Chapter 3

Values and Variables

In this chapter we explore some building blocks that are used to develop C++ programs. We experiment with the following concepts:

- numeric values
- variables
- declarations
- assignment
- identifiers
- reserved words

In the next chapter we will revisit some of these concepts in the context of other data types.

3.1 Integer Values

The number four (4) is an example of a *numeric* value. In mathematics, 4 is an *integer* value. Integers are whole numbers, which means they have no fractional parts, and an integer can be positive, negative, or zero. Examples of integers include 4, −19, 0, and −1005. In contrast, 4.5 is not an integer, since it is not a whole number.

C++ supports a number of numeric and non-numeric values. In particular, C++ programs can use integer values. It is easy to write a C++ program that prints the number four, as Listing 3.1 (number4.cpp) shows.

Listing 3.1: number4.cpp

```
#include <iostream>

int main() {
    std::cout << 4 << '\n';
}
```


Notice that unlike the programs we saw earlier, Listing 3.1 (number4.cpp) does not use quotation marks (`"`). The number 4 appears unadorned with no quotes. The expression `'\n'` represents a single newline character. Multiple characters comprising a string appear in double quotes (`"`), but, in C++, a single character represents a distinct type of data and is enclosed within single quotes (`'`). (Section 3.8 provides more information about C++ characters.) Compare Listing 3.1 (number4.cpp) to Listing 3.2 (number4-alt.cpp).

Listing 3.2: number4-alt.cpp

```
#include <iostream>

int main() {
    std::cout << "4\n";
}
```

Both programs behave identically, but Listing 3.1 (number4.cpp) prints the value of the number four, while Listing 3.2 (number4-alt.cpp) prints a message containing the digit four. The distinction here seems unimportant, but we will see in Section 3.2 that the presence or absence of the quotes can make a big difference in the output.

The statement

```
std::cout << "4\n";
```

sends one thing to the output stream, the string `"4\n"`. The statement

```
std::cout << 4 << '\n';
```

sends two things to the output stream, the integer value 4 and the newline character `'\n'`.

In published C++ code you sometimes will see a statement such as the following:

```
std::cout << 4 << std::endl;
```

This statement on the surface behaves exactly like the following statement:

```
std::cout << 4 << '\n';
```

but the two expressions `std::endl` and `'\n'` do not mean exactly the same thing. The `std::endl` expression does involve a newline character, but it also performs some additional work that normally is not necessary.

Programs that do significant printing may execute faster if they terminate their output lines with `'\n'` instead of `std::endl`. The difference in speed is negligible when printing to the console, but the different can be great when printing to files or other output streams. For most of the programs we consider, the difference in program execution speed between the two is imperceptible; nonetheless, we will prefer `'\n'` for printing newlines because it is a good habit to form (and it requires five fewer keystrokes when editing code).

The three major modern computing platforms are Microsoft Windows, Apple macOS, and Linux. Windows handles newlines differently from macOS and Linux. Historically, the character `'\n'` represents a *new line*, usually known as a *line feed* or *LF* for short, and the character `'\r'` means *carriage return*, or *CR* for short. The terminology comes from old-fashioned typewriters which feed a piece of paper into a roller on a carriage that moves to the left as the user types (so the imprinted symbols form left to right). At the end of a line, the user must advance the roller so as to move the paper up by one line (LF) and move the carriage back all the way to its left (CR). Windows uses the character sequence CR LF for newlines, while macOS and Linux use LF. This can be an issue when attempting to edit text files written with an editor on one platform with an editor on a different platform.

The good news is that the C++ standard guarantees that the `std::cout` output stream translates the `'\n'` character as it appears in C++ source code into the correct character sequence for the target platform. This means you can print `'\n'` via `std::cout`, and it will behave identically on all the major platforms.

In C++ source code, integers may not contain commas. This means we must write the number two thousand, four hundred sixty-eight as `2468`, not `2,468`. Modern C++ does support single quotes (`'`) as digit separators, as in `2'468`. Using digit separators can improve the human comprehension reading large numbers in C++ source code.

In mathematics, integers are unbounded; said another way, the set of mathematical integers is infinite. In C++ the range of integers is limited because all computers have a finite amount of memory. The exact range of integers supported depends on the computer system and particular C++ compiler. C++ on most 32-bit computer systems can represent integers in the range −2,147,483,648 to +2,147,483,647.

What happens if you exceed the range of C++ integers? Try Listing 3.3 (exceed.cpp) on your system.

Listing 3.3: `exceed.cpp`

```
#include <iostream>

int main() {
    std::cout << -3000000000 << '\n';
```

```
}
```

Negative three billion is too large for 32-bit integers, however, and the program's output is obviously wrong:

```
1294967296
```

The number printed was not even negative! Most C++ compilers will issue a warning about this statement. Section 4.6 explores errors vs. warnings in more detail. If the compiler finds an error in the source, it will not generate the executable code. A warning indicates a potential problem and does not stop the compiler from producing an executable program. Here we see that the programmer should heed this warning because the program's execution produces meaningless output.

This limited range of values is common among programming languages since each number is stored in a fixed amount of memory. Larger numbers require more storage in memory. In order to model the infinite set of mathematical integers an infinite amount of memory would be needed! As we will see later, C++ supports an integer type with a greater range. Section 4.8.1 provides some details about the implementation of C++ integers.

