© samxmeg/iStockphoto.

Chapter Two: Fundamental Data Types

Chapter Goals

- To define and initialize variables and constants
- To understand the properties and limitations of integer and floating-point numbers
- To write arithmetic expressions and assignment statements in C++
- To appreciate the importance of comments and good code layout
- To create programs that read and process input, and display the results
- To process strings, using the standard C++ `string` type

Topic 1

1. Variables
2. Arithmetic
3. Input and output
4. Problem solving: first do it by hand
5. Strings
6. Chapter summary

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

Variable

- A variable has a type, a name, and can have different values at different times.

- A variable is like a container.
- Type tells the memory size.
- Name is like a label on that container.
- Value is the actual content.

Variable Definitions

- When creating variables, the programmer specifies the type of information to be stored.
  - (more on types later)
- 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: Example

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 hold integers. Other variable types covered later will hold strings and floating-point numbers.

`= 6`

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

Like all statements, it must end with a semicolon.

Assignment Operator =

- Unlike most operators, assignment Operator = runs from right to left.
- First it evaluate the right expression’s value, then copy that value to the left hand side variable.

```
int unit_price = 6;
int num_units = 2;
int total_price = unit_price * num_units;
```

Variable Definitions: More Examples

Table 1: Variable Definitions in C++	
	Comment
`int cans = 6;`	Defines an integer variable and initializes it with 6.
`int total = cans + bottles;`	The initial value need not be a constant. (Of course, cans and bottles must have been previously defined.)
`int bottles = "10";`	*Error: You cannot initialize an* `int` *variable with a* `string`*.*
`int bottles;`	Defines an integer variable without initializing it. This can be a cause for errors—see Common Error 2.2.
`int cans, bottles;`	Defines two integer variables in a single statement. In this book, we will define each variable in a separate statement.
`bottles = 1;`	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.

Number Types

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

There are rules for writing literal values:

Number Literals: Table 2

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 × 106 or 1000000. Numbers in exponential notation always have type double.
2.96E-2	`double`	Negative exponent: 2.96 × 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.

Number Type Examples

- What is the C++ type of each of the following numbers? Write "error" if the number is not valid.

3	
-3	
3.14	
3.0	
3E-6	
300,000	
3 14/100	

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**.

Variable Naming Rules

1. Variable names must start with a letter or the underscore (_) character, and the remaining characters must be letters numbers, or underscores.
2. Do not 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. See Appendix B.

Variable Name Examples: Table 3

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)
`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.*
`ltr/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

Assignment Statement Example

- An *assignment statement*

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

```
// A variable is only declared once.
int cans_per_pack = 6;
//A variable can be used many times
//after its declaration.
cans_per_pack = 8;
```

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

The previous value is replaced.

Assignment Statement: Defining vs. Assigning

- 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`**.
  - A variable's definition must occur ***only once*** in a program
- The second statement is an *assignment statement.* An *existing* variable's contents are replaced.
  - The same variable may be in several assignment statements in a program.

The Meaning of the Assignment = Symbol

- 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 ?

Assignment Examples

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

1. First statement assigns 11 to counter
2. Second statement looks up what is currently in counter (11)
3. *Then it adds 2 and copies* the result of the addition *into* the variable on the left, changing counter to 13

Constants

- Sometimes the programmer knows certain values just from analyzing the problem
  - For this kind of information, 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 double BOTTLE_VOLUME = 2;
```

Constants Prevent Unclear Numbers in Code

Another good reason for using constants:

```
double volume = bottles * 2;
```

What does that 2 mean?

If we use a constant there is no question:

```
double volume = bottles * BOTTLE_VOLUME;
```

Any questions?

Constants Prevent Unclear Numbers in Code (2)

And still another good reason for using constants:

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

What does *that* 2 mean?

— *WHICH 2?*

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

Constants Prevent Unclear Numbers in Code (3)

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 2.23 (because we are now using a bottle with a different shape)

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

Constants again

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;
```

Comments

- *Comments* are explanations for human readers of your code (other programmers or your instructor).
- The compiler ignores comments completely.
- A leading double slash // tells the compiler the remainder of this line is a comment, to be ignored
- For example,

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

Comments: // or `/*   multi-line */`

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, where the compiler ignores everything between `/*` and `*/`

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

Complete Program: volume1.cpp

```
/*
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;

   cout << "A six-pack of 12-ounce cans contains "
      << total_volume << " liters." << endl;

   const double BOTTLE_VOLUME = 2; // Two-liter bottle

   total_volume = total_volume + BOTTLE_VOLUME;

   cout << "A six-pack and a two-liter bottle contain "
      << total_volume << " liters." << endl;

   return 0;
}
```

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?

```
cout << bottle_volume << endl; // Unpredictable
```

More Numeric Types in C++

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 **`float`** can only store 6–7 digits.

The `float` and `long double` 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.

- Most contemporary compilers use this type when a programmer asks for a **`double`** so just choose **`double`**.

Floating Point

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:

- a sequence of the significant digits: 296
- 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.

Range of unsigned int Type

- How many unsigned (zero or positive) integer can 3-bit represent?
- Each bit is either 0 or 1.

Binary representation	Decimal equivalent
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

- *3-bit binary number represents* ***UNSIGNED*** *integers in* $[0, 2^3-1]$
- *An int in C++ has 32 bits.*
- *32-bit represents* $[0, 2^{32}-1]$ ***UNSIGNED*** *integers, where* $2^{32}-1$ *is 4294967295.*

Range of signed int Type

- How many signed (negative, zero or positive) integers can 3-bit represent?
- Each bit is either 0 or 1.

Binary representation	Decimal equivalent
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

- *3-bit binary number represents signed integers in* $[-2^2, 2^2-1]$.
- *An int in C++ has 32 bits. In signed number, the leftmost bit is sign bit, 0 is non-negative, 1 is negative.*
- *32-bit represents* $[-2^{31}, 2^{31}-1]$ ***SIGNED*** *integers, where* $2^{31}-1$ *is 2147483647.*

Numeric Types in C++: Table 4

Type	Typical Range	Typical Size (Bytes)
`int`	–2,147,483,648 … 2,147,483,647 (about 2 billion)	4
`unsigned`	0 … 4294967295	4
`short`	–32,768 … 32,767	2
`unsigned short`	0 … 65,535	2
`long long`	–9,223,372,036,854,775,808 … 9,223,372,036,854,775,807	8
`double`	$\pm 10^{308}$ with about 15 significant decimal digits	8
`float`	$\pm 10^{38}$ with about 7 significant decimal digits	4

The C++ Standard does not completely specify the number of bytes or ranges.

Values above are typical.

Numeric Types: `short and long, unsigned`

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`**
- **`short`** typically has a range from –32,768 to 32,767
- **`unsigned short`** has range 0 to 65,535. (2^{16} -1)

A **`short`** value uses 16 bits, which can encode 2^{16} = 65,536 values.

Integer Overflow

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 WRONG.

Integer Overflow Example

For example:

```
int one_billion = 1000000000;
cout << 3 * one_billion << endl;
```

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*.

Common Error – Roundoff Errors

This program produces the wrong output, even though it uses the very precise `double` variable type:

```
#include <iostream>
using namespace std;
int main()
{
   double price = 4.35;
   int cents = 100 * price;
          // Should be 100 * 4.35 = 435
   cout << cents << endl;
          // Prints 434!
   return 0;
}
```

Why?

Common Error – Roundoff Errors, continued

- In computers, numbers are binary, not decimal.

- In the binary system, there is no exact representation for decimal 4.35, just as there is no exact representation for ⅓ in the decimal system (nor in binary).
- The binary representation is just a little less than 4.35, so 100 times that value is just a little less than 435.
  - And when a `double` value is assigned to an `int` variable, as in

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

The fractional part is simply discarded (truncated).

- The remedy is to add 0.5 in order to **round** to the nearest integer:

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

Defining Variables with "auto" (C++11 and later)

- Instead of providing a type for a variable, you can use the reserved word `auto`.
- The type is automatically deduced from the type of the initialization data:

```
auto cans = 6; // This variable has type int
const auto CAN_VOLUME = 0.355; // type is double
```

The auto type is handy for complex types like pointers to structures and objects, to be discussed in later chapters.