different from can_volume.
 6pack	Error: Variable names cannot start with a number.
 can volume	Error: Variable names cannot contain spaces.
 double	Error: You cannot use a reserved word as a variable name.
 1ltr/fl.oz	Error: You cannot use symbols such as / or .

The Assignment Statement

- The contents in variables can “vary” over time (hence the name!).
- Variables can be changed by
 - assigning to them
 - The assignment statement
 - using the increment or decrement operator
 - inputting into them
 - The input statement

The Assignment Statement

- An *assignment statement*

stores a new value in a variable,
replacing the previously stored value.

The Assignment Statement

```
cans_per_pack = 8;
```

This assignment statement changes the value stored in `cans_per_pack` to be 8.

The previous value is replaced.

The Assignment Statement

SYNTAX 2.2 Assignment

This is an initialization
of a new variable,
NOT an assignment.

```
double total = 0;
```

This is an assignment.

```
.  
.   
total = bottles * BOTTLE_VOLUME;  
.
```

The name of a previously
defined variable

The expression that replaces the previous value

```
.  
.   
total = total + cans * CAN_VOLUME;  
.
```

The same name
can occur on both sides.

The Assignment Statement

- There is an important difference between a variable definition and an assignment statement:

```
int cans_per_pack = 6; // Variable definition
...
cans_per_pack = 8; // Assignment statement
```

- The first statement is the *definition* of `cans_per_pack`.
- The second statement is an *assignment statement*.
An *existing* variable's contents are replaced.

The Assignment Statement

- The = in an assignment does ***not*** mean the left hand side is equal to the right hand side as it does in math.
- = is an instruction to do something:
 copy the value of the expression on the right
 into the variable on the left.
- Consider what it would mean, mathematically, to state:
 counter = counter + 2;

counter *EQUALS* counter + 2 ?

The Assignment Statement

```
counter = 11; // set counter to 11  
counter = counter + 2; // increment
```

The Assignment Statement

```
counter = 11; // set counter to 11  
counter = counter + 2; // increment
```

- 
1. Look up what is currently in counter (11)

The Assignment Statement

```
counter = 11; // set counter to 11  
counter = counter + 2; // increment
```

1. Look up what is currently in counter (11)
2. Add 2 to that value (13)

The Assignment Statement

```
counter = 11; // set counter to 11  
counter = counter + 2; // increment
```

1. Look up what is currently in counter (11)
2. Add 2 to that value (13)
3. copy the result of the addition expression into the variable on the left, changing counter

```
printf("` ` %d' ', counter);
```

13 is shown

Constants

- Sometimes the programmer knows certain values just from analyzing the problem, for this kind of information, programmers use the reserved word **const**.
- The reserved word **const** is used to define a constant.
- A **const** is a variable whose contents cannot be changed and must be set when created.
(Most programmers just call them constants, not variables.)
- Constants are commonly written using capital letters to distinguish them visually from regular variables:

```
const int BOTTLE_VOLUME = 2;
```

Constants

Another good reason for using constants:

```
int volume = bottles * 2;
```

What does that 2 mean?

Constants

If we use a constant there is no question:

```
int volume = bottles * BOTTLE_VOLUME;
```

Any questions?

Constants

And still another good reason for using constants:

```
int bottle_volume = bottles * 2;  
int can_volume   = cans   * 2;
```

What does *that* 2 mean?

— WHICH 2?

That 2

is called a “***magic number***”

(so is that one)

because it would require magic to know what 2 means.

It is not good programming practice to use magic numbers.
Use constants.

Constants

And it can get even worse ...

Suppose that the number 2 appears hundreds of times throughout a five-hundred-line program?

Now we need to change the BOTTLE_VOLUME to 3
(because we are now using a bottle with a different shape)

How to change *only* some of those magic numbers 2's?

Constants

Constants to the rescue!

```
const double BOTTLE_VOLUME = 2.23;  
const double CAN_VOLUME = 2;
```

...

```
double bottle_volume = bottles * BOTTLE_VOLUME;  
double can_volume = cans * CAN_VOLUME;
```

(Look, no magic numbers!)

float is a 32bit single precision floating point number : 1 bit for the sign, 8 bits for the exponent and 23 for the value.

double is 64 bit double precision floating point number : 1 bit for the sign, 11 bits for the exponent, and 52 bits for the value.

Comments

- *Comments* are explanations for human readers of your code (other programmers).
- The compiler ignores comments completely.

```
double can_volume = 0.355; // Liters in a 12-ounce can
```

Comment



Comments

Comments can be written in two styles:

- Single line:

```
double can_volume = 0.355; // Liters in a 12-ounce can
```

The compiler ignores everything after `//` to the end of line

- Multiline for longer comments:

```
/*  
    This program computes the volume (in liters)  
    of a six-pack of soda cans.  
*/
```

Notice All the Issues Covered So Far

```
/*
This program computes the volume (in liters) of a six-pack of soda
cans and the total volume of a six-pack and a two-liter bottle.
*/

int main()
{
    int cans_per_pack = 6;
    const double CAN_VOLUME = 0.355; // Liters in a 12-ounce can
    double total_volume = cans_per_pack * CAN_VOLUME;

    printf("A six-pack of 12-ounce cans contains %f liters",
total_volume);
    const double BOTTLE_VOLUME = 2; // Two-liter bottle

    total_volume = total_volume + BOTTLE_VOLUME;

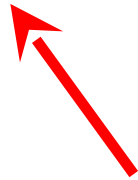
    printf("A six-pack and a two-liter bottle contain %f
}    liters", total_volume);
```

Common Error – Using Undefined Variables

You must define a variable before you use it for the first time. For example, the following sequence of statements would not be legal:

```
double can_volume = 12 * liter_per_ounce;  
double liter_per_ounce = 0.0296;
```

? ?



Statements are compiled in top to bottom order.

When the compiler reaches the first statement, it does not know that `liter_per_ounce` will be defined in the next line, and it reports an error.

Common Error – Using Uninitialized Variables

Initializing a variable is not required, but there is always a value in every variable, even uninitialized ones. Some value will be there, the flotsam left over from some previous calculation or simply the random value there when the transistors in RAM were first turned on.

```
int bottles; // Forgot to initialize  
int bottle_volume = bottles * 2; // Result is unpredictable
```

What value would be output from the following statement?

```
printf("`%f\n'", bottle_volume); // Unpredictable
```

Numeric Types

In addition to the `int` and `double` types, C has several other numeric types.

C has two other floating-point types.

The `float` type uses half the storage of the `double` type that we use in this book, but it can only store 6–7 digits.

Numeric Types

Many years ago, when computers had far less memory than they have today, `float` was the standard type for floating-point computations, and programmers would *indulge in the luxury of “double precision”* only when they really needed the additional digits.

Today, the `float` type is rarely used.

The third type is called `long double` and is for quadruple precision (19 digits). Most contemporary compilers use this type when a programmer asks for a `double` so just choosing `double` is what is done most often.

Numeric Types

By the way, these numbers are called “floating-point” because of their internal representation in the computer.

Consider the numbers 29600, 2.96, and 0.0296.

They can be represented in a very similar way: namely, as a sequence of the significant digits: 296 and an indication of the position of the decimal point.

When the values are multiplied or divided by 10, only the position of the decimal point changes; it “floats”.

Computers use base 2, not base 10, but the principle is the same.

Numeric Types

Table 4 Number Types

Type	Typical Range	Typical Size
int	−2,147,483,648 . . . 2,147,483,647 (about 2 billion)	4 bytes
unsigned	0 . . . 4,294,967,295	4 bytes
short	−32,768 . . . 32,767	2 bytes
unsigned short	0 . . . 65,535	2 bytes
double	The double-precision floating-point type, with a range of about $\pm 10^{308}$ and about 15 significant decimal digits	8 bytes
float	The single-precision floating-point type, with a range of about $\pm 10^{38}$ and about 7 significant decimal digits	4 bytes

Numeric Types

In addition to the `int` type, C has these additional integer types: `short`, `long`.

For each integer type, there is an unsigned equivalent: `unsigned short`, `unsigned long`

For example, the `short` type typically has a range from $-32,768$ to $32,767$, whereas `unsigned short` has a range from 0 to $65,535$. These strange-looking limits are the result of the use of binary numbers in computers.

A `short` value uses 16 binary digits, which can encode $2^{16} = 65,536$ values.

Numeric Types

The C Standard does not completely specify the number of bytes or ranges for numeric types.

Table 4 showed typical values.

Numeric Types

Some compiler manufacturers have added other types like:

long long

long long

−9,223,372,036,854,775,808 ... 9,223,372,036,854,775,807

8 bytes

This type is not in the C standard.

Numeric Ranges and Precisions

The `int` type has a *limited range*:

On most platforms, it can represent numbers up to a little more than two billion.

For many applications, this is not a problem, but you cannot use an `int` to represent the world population.

If a computation yields a value that is outside the `int` range, the result *overflows*.

No error is displayed.

Instead, the result is *truncated* to fit into an `int`, yielding a value that is most likely not what you thought.

Numeric Ranges and Precisions

For example:

```
int one_billion = 1000000000;  
printf(``%d\n'', 3 * one_billion);
```

displays `-1294967296` because the result is larger than an `int` can hold.

In situations such as this, you could instead use the `double` type.

However, you will need to think about a related issue: *roundoff errors*.

Arithmetic Operators

C has the same arithmetic operators as a calculator:



* for multiplication: $a * b$

(not $a \cdot b$ or ab as in math)

/ for division: a / b

(not \div or a fraction bar as in math)

+ for addition: $a + b$

- for subtraction: $a - b$

Arithmetic Operators

Just as in regular algebraic notation,
* and / have higher precedence
than + and −.

In $a + b / 2$,
the $b / 2$ happens first.

Increment and Decrement

- Changing a variable by adding or subtracting 1 is so common that there is a special shorthand for these:

The increment and decrement operators.

```
counter++; // add 1 to counter
```

```
counter--; // subtract 1 from counter
```


Integer Division and Remainder

The % operator computes the remainder of an integer division.

It is called the ***modulus operator***
(also modulo and mod)



It has nothing to do with the % key on a calculator

Integer Division and Remainder

Time to break open the bank.

You want to determine the value in dollars and cents stored in the bank.

You obtain the dollars through an integer division by 100.

The integer division discards the remainder.

To obtain the remainder, use the % operator:

```
int pennies = 1729;  
int dollars = pennies / 100; // Sets dollars to 17  
int cents = pennies % 100; // Sets cents to 29
```

Integer Division and Remainder

```
dollars = bank / 100;
```

```
cents = bank % 100;
```

Converting Floating-Point Numbers to Integers

- When a floating-point value is assigned to an integer variable, the fractional part is discarded:

```
double price = 2.55;  
int dollars = price;  
           // Sets dollars to 2
```

- You probably want to round to the *nearest* integer.
To round a positive floating-point value to the nearest integer, add 0.5 and then convert to an integer:

```
int dollars = price + 0.5;  
           // Rounds to the nearest integer
```

Powers and Roots

What about this?

$$b + \left(1 + \frac{r}{100} \right)^n$$

Inside the parentheses is easy:

$$1 + (r / 100)$$

But that raised to the n ?

Powers and Roots

- In C, there are no symbols for powers and roots. To compute them, you must call *functions*.
- The C library defines many mathematical functions such as **sqrt** (square root) and **pow** (raising to a power).
- To use the functions in this library, called the **cmath** library, you must place the line:

```
#include <math.h> in C
```

at the top of your program file.

Powers and Roots

The power function has the base followed by a comma followed by the power to raise the base to:

```
pow(base, exponent)
```

Using the `pow` function:

```
b * pow(1 + r / 100, n)
```

Powers and Roots

Table 5 Arithmetic Expressions

Mathematical Expression	C Expression	Comments
$\frac{x + y}{2}$	<code>(x + y) / 2</code>	The parentheses are required; <code>x + y / 2</code> computes $x + \frac{y}{2}$.
$\frac{xy}{2}$	<code>x * y / 2</code>	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1 + \frac{r}{100}\right)^n$	<code>pow(1 + r / 100, n)</code>	Remember to add <code>#include <cmath></code> to the top of your program.
$\sqrt{a^2 + b^2}$	<code>sqrt(a * a + b * b)</code>	<code>a * a</code> is simpler than <code>pow(a, 2)</code> .
$\frac{i + j + k}{3}$	<code>(i + j + k) / 3.0</code>	If <i>i</i> , <i>j</i> , and <i>k</i> are integers, using a denominator of 3.0 forces floating-point division.

Other Mathematical Functions

Table 6 Other Mathematical Functions

Function	Description
<code>sin(x)</code>	sine of x (x in radians)
<code>cos(x)</code>	cosine of x
<code>tan(x)</code>	tangent of x
<code>log10(x)</code>	(decimal log) $\log_{10}(x)$, $x > 0$
<code>abs(x)</code>	absolute value $ x $

Common Error – Unintended Integer Division

- If both arguments of `/` are integers, the remainder is discarded:

`7 / 3` is `2`, not `2.5`

- but

`7.0 / 4.0`

`7 / 4.0`

`7.0 / 4`

- all yield `1.75`.

Common Error – Unintended Integer Division

It is unfortunate that C uses the same symbol: `/` for both integer and floating-point division. These are really quite different operations.

It is a common error to use integer division by accident. Consider this segment that computes the average of three integers:

```
printf("Please enter your last three test scores: ");  
int s1, s2, s3;  
scanf("%d %d %d", &s1, &s2, &s3);  
double average = (s1 + s2 + s3) / 3;  
printf("Your average score is %f\n", average);
```

↑
↑
↑
Error

Common Error – Unintended Integer Division

What could be wrong with that?

Of course, the average of `s1`, `s2`, and `s3` is

$$(s1 + s2 + s3) / 3$$

Here, however, the `/` does not mean division in the mathematical sense.

It denotes integer division because both `(s1 + s2 + s3)` and `3` are integers.

Common Error – Unintended Integer Division

For example, if the scores add up to 14,
the average is computed to be 4.

WHAT?

Yes, the result of the integer division of 14 by 3 is 4
How many times does 3 evenly divide into 14?
Right!

That integer 4 is then moved into the floating-point
variable **average**.

So 4.0 is stored.

That's not what I want!

Common Error – Unintended Integer Division

The remedy is to make the numerator or denominator into a floating-point number:

```
double total = s1 + s2 + s3;  
double average = total / 3;
```

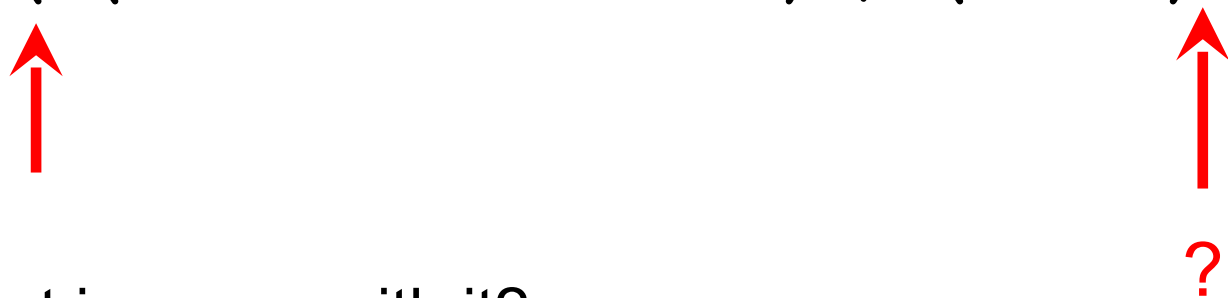
or

```
double average = (s1 + s2 + s3) / 3.0;
```

Common Error – Unbalanced Parentheses

Consider the expression

$(- (b * b - 4 * a * c) / (2 * a)$



What is wrong with it?

The parentheses are *unbalanced*.
This is very common with complicated expressions.

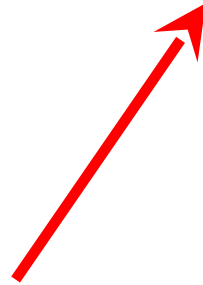
Common Error – Unbalanced Parentheses

Now consider this expression

`- (b * b - (4 * a * c))) / 2 * a)`

It is still is not correct.

There are too many closing parentheses.



Common Error – Unbalanced Parentheses – A Solution

The Muttering Method

Count (not out loud, of course!)

starting with 1 at the 1st parenthesis

add one for each (
 subtract one for each)

$$- \left(b * b - \left(4 * a * c \right) \right) / 2 * a$$

-1 OH NO!
(still to yourself – careful!)

If your count is not 0 when you finish, or if you ever drop to -1, STOP, something is wrong.

Common Error – Forgetting Header Files

Every program that carries out input or output needs the `<stdio.h>` header in C.

If you use mathematical functions such as `sqrt`, you need to include `<math.h>` in C.

If you forget to include the appropriate header file, the compiler will not know symbols such as `cout` or `sqrt`.

If the compiler complains about an undefined function or symbol, check your header files.

Common Error – Roundoff Errors

This program produces the wrong output:

```
#include <stdio.h>

int main()
{
    double price = 4.35;
    int cents = 100 * price;
        // Should be 100 * 4.35 = 435
    printf("cent is %d \n", cents);
        // Prints 434!
    return 0;
}
```

Why?

Common Error – Roundoff Errors

- In the processor hardware, numbers are represented in the binary number system, not in decimal.
- In the binary system, there is no exact representation for 4.35, just as there is no exact representation for $\frac{1}{3}$ in the decimal system.
The representation used by the computer is just a little less than 4.35, so 100 times that value is just a little less than 435.
- The remedy is to add 0.5 in order to round to the nearest integer:

```
int cents = 100 * price + 0.5;
```

Spaces in Expressions

It is easier to read

```
x1 = (-b + sqrt(b * b - 4 * a * c)) / (2 * a) ;
```

than

```
x1=(-b+sqrt(b*b-4*a*c))/(2*a) ;
```

It really is easier to read with spaces!

So always use spaces around all operators: $+$ $-$ $*$ $/$ $\%$ $=$

Spaces in Expressions

However, don't put a space after a *unary* minus:
that's a `-` used to negate a single quantity like this: `-b`

That way, it can be easily distinguished from a *binary* minus,
as in `a - b`

It is customary *not* to put a space after a function name.

Write `sqrt(x)`
not `sqrt (x)`

Occasionally, you need to store a value into a variable of a different type.

Whenever there is the risk of information loss, the compiler generally issues a warning.

It's not a compilation error to lose information. But it may not be what a programmer intended.

Casts

For example, if you store a `double` value into an `int` variable, information is lost in two ways:

The fractional part *will* be lost.

```
int n = 1.99999; // NO
```

1 is stored
(the decimal part is truncated)

The magnitude may be too large.

```
int n = 1.0E100; // NO
```

is not likely to work, because 10100 is larger than the largest representable integer.

A ***cast*** is a conversion from one type (such as **`double`**) to another type (such as **`int`**).

This is not safe in general, but if you know it to be safe in a particular circumstance, casting is the *only* way to do the conversion.

Nevertheless, sometimes you do want to convert a floating-point value into an integer value.

If you are prepared to lose the fractional part and you know that this particular floating point number is not larger than the largest possible integer, then you can turn off the warning by using a cast.

.

It's not really about turning off warnings...

Sometimes you *need* to cast a value to a different type.

Consider money.
(A good choice of topic)

```
double change; // change owed  
change = 999.89;
```

To annoy customers who actually want change when they pay with \$1000 bills, we say:

“Sorry, we can only give change in pennies.”

(in a pleasantly lilting voice, of course)

How many pennies do we owe them?

We need to cast the change owed into the correct type for pennies.

A bank would round down to the nearest penny, of course, but we will do the right thing (even to this annoying customer)...

How to "round up" to the next whole penny?

Add 0.5 to the change and then **cast** that amount into an **int** value, storing the number of pennies into an **int** variable.

```
int cents; // pennies owed
```

You express a cast in C using **(type_name) expression**

Casts

```
double change = 99.98;  
int cents = (int) (10 * change + 0.5);
```

You put the value you want to convert inside the ()

and you get the value converted to the type.

Try

```
printf("` `%d\n'", (int) change); //99
```

Try

```
printf("` `%d\n'", cents); //1000
```

Combining Assignment and Arithmetic

In C, you can combine arithmetic and assignments.
For example, the statement

```
total += cans * CAN_VOLUME;
```

is a shortcut for

```
total = total + cans * CAN_VOLUME;
```

Similarly,

```
total *= 2;
```

is another way of writing

```
total = total * 2;
```

Many programmers *prefer* using this form of coding.

Input

- Sometimes the programmer does not know what should be stored in a variable – but the user does.
- The programmer must get the input value from the user
 - Users need to be prompted
(how else would they know they need to type something?)
 - Prompts are done in output statements
- The keyboard needs to be read from
 - This is done with an input statement

The input statement

- To read values from the keyboard, we have a function called **scanf** in C.

```
scanf( ``%d'' , &bottles) ;
```

Of course, **bottles** must be defined as integer earlier.

Input

You can read more than one value in a single input statement:

```
printf("Enter the number of bottles and cans: ");  
scanf("%d%d", &bottles, &cans);
```

The user can supply both inputs on the same line:

```
Enter the number of bottles and cans: 2 6
```

Input

You can read more than one value in a single input statement:

```
printf("Enter the number of bottles  
and cans: ");  
scanf("%d%d", &bottles, &cans);
```

Alternatively, the user can press the Enter key after each input:

```
Enter the number of bottles and cans: 2  
6
```

Formatted Output

	Commencement			Term	Expiration	
	Month	Day	Year		Month	Day
Ornamental	June	14	1926	5 yr	June	14
Ornamental	April	24	1926	5 yr	April	24
slager	Mar	14	1923	5 yr	Mar	14
slager	Feb.	9	1923	old	Mar	14
slager	Oct	20	1925	5 yr	Oct	20
Lab ¹ ¹⁰ ¹¹ ¹² ¹³ ¹⁴ ¹⁵ ¹⁶ ¹⁷ ¹⁸ ¹⁹ ²⁰ ²¹ ²² ²³ ²⁴ ²⁵ ²⁶ ²⁷ ²⁸ ²⁹ ³⁰ ³¹ ³² ³³ ³⁴ ³⁵ ³⁶ ³⁷ ³⁸ ³⁹ ⁴⁰ ⁴¹ ⁴² ⁴³ ⁴⁴ ⁴⁵ ⁴⁶ ⁴⁷ ⁴⁸ ⁴⁹ ⁵⁰ ⁵¹ ⁵² ⁵³ ⁵⁴ ⁵⁵ ⁵⁶ ⁵⁷ ⁵⁸ ⁵⁹ ⁶⁰ ⁶¹ ⁶² ⁶³ ⁶⁴ ⁶⁵ ⁶⁶ ⁶⁷ ⁶⁸ ⁶⁹ ⁷⁰ ⁷¹ ⁷² ⁷³ ⁷⁴ ⁷⁵ ⁷⁶ ⁷⁷ ⁷⁸ ⁷⁹ ⁸⁰ ⁸¹ ⁸² ⁸³ ⁸⁴ ⁸⁵ ⁸⁶ ⁸⁷ ⁸⁸ ⁸⁹ ⁹⁰ ⁹¹ ⁹² ⁹³ ⁹⁴ ⁹⁵ ⁹⁶ ⁹⁷ ⁹⁸ ⁹⁹ ¹⁰⁰ ¹⁰¹ ¹⁰² ¹⁰³ ¹⁰⁴ ¹⁰⁵ ¹⁰⁶ ¹⁰⁷ ¹⁰⁸ ¹⁰⁹ ¹¹⁰ ¹¹¹ ¹¹² ¹¹³ ¹¹⁴ ¹¹⁵ ¹¹⁶ ¹¹⁷ ¹¹⁸ ¹¹⁹ ¹²⁰ ¹²¹ ¹²² ¹²³ ¹²⁴ ¹²⁵ ¹²⁶ ¹²⁷ ¹²⁸ ¹²⁹ ¹³⁰ ¹³¹ ¹³² ¹³³ ¹³⁴ ¹³⁵ ¹³⁶ ¹³⁷ ¹³⁸ ¹³⁹ ¹⁴⁰ ¹⁴¹ ¹⁴² ¹⁴³ ¹⁴⁴ ¹⁴⁵ ¹⁴⁶ ¹⁴⁷ ¹⁴⁸ ¹⁴⁹ ¹⁵⁰ ¹⁵¹ ¹⁵² ¹⁵³ ¹⁵⁴ ¹⁵⁵ ¹⁵⁶ ¹⁵⁷ ¹⁵⁸ ¹⁵⁹ ¹⁶⁰ ¹⁶¹ ¹⁶² ¹⁶³ ¹⁶⁴ ¹⁶⁵ ¹⁶⁶ ¹⁶⁷ ¹⁶⁸ ¹⁶⁹ ¹⁷⁰ ¹⁷¹ ¹⁷² ¹⁷³ ¹⁷⁴ ¹⁷⁵ ¹⁷⁶ ¹⁷⁷ ¹⁷⁸ ¹⁷⁹ ¹⁸⁰ ¹⁸¹ ¹⁸² ¹⁸³ ¹⁸⁴ ¹⁸⁵ ¹⁸⁶ ¹⁸⁷ ¹⁸⁸ ¹⁸⁹ ¹⁹⁰ ¹⁹¹ ¹⁹² ¹⁹³ ¹⁹⁴ ¹⁹⁵ ¹⁹⁶ ¹⁹⁷ ¹⁹⁸ ¹⁹⁹ ²⁰⁰ ²⁰¹ ²⁰² ²⁰³ ²⁰⁴ ²⁰⁵ ²⁰⁶ ²⁰⁷ ²⁰⁸ ²⁰⁹ ²¹⁰ ²¹¹ ²¹² ²¹³ ²¹⁴ ²¹⁵ ²¹⁶ ²¹⁷ ²¹⁸ ²¹⁹ ²²⁰ ²²¹ ²²² ²²³ ²²⁴ ²²⁵ ²²⁶ ²²⁷ ²²⁸ ²²⁹ ²³⁰ ²³¹ ²³² ²³³ ²³⁴ ²³⁵ ²³⁶ ²³⁷ ²³⁸ ²³⁹ ²⁴⁰ ²⁴¹ ²⁴² ²⁴³ ²⁴⁴ ²⁴⁵ ²⁴⁶ ²⁴⁷ ²⁴⁸ ²⁴⁹ ²⁵⁰ ²⁵¹ ²⁵² ²⁵³ ²⁵⁴ ²⁵⁵ ²⁵⁶ ²⁵⁷ ²⁵⁸ ²⁵⁹ ²⁶⁰ ²⁶¹ ²⁶² ²⁶³ ²⁶⁴ ²⁶⁵ ²⁶⁶ ²⁶⁷ ²⁶⁸ ²⁶⁹ ²⁷⁰ ²⁷¹ ²⁷² ²⁷³ ²⁷⁴ ²⁷⁵ ²⁷⁶ ²⁷⁷ ²⁷⁸ ²⁷⁹ ²⁸⁰ ²⁸¹ ²⁸² ²⁸³ ²⁸⁴ ²⁸⁵ ²⁸⁶ ²⁸⁷ ²⁸⁸ ²⁸⁹ ²⁹⁰ ²⁹¹ ²⁹² ²⁹³ ²⁹⁴ ²⁹⁵ ²⁹⁶ ²⁹⁷ ²⁹⁸ ²⁹⁹ ³⁰⁰ ³⁰¹ ³⁰² ³⁰³ ³⁰⁴ ³⁰⁵ ³⁰⁶ ³⁰⁷ ³⁰⁸ ³⁰⁹ ³¹⁰ ³¹¹ ³¹² ³¹³ ³¹⁴ ³¹⁵ ³¹⁶ ³¹⁷ ³¹⁸ ³¹⁹ ³²⁰ ³²¹ ³²² ³²³ ³²⁴ ³²⁵ ³²⁶ ³²⁷ ³²⁸ ³²⁹ ³³⁰ ³³¹ ³³² ³³³ ³³⁴ ³³⁵ ³³⁶ ³³⁷ ³³⁸ ³³⁹ ³⁴⁰ ³⁴¹ ³⁴² ³⁴³ ³⁴⁴ ³⁴⁵ ³⁴⁶ ³⁴⁷ ³⁴⁸ ³⁴⁹ ³⁵⁰ ³⁵¹ ³⁵² ³⁵³ ³⁵⁴ ³⁵⁵ ³⁵⁶ ³⁵⁷ ³⁵⁸ ³⁵⁹ ³⁶⁰ ³⁶¹ ³⁶² ³⁶³ ³⁶⁴ ³⁶⁵ ³⁶⁶ ³⁶⁷ ³⁶⁸ ³⁶⁹ ³⁷⁰ ³⁷¹ ³⁷² ³⁷³ ³⁷⁴ ³⁷⁵ ³⁷⁶ ³⁷⁷ ³⁷⁸ ³⁷⁹ ³⁸⁰ ³⁸¹ ³⁸² ³⁸³ ³⁸⁴ ³⁸⁵ ³⁸⁶ ³⁸⁷ ³⁸⁸ ³⁸⁹ ³⁹⁰ ³⁹¹ ³⁹² ³⁹³ ³⁹⁴ ³⁹⁵ ³⁹⁶ ³⁹⁷ ³⁹⁸ ³⁹⁹ ⁴⁰⁰ ⁴⁰¹ ⁴⁰² ⁴⁰³ ⁴⁰⁴ ⁴⁰⁵ ⁴⁰⁶ ⁴⁰⁷ ⁴⁰⁸ ⁴⁰⁹ ⁴¹⁰ ⁴¹¹ ⁴¹² ⁴¹³ ⁴¹⁴ ⁴¹⁵ ⁴¹⁶ ⁴¹⁷ ⁴¹⁸ ⁴¹⁹ ⁴²⁰ ⁴²¹ ⁴²² ⁴²³ ⁴²⁴ ⁴²⁵ ⁴²⁶	Mar	15	1924	5 yr	Mar	15
Quinn	Nov.	14	1924	5 yr	Nov	14
Sr.	Sept	5	1925	5 yr	Sept	5
Dodge	May	22	1926	5 yr	May	22
I.						

Formatted Output

- When you print an amount in dollars and cents, you usually want it to be *rounded* to two significant digits.
- You learned how to actually round off and store a value but, for output, we want to round off *only* for display.

Which do you think the user prefers to see on her gas bill?

Price per liter: \$1.22

or

Price per liter: \$1.21997

Formatted Output

You can combine manipulators and values to be displayed into a single statement:

```
price_per_liter = 1.21997;  
printf("Price per liter: $%.2f \n",  
       price_per_liter);
```

This code produces this output:

```
Price per liter: $1.22
```

Formatted Output

This code:

```
double price_per_ounce_1 = 10.2372;
double price_per_ounce_2 = 117.2;
double price_per_ounce_3 = 6.9923435;

printf("%8.2f\n", price_per_ounce_1);    printf("%8.2f\n",
price_per_ounce_2);    printf("%8.2f\n",
price_per_ounce_3);    printf("-----\n");
```

produces this output:

```
    10.24
   117.20
     6.99
-----
```


A Complete Program for Volumes

```
#include <stdio.h>

int main()
{
    const double CANS_PER_PACK = 6;
    double pack_price, can_volume;

    // Read price per pack
    printf("Please enter the price for a six-pack: ");
    scanf("%f", &pack_price);

    // Read can volume
    printf("Please enter the volume for each can (in ounces): ");
    scanf("%f", &can_volume);
```

A Complete Program for Volumes

```
// Compute pack volume
double pack_volume = can_volume * CANS_PER_PACK;

// Compute and print price per ounce
double price_per_ounce = pack_price / pack_volume;

printf("Price per ounce: %f\n", price_per_ounce);

return 0;
}
```

Chapter Summary

Write variable definitions in C.

- A variable is a storage location with a name.
- When defining a variable, you usually specify an initial value.
- When defining a variable, you also specify the type of its values.
- Use the `int` type for numbers that cannot have a fractional part.
- Use the `double` type for floating-point numbers.



Chapter Summary

- An assignment statement stores a new value in a variable, replacing the previously stored value.
- The assignment operator `=` does *not* denote mathematical equality.
- You cannot change the value of a variable that is defined as **`const`**.
- Use comments to add explanations for humans who read your code.
The compiler ignores comments.



Use the arithmetic operations in C

- Use `*` for multiplication and `/` for division.
- The `++` operator adds 1 to a variable; the `--` operator subtracts 1.
- If both arguments of `/` are integers, the remainder is discarded.
- The `%` operator computes the remainder of an integer division.
- Assigning a floating-point variable to an integer drops the fractional part.
- The C++ library defines many mathematical functions such as `sqrt` (square root) and `pow` (raising to a power).

Chapter Summary

Write programs that read user input and write formatted output.

- Use `scanf` to read a value and place it in a variable.
- You specify how values should be formatted.

Carry out hand calculations when developing an algorithm.

- Pick concrete values for a typical situation to use in a hand calculation.