3.2 Variables and Assignment

In algebra, variables are used to represent numbers. The same is true in C++, except C++ variables also can represent values other than numbers. Listing 3.4 (variable.cpp) uses a variable to store an integer value and then prints the value of the variable.

Listing 3.4: `variable.cpp`

```
#include <iostream>

int main() {
    int x;
    x = 10;
    std::cout << x << '\n';
}
```

The `main` function in Listing 3.4 (variable.cpp) contains three statements:

- `int x;`

  This is a *declaration* statement. All variables in a C++ program must be declared. A declaration specifies the type of a variable. The word `int` indicates that the variable is an integer. The name of the integer variable is `x`. We say that variable `x` has type `int`. C++ supports types other than integers, and some types require more or less space in the computer's memory. The compiler uses the declaration to reserve the proper amount of memory to store the variable's value. The declaration enables the compiler to verify the programmer is using the variable properly within the program; for example, we will see that integers can be added together just like in mathematics. For some other data types, however, addition is not possible and so is not allowed. The compiler can ensure that a variable involved in an addition operation is compatible with addition. It can report an error if it is not.

  The compiler will issue an error if a programmer attempts to use an undeclared variable. The compiler cannot deduce the storage requirements and cannot verify the variable's proper usage if it not declared. Once declared, a particular variable cannot be redeclared in the same context. A variable may not change its type during its lifetime.

- `x = 10;`

  This is an *assignment* statement. An assignment statement associates a value with a variable. The key to an assignment statement is the symbol = which is known as the *assignment operator*. Here the value 10 is being assigned to the variable `x`. This means the value 10 will be stored in the memory location the compiler has reserved for the variable named `x`. We need not be concerned about where the variable is stored in memory; the compiler takes care of that detail.

  After we declare a variable we may assign and reassign it as often as necessary.

- `std::cout << x << '\n';`

  This statement prints the variable `x`'s current value.

Note that the lack of quotation marks here is very important. If `x` has the value 10, the statement

```
std::cout << x << '\n';
```

prints `10`, the value of the variable `x`, but the statement

```
std::cout << "x" << '\n';
```

prints `x`, the message containing the single letter *x*.

The meaning of the assignment operator (=) is different from equality in mathematics. In mathematics, = asserts that the expression on its left is equal to the expression on its right. In C++, = makes the variable on its left take on the value of the expression on its right. It is best to read `x = 5` as "`x` is assigned the value 5," or "`x` gets the value 5." This distinction is important since in mathematics equality is symmetric: if $x = 5$, we know $5 = x$. In C++, this symmetry does not exist; the statement

```
5 = x;
```

attempts to reassign the value of the literal integer value 5, but this cannot be done, because 5 is always 5 and cannot be changed. Such a statement will produce a compiler error:

error C2106: '=' : left operand must be l-value

Variables can be reassigned different values as needed, as Listing 3.5 (multipleassignment.cpp) shows.

Listing 3.5: `multipleassignment.cpp`

```
#include <iostream>

int main() {
    int x;
    x = 10;
    std::cout << x << '\n';
    x = 20;
    std::cout << x << '\n';
    x = 30;
    std::cout << x << '\n';
}
```

Observe the each print statement in Listing 3.5 (multipleassignment.cpp) is identical, but when the program runs the print statements produce different results.

A variable may be given a value at the time of its declaration; for example, Listing 3.6 (variable-init.cpp) is a variation of Listing 3.4 (variable.cpp).

Listing 3.6: variable-init.cpp

```
#include <iostream>

int main() {
    int x = 10;
    std::cout << x << '\n';
}
```

Notice that in Listing 3.6 (variable-init.cpp) the declaration and assignment of the variable x is performed in one statement instead of two. This combined declaration and immediate assignment is called *initialization*.

C++ supports another syntax for initializing variables as shown in Listing 3.7 (alt-variable-init.cpp).

Listing 3.7: alt-variable-init.cpp

```
#include <iostream>

int main() {
    int x{10};
    std::cout << x << '\n';
}
```

This alternate form is not commonly used for simple variables, but it necessary for initializing more complicated kinds of variables called *objects*. We introduce objects in Chapter 13 and Chapter 14.

Multiple variables of the same type can be declared and, if desired, initialized in a single statement. The following statements declare three variables in one declaration statement:

```
int x, y, z;
```

The following statement declares three integer variables and initializes two of them:

```
int x = 0, y, z = 5;
```

Here y's value is undefined. The declarations may be split up into multiple declaration statements:

```
int x = 0;
int y;
int z = 5;
```

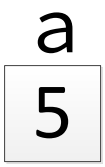
In the case of multiple declaration statements the type name (here int) must appear in each statement.

The compiler maps a variable to a location in the computer's memory. We can visualize a variable and its corresponding memory location as a box as shown in Figure 3.1.

We name the box with the variable's name. Figure 3.2 shows how the following sequence of C++ code affects memory.

```
int a, b;
a = 2;
```


Figure 3.1 Representing a variable and its memory location as a box




```
b = 5;
a = b;
b = 4;
```

Importantly, the statement

```
a = b;
```

does not mean `a` and `b` refer to the same box (memory location). After this statement `a` and `b` still refer to separate boxes (memory locations). It simply means the value stored in `b`'s box (memory location) has been copied to `a`'s box (memory location). `a` and `b` remain distinct boxes (memory locations). The original value found in `a`'s box is overwritten when the contents of `b`'s box are copied into `a`. After the assignment of `b` to `a`, the reassignment of `b` to 4 does not affect `a`.

3.3 Identifiers

While mathematicians are content with giving their variables one-letter names like `x`, programmers should use longer, more descriptive variable names. Names such as `altitude`, `sum`, and `user_name` are much better than the equally permissible `a`, `s`, and `u`. A variable's name should be related to its purpose within the program. Good variable names make programs more readable by humans. Since programs often contain many variables, well-chosen variable names can render an otherwise obscure collection of symbols more understandable.

C++ has strict rules for variable names. A variable name is one example of an *identifier*. An identifier is a word used to name things. One of the things an identifier can name is a variable. We will see in later chapters that identifiers name other things such as functions and classes. Identifiers have the following form:

- Identifiers must contain at least one character.
- The first character must be an alphabetic letter (upper or lower case) or the underscore

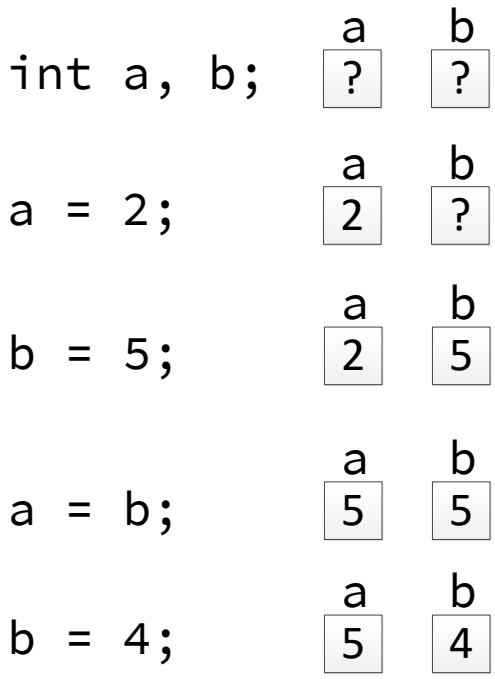
  ```
  ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_
  ```

- The remaining characters (if any) may be alphabetic characters (upper or lower case), the underscore, or a digit

  ```
  ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_0123456789
  ```

- No other characters (including spaces) are permitted in identifiers.

Figure 3.2 How memory changes during variable assignment



- A reserved word cannot be used as an identifier (see Table 3.1).

Here are some examples of valid and invalid identifiers:

- All of the following words are valid identifiers and so qualify as variable names: `x`, `x2`, `total`, `port_22`, and `FLAG`.
- None of the following words are valid identifiers: `sub-total` (dash is not a legal symbol in an identifier), `first entry` (space is not a legal symbol in an identifier), `4all` (begins with a digit), `#2` (pound sign is not a legal symbol in an identifier), and `class` (`class` is a reserved word).

C++ reserves a number of words for special use that could otherwise be used as identifiers. Called *reserved words* or *keywords*, these words are special and are used to define the structure of C++ programs and statements. Table 3.1 lists all the C++ reserved words.

The purposes of many of these reserved words are revealed throughout this book.

You may not use any of the reserved words in Table 3.1 as identifiers. Fortunately, if you accidentally attempt to use one of the reserved words in a program as a variable name, the compiler will issue an error (see Section 4.6 for more on compiler errors).

In Listing 2.1 (simple.cpp) we used several reserved words: `using`, `namespace`, and `int`. Notice that `include`, `cout`, and `main` are not reserved words.

Some programming languages do not require programmers to declare variables before they are used; the type of a variable is determined by how the variable is used. Some languages allow the same variable

alignas	decltype	namespace	struct
alignof	default	new	switch
and	delete	noexcept	template
and_eq	double	not	this
asm	do	not_eq	thread_local
auto	dynamic_cast	nullptr	throw
bitand	else	operator	true
bitor	enum	or	try
bool	explicit	or_eq	typedef
break	export	private	typeid
case	extern	protected	typename
catch	false	public	union
char	float	register	unsigned
char16_t	for	reinterpret_cast	using
char32_t	friend	return	virtual
class	goto	short	void
compl	if	signed	volatile
const	inline	sizeof	wchar_t
constexpr	int	static	while
const_cast	long	static_assert	xor
continue	mutable	static_cast	xor_eq

Table 3.1: C++ reserved words. C++ reserves these words for specific purposes in program construction. None of the words in this list may be used as an identifier; thus, you may not use any of these words to name a variable.

to assume different types as its use differs in different parts of a program. Such languages are known as *dynamically-typed languages*. C++ is a *statically-typed language*. In a statically-typed language, the type of a variable must be explicitly specified before it is used by statements in a program. While the requirement to declare all variables may initially seem like a minor annoyance, it offers several advantages:

- When variables must be declared, the compiler can catch typographical errors that dynamically-typed languages cannot detect. For example, consider the following section of code:

  ```
  int ZERO;
  ZER0 = 1;
  ```

  The identifier in the first line ends with a capital "Oh." In the second line, the identifier ends with the digit zero. The distinction may be difficult or impossible to see in a particular editor or printout of the code. A C++ compiler would immediately detect the typo in the second statement, since `ZER0` (last letter a zero) has not been declared. A dynamically-typed language would create two variables: `ZERO` and `ZER0`.

- When variables must be declared, the compiler can catch invalid operations. For example, a variable may be declared to be of type `int`, but the programmer may accidentally assign a non-numeric value to the variable. In a dynamically-typed language, the variable would silently change its type introducing an error into the program. In C++, the compiler would report the improper assignment as error, since once declared a C++ variable cannot change its type.

- Ideally, requiring the programmer to declare variables forces the programmer to plan ahead and think more carefully about the variables a program might require. The purpose of a variable is tied to its type, so the programmer must have a clear notion of the variable's purpose before declaring it. When

variable declarations are not required, a programmer can "make up" variables as needed as the code is written. The programmer need not do the simple double check of the variable's purpose that writing the variable's declaration requires. While declaring the type of a variable specifies its purpose in only a very limited way, any opportunity to catch such errors is beneficial.

- Statically-typed languages are generally more efficient than dynamically-typed languages. The compiler knows how much storage a variable requires based on its type. The space for that variable's value will not change over the life of the variable, since its type cannot change. In a dynamically typed language that allows a variable to change its type, if a variable's type changes during program execution, the storage it requires may change also, so memory for that variable must be allocated elsewhere to hold the different type. This memory reallocation at run time slows down the program's execution.

C++ is a case-sensitive language. This means that capitalization matters. `if` is a reserved word, but none of `If`, `IF`, or `iF` are reserved words. Identifiers are case sensitive also; the variable called `Name` is different from the variable called `name`.

Since it can be confusing to human readers, you should not distinguish variables merely by names that differ in capitalization. For the same reason, it is considered poor practice to give a variable the same name as a reserved word with one or more of its letters capitalized.

3.4 Additional Integer Types

C++ supports several other integer types. The type `short int`, which may be written as just `short`, represents integers that may occupy fewer bytes of memory than the `int` type. If the `short` type occupies less memory, it necessarily must represent a smaller range of integer values than the `int` type. The C++ standard does not require the `short` type to be smaller than the `int` type; in fact, they may represent the same set of integer values. The `long int` type, which may be written as just `long`, may occupy more storage than the `int` type and thus be able to represent a larger range of values. Again, the standard does not require the `long` type to be bigger then the `int` type. Finally, the `long long int` type, or just `long long`, may be larger than a `long`. The C++ standard guarantees the following relative ranges of values hold:

$$\texttt{short int} \leq \texttt{int} \leq \texttt{long int} \leq \texttt{long long int}$$

On a small embedded device, for example, all of these types may occupy the exact same amount of memory and, thus, there would be no advantage of using one type over another. On most systems, however, there will some differences in the ranges.

C++ provides integer-like types that exclude negative numbers. These types include the word *unsigned* in their names, meaning they do not allow a negative sign. The unsigned types come in various potential sizes in the same manner as the signed types. The C++ standard guarantees the following relative ranges of unsigned values:

$$\texttt{unsigned short} \leq \texttt{unsigned} \leq \texttt{unsigned long} \leq \texttt{unsigned long long}$$

Table 3.2 lists the differences among the signed and unsigned integer types in Visual C++. Notice that the corresponding signed and unsigned integer times occupy the same amount of memory. As a result, the unsigned types provide twice the range of positive values available to their signed counterparts. For applications that do not require negative numbers, the `unsigned` type may be a more appropriate option.

Type Name	Short Name	Storage	Smallest Magnitude	Largest Magnitude
`short int`	`short`	2 bytes	−32,768	32,767
`int`	`int`	4 bytes	−2,147,483,648	2,147,483,647
`long int`	`long`	4 bytes	−2,147,483,648	2,147,483,647
`long long int`	`long long`	8 bytes	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
`unsigned short`	`unsigned short`	2 bytes	0	65,535
`unsigned int`	`unsigned`	4 bytes	0	4,294,967,295
`unsigned long int`	`unsigned long`	4 bytes	0	4,294,967,295
`unsigned long long int`	`unsigned long long`	8 bytes	0	18,446,744,073,709,551,615

Table 3.2: Characteristics of Visual C++ Integer Types

Within the source code, any unadorned numerical literal without a decimal point is interpreted as an `int` literal; for example, in the statement

```
int x = 4456;
```

the literal value `4456` is an `int`. In order to represent 4456 as an `long`, append an `L`, as in

```
long x = 4456L;
```

C++ also permits the lower-case *l* (*elle*), as in

```
long x = 4456l;
```

but you should avoid it since on many display and printer fonts it looks too much like the digit `1` (one). Use the `LL` suffix for `long long` literals. The suffixes for the unsigned integers are `u` (`unsigned`), `us` (`unsigned short`), `uL` (`unsigned long`), and `uLL` (`unsigned long long`). The capitalization is unimportant, although capital `L`s are preferred.

Within C++ source code all integer literals are `int` values unless an `L` or `l` is appended to the end of the number; for example, `2` is an `int` literal, while `2L` is a `long` literal.

3.5 Floating-point Types

Many computational tasks require numbers that have fractional parts. For example, the formula from mathematics to compute the area of a circle given the circle's radius, involves the value π, which is approximately 3.14159. C++ supports such non-integer numbers, and they are called *floating-point* numbers. The name comes from the fact that during mathematical calculations the decimal point can move or "float" to various positions within the number to maintain the proper number of significant digits. The types `float` and `double` represent different types of floating-point numbers. The type `double` is used more often, since it stands for "double-precision floating-point," and it can represent a wider range of values with more digits of precision. The `float` type represents single-precision floating-point values that are less precise. Table 3.3 provides some information about floating-point values as commonly implemented on 32-bit computer systems. Floating point numbers can be both positive and negative.

As you can see from Table 3.3, `double`s provide more precision at the cost of using more memory.

Listing 3.8 (pi-print.cpp) prints an approximation of the mathematical value π.

Type	Storage	Smallest Magnitude	Largest Magnitude	Minimum Precision
`float`	4 bytes	1.17549×10^{-38}	$3.40282 \times 10^{+38}$	6 digits
`double`	8 bytes	2.22507×10^{-308}	$1.79769 \times 10^{+308}$	15 digits
`long double`	8 bytes	2.22507×10^{-308}	$1.79769 \times 10^{+308}$	15 digits

Table 3.3: Characteristics of Floating-point Numbers on 32-bit Computer Systems

Listing 3.8: `pi-print.cpp`

```
#include <iostream>

int main() {
    double pi = 3.14159;
    std::cout << "Pi = " << pi << '\n';
    std::cout << "or " << 3.14 << " for short" << '\n';
}
```

The first line in Listing 3.8 (pi-print.cpp) declares a variable named `pi` and assigns it a value. The second line in Listing 3.8 (pi-print.cpp) prints the value of the variable `pi`, and the third line prints a literal value. Any literal numeric value with a decimal point in a C++ program automatically has the type `double`, so

```
3.14
```

has type `double`. To make a literal floating-point value a `float`, you must append an `f` or `F` to the number, as in

```
3.14f
```

(The `f` or `F` suffix is used with literal values only; you cannot change a `double` variable into a `float` variable by appending an `f`. Attempting to do so would change the name of the variable!)

All floating-point literals are `double` values unless an `f` or `F` is appended to the end of the number; for example, `2.0` is a `double` literal, while `2.0f` is a `float` literal.

Floating-point numbers are an approximation of mathematical real numbers. As in the case of the `int` data type, the range of floating-point numbers is limited, since each value requires a fixed amount of memory. In some ways, though, `int`s are very different from `double`s. Any integer within the range of the `int` data type can be represented exactly. This is not true for the floating-point types. Consider the real number π. Since π contains an infinite number of digits, a floating-point number with finite precision can only approximate its value. Since the number of digits available is limited, even numbers with a finite number of digits have no exact representation; for example, the number 23.3123400654033989 contains too many digits for the `double` type and must be approximated as 23.3023498654034. Section 4.8.2 contains more information about the consequences of the inexact nature of floating-point numbers.

We can express floating-point numbers in scientific notation. Since most programming editors do not provide superscripting and special symbols like $\times$, C++ slightly alters the normal scientific notation. The number 6.022×10^{23} is written `6.022e23`. The number to the left of the `e` (we can use capital `E` as well)

is the mantissa, and the number to the right of the `e` is the exponent of 10. As another example, -5.1×10^4 is expressed in C++ as `-5.1e-4`. Listing 3.9 (scientificnotation.cpp) prints some scientific constants using scientific notation.

Listing 3.9: `scientificnotation.cpp`

```
#include <iostream>

int main() {
    double avogadros_number = 6.022e23, c = 2.998e8;
    std::cout << "Avogadro's number = " << avogadros_number << '\n';
    std::cout << "Speed of light = " << c << '\n';
}
```

Section 4.8.2 provides some insight into the implementation of C++ floating-point values and explains how internally all floating-point numbers are stored in exponential notation with a mantissa and exponent.

3.6 Constants

In Listing 3.9 (scientificnotation.cpp), Avogadro's number and the speed of light are scientific constants; that is, to the degree of precision to which they have been measured and/or calculated, they do not vary. C++ supports named constants. Constants are declared like variables with the addition of the `const` keyword:

```
const double PI = 3.14159;
```

Once declared and initialized, a constant can be used like a variable in all but one way—a constant may not be reassigned. It is illegal for a constant to appear on the left side of the assignment operator (=) outside its declaration statement. A subsequent statement like

```
PI = 2.5;
```

would cause the compiler to issue an error message:

error C3892: 'PI' : you cannot assign to a variable that is const

and fail to compile the program. Since the scientific constants do not change, Listing 3.10 (const.cpp) is a better version of Listing 3.9 (scientificnotation.cpp).

Listing 3.10: `const.cpp`

```
#include <iostream>

int main() {
    const double avogadros_number = 6.022e23, c = 2.998e8;
    std::cout << "Avogadro's number = " << avogadros_number << '\n';
    std::cout << "Speed of light = " << c << '\n';
}
```

Since it is illegal to assign a constant outside of its declaration statement, all constants **must** initialized where they are declared.

By convention, C++ programmers generally express constant names in all capital letters; in this way, within the source code a human reader can distinguish a constant quickly from a variable.

3.7 Other Numeric Types

C++ supports several other numeric data types:

- `long int`—typically provides integers with a greater range than the `int` type; its abbreviated name is `long`. It is guaranteed to provide a range of integer values at least as large as the `int` type.

  An integer literal with a `L` suffix, as in `19L`, has type `long`. A lower case elle (`l`) is allowed as a suffix as well, but you should not use it because it is difficult for human readers to distinguish between `l` (lower case *elle*) and `1` (digit *one*). (The `L` suffix is used with literal values only; you cannot change an `int` variable into a `long` by appending an `L`. Attempting to do so would change the name of the variable!)

- `short int`—typically provides integers with a smaller range than the `int` type; its abbreviated name is `short`. It is guaranteed that the range of `int`s is at least as big as the range of `short`s.

- `unsigned int`—is restricted to nonnegative integers; its abbreviated name is `unsigned`. While the `unsigned` type is limited in nonnegative values, it can represent twice as many positive values as the `int` type. (The name `int` is actually the short name for `signed int` and `int` can be written as `signed`.)

- `long double`—can extend the range and precision of the `double` type.

While the C++ language standard specifies minimum ranges and precision for all the numeric data types, a particular C++ compiler may exceed the specified minimums.

C++ provides such a variety of numeric types for specialized purposes usually related to building highly efficient programs. We will have little need to use many of these types. Our examples will use mainly the numeric types `int` for integers, `double` for an approximation of real numbers, and, less frequently, `unsigned` when nonnegative integral values are needed.

3.8 Characters

The `char` data type is used to represent single characters: letters of the alphabet (both upper and lower case), digits, punctuation, and control characters (like newline and tab characters). Most systems support the American Standard Code for Information Interchange (ASCII) character set. Standard ASCII can represent 128 different characters. Table 3.4 lists the ASCII codes for various characters.

In C++ source code, characters are enclosed by single quotes (`'`), as in

```
char ch = 'A';
```

Standard (double) quotes (`"`) are reserved for strings, which are composed of characters, but strings and `char`s are very different. C++ strings are covered in Section 11.2.6. The following statement would produce a compiler error message:

```
ch = "A";
```

since a string cannot be assigned to a character variable.

Internally, `char`s are stored as integer values, and C++ permits assigning numeric values to `char` variables and assigning characters to numeric variables. The statement

0	*null*	16	32	*space*	48	0	64	@	80	P	96	`	112	p
1		17	33	!	49	1	65	A	81	Q	97	a	113	q
2		18	34	"	50	2	66	B	82	R	98	b	114	r
3		19	35	#	51	3	67	C	83	S	99	c	115	s
4		20	36	$	52	4	68	D	84	T	100	d	116	t
5		21	37	%	53	5	69	E	85	U	101	e	117	u
6		22	38	&	54	6	70	F	86	V	102	f	118	v
7	*bell*	23	39	'	55	7	71	G	87	W	103	g	119	w
8	*backspace*	24	40	(	56	8	72	H	88	X	104	h	120	x
9	*tab*	25	41	)	57	9	73	I	89	Y	105	i	121	y
10	*newline*	26	42	*	58	:	74	J	90	Z	106	j	122	z
11		27	43	+	59	;	75	K	91	[	107	k	123	{
12	*form feed*	28	44	,	60	<	76	L	92	\	108	l	124	|
13	*return*	29	45	-	61	=	77	M	93	]	109	m	125	}
14		30	46	.	62	>	78	N	94	^	110	n	126	~
15		31	47	/	63	?	79	O	95	_	111	o	127	

Table 3.4: ASCII codes for characters

```
ch = 65;
```

assigns a number to a `char` variable to show that this perfectly legal. The value 65 is the ASCII code for the character *A*. If `ch` is printed, as in

```
ch = 65;
std::cout << ch;
```

the corresponding character, *A*, would be printed because `ch`'s declared type is `char`, not `int` or some other numeric type.

Listing 3.11 (charexample.cpp) shows how characters can be used within a program.

Listing 3.11: `charexample.cpp`

```
#include <iostream>

int main() {
    char ch1, ch2;
    ch1 = 65;
    ch2 = 'A';
    std::cout << ch1 << ", " << ch2 << ", " << 'A' << '\n';
}
```

The program displays

```
A, A, A
```

The first *A* is printed because the statement

```
ch1 = 65;
```

assigns the ASCII code for *A* to `ch1`. The second *A* is printed because the statement

```
ch2 = 'A';
```

assigns the literal character *A* to `ch2`. The third *A* is printed because the literal character `'A'` is sent directly to the output stream.

Integers and characters can be freely assigned to each other, but the range of `char`s is much smaller than the range of `int`s, so care must be taken when assigning an `int` value to a `char` variable.

Some characters are *non-printable* characters. The ASCII chart lists several common non-printable characters:

- `'\n'`—the newline character
- `'\r'`—the carriage return character
- `'\b'`—the backspace character
- `'\a'`—the "alert" character (causes a "beep" sound or other tone on some systems)
- `'\t'`—the tab character
- `'\f'`—the formfeed character
- `'\0'`—the *null* character (used in C strings, see Section 11.2.6)

These special non-printable characters begin with a backslash (`\`) symbol. The backslash is called an *escape* symbol, and it signifies that the symbol that follows has a special meaning and should not be interpreted literally. This means the literal backslash character must be represented as two backslashes: `'\\'`.

These special non-printable character codes can be embedded within strings. To embed a backslash within a string, you must escape it; for example, the statement

```
std::cout << "C:\\Dev\\cppcode" << '\n';
```

would print

```
C:\Dev\cppcode
```

See what this statement prints:

```
std::cout << "AB\bCD\aEF" << '\n';
```

The following two statements behave identically:

```
std::cout << "End of line" << '\n';
std::cout << "End of line\n";
```

On the Microsoft Windows platform, the character sequence `"\r\n"` (carriage return, line feed) appears at the end of lines in text files. Under Unix and Linux, lines in text files end with `'\n'` (line feed). On Apple Macintosh systems, text file lines end with the `'\r'` (carriage return) character. The compilers that adhere to the C++ standard will ensure that the `'\n'` character in a C++ program when sent to the output stream will produce the correct end-of-line character sequence for the given platform.

3.9 Enumerated Types

C++ allows a programmer to create a new, very simple type and list all the possible values of that type. Such a type is called an *enumerated type*, or an *enumeration type*. The `enum` keyword introduces an enumerated type. The following shows the simplest way to define an enumerated type:

```
enum Color { Red, Orange, Yellow, Green, Blue, Violet };
```

Here, the new type is named `Color`, and a variable of type `Color` may assume one of the values that appears in the list of values within the curly braces. The semicolon following the close curly brace is required. Sometimes the enumerated type definition is formatted as

```
enum Color {
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Violet
};
```

but the formatting makes no different to the compiler.

The values listed with the curly braces constitute *all* the values that a variable of the enumerated type can attain. The name of each value of an enumerated type must be a valid C++ identifier (see Section 3.3).

Given the `Color` type defined as above, we can declare and use variables of the `enum` type as shown by the following code fragment:

```
Color myColor;
myColor = Orange;
```

Here the variable `myColor` has our custom type `Color`, and its value is `Orange`.

When declaring enumerated types in this manner it is illegal to reuse an enumerated value name within another enumerated type within the same program. In the following code, the enumerated value `Light` appears in both the `Shade` type and `Weight` type:

```
enum Shade { Dark, Dim, Light, Bright };
enum Weight { Light, Medium, Heavy };
```

These two enumerated types are incompatible because they share the value `Light`, and so the compiler will issue an error.

This style of enumerated type definition is known as an *unscoped enumeration*. C++ inherits this unscoped enumeration style from the C programming language. The C++ standards committee introduced relatively recently an enhanced way of defining enumerated types known as *scoped enumerations*, also known as *enumeration classes*. Scoped enumerations solve the problem of duplicate enumeration values in different types. The following definitions are legal within the same program:

```
enum class Shade { Dark, Dim, Light, Bright };
enum class Weight { Light, Medium, Heavy };
```

When referring to a value from a scoped enumeration we must prepend the name of its type (class), as in


```
Shade color = Shade::Light;
Weight mass = Weight::Light;
```

In this case `Shade` and `Weight` are the scoped enumeration types defined above. Prefixing the type name to the value with the `::` operator enables the compiler to distinguish between the two different values. Scoped enumerations require the type name prefix even if the program contains no other enumerated types. In modern C++ development, scoped enumerations are preferable to unscoped enumerations. You should be familiar with unscoped enumerations, though, as a lot of published C++ code and older C++ books use unscoped enumerations.

Whether scoped or unscoped, the value names within an `enum` type must be unique. The convention in C++ is to captialize the first letter of an `enum` type and its associated values, although the language does not enforce this convention.

An `enum` type is handy for representing a small number of discrete, non-numeric options. For example, consider a program that controls the movements made by a small robot. The allowed orientations are forward, backward, left, right, up, and down. The program could encode these movements as integers, where 0 means left, 1 means backward, etc. While that implementation will work, it is not ideal. Integers may assume many more values than just the six values expected. The compiler cannot ensure that an integer variable representing a robot move will stay in the range 0...5. What if the programmer makes a mistake and under certain rare circumstances assigns a value outside of the range 0...5? The program then will contain an error that may result in erratic behavior by the robot. With `enum` types, if the programmer uses only the named values of the `enum` type, the compiler will ensure that such a mistake cannot happen.

A particular enumerated type necessarily has far fewer values than a type such as `int`. Imagine making an integer `enum` type and having to list all of its values! (The standard 32-bit `int` type represents over four billion values.) Enumerated types, therefore, are practical only for types that have a relatively small range of values.

3.10 Type Inference with auto

C++ requires that a variable be declared before it is used. Ordinarily this means specifying the variable's type, as in

```
int count;
char ch;
double limit;
```

A variable may be initialized when it is declared:

```
int count = 0;
char ch = 'Z';
double limit = 100.0;
```

Each of the values has a type: `0` is an `int`, `'Z'` is a `char`, and `0.0` is a `double`. The `auto` keyword allows the compiler to automatically deduce the type of a variable if it is initialized when it is declared:

```
auto count = 0;
auto ch = 'Z';
auto limit = 100.0;
```

The `auto` keyword may **not** be used without an accompanying initialization; for example, the following declaration is illegal:

```
auto x;
```

because the compiler cannot deduce x's type.

Automatic type inference is supported only by compilers that comply with the latest C++11 standard. Programmers using older compilers must specify a variable's exact type during the variable's declaration.

Automatic type deduction with `auto` is not useful to beginning C++ programmers. It is just as easy to specify the variable's type. The value of `auto` will become clearer when we consider some of the more advanced features of C++ (see Section 20.2).

3.11 Exercises

1. Will the following lines of code print the same thing? Explain why or why not.

```
std::cout << 6 << '\n';
std::cout << "6" << '\n';
```

2. Will the following lines of code print the same thing? Explain why or why not.

```
std::cout << x << '\n';
std::cout << "x" << '\n';
```

3. What is the largest `int` available on your system?
4. What is the smallest `int` available on your system?
5. What is the largest `double` available on your system?
6. What is the smallest `double` available on your system?
7. What C++ data type represents nonnegative integers?
8. What happens if you attempt to use a variable within a program, and that variable is not declared?
9. What is wrong with the following statement that attempts to assign the value ten to variable x?

```
10 = x;
```

10. Once a variable has been properly declared and initialized can its value be changed?
11. What is another way to write the following declaration and initialization?

```
int x = 10;
```

12. In C++ can you declare more than variable in the same declaration statement? If so, how?
13. In the declaration

```
int a;
int b;
```

do a and b represent the same memory location?

14. Classify each of the following as either a *legal* or *illegal* C++ identifier:

    (a) `fred`
    (b) `if`
    (c) `2x`
    (d) `-4`
    (e) `sum_total`
    (f) `sumTotal`
    (g) `sum-total`
    (h) `sum total`
    (i) `sumtotal`
    (j) `While`
    (k) `x2`
    (l) `Private`
    (m) `public`
    (n) `$16`
    (o) `xTwo`
    (p) `_static`
    (q) `_4`
    (r) `___`
    (s) `10%`
    (t) `a27834`
    (u) `wilma's`

15. What can you do if a variable name you would like to use is the same as a reserved word?

16. Why does C++ require programmers to declare a variable before using it? What are the advantages of declaring variables?

17. What is the difference between `float` and `double`?

18. How can a programmer force a floating-point literal to be a `float` instead of a `double`?

19. How is the value 2.45×10^{-5} expressed as a C++ literal?

20. How can you ensure that a variable's value can never be changed after its initialization?

21. How can you extend the range of `int` on some systems?

22. How can you extend the range and precision of `double` on some systems?

23. Write a program that prints the ASCII chart for all the values from 0 to 127.

24. Is `"i"` a string literal or character literal?

25. Is `'i'` a string literal or character literal?

26. Is it legal to assign a `char` value to an `int` variable?

27. Is it legal to assign an `int` value to a `char` variable?

28. What is printed by the following code fragment?

```
int x;
x = 'A';
std::cout << x << '\n';
```

29. What is the difference between the character `'n'` and the character `'\n'`?

30. Write a C++ program that simply emits a beep sound when run.

31. Create an unscoped enumeration type that represents the days of the week.

32. Create a scoped enumeration type that represents the days of the week.

33. Create an unscoped enumeration type that represents the months of the year.

34. Create a scoped enumeration type that represents the months of the year.

35. Determine the exact type of each of the following variables:

    (a) `auto a = 5;`

    (b) `auto b = false;`

    (c) `auto c = 9.3;`

    (d) `auto d = 5.1f;`

    (e) `auto e = 5L;`

Chapter 4

Expressions and Arithmetic

This chapter uses the C++ numeric types introduced in Chapter 3 to build expressions and perform arithmetic. Some other important concepts are covered—user input, source formatting, comments, and dealing with errors.

4.1 Expressions

A literal value like 34 and a properly declared variable like x are examples of simple *expressions*. We can use operators to combine values and variables and form more complex expressions. Listing 4.1 (adder.cpp) shows how the addition operator (+) is used to add two integers.

Listing 4.1: `adder.cpp`

```
#include <iostream>

int main() {
    int value1, value2, sum;
    std::cout << "Please enter two integer values: ";
    std::cin >> value1 >> value2;
    sum = value1 + value2;
    std::cout << value1 << " + " << value2 << " = " << sum << '\n';
}
```

In Listing 4.1 (adder.cpp):

- `int value1, value2, sum;`

  This statement declares three integer variables, but it does not initialize them. As we examine the rest of the program we will see that it would be superfluous to assign values to the variables here.

- `std::cout << "Please enter two integer values: ";`

  This statement prompts the user to enter some information. This statement is our usual print statement, but it is not terminated with the end-of-line marker `'\n'`. This is because we want the cursor to remain at the end of the printed line so when the user types in values they appear on the same line as the message prompting for the values. When the user presses the enter key to complete the input, the cursor will automatically move down to the next line.

- `std::cin >> value1 >> value2;`

  This statement causes the program's execution to stop until the user types two numbers on the keyboard and then presses enter. The first number entered will be assigned to `value1`, and the second number entered will be assigned to `value2`. Once the user presses the enter key, the value entered is assigned to the variable. The user may choose to type one number, press enter, type the second number, and press enter again. Instead, the user may enter both numbers separated by one of more spaces and then press enter only once. The program will not proceed until the user enters two numbers.

The `std::cin` input stream object can assign values to multiple variables in one statement, as shown here:

```
int num1, num2, num3;
std::cin >> num1 >> num2 >> num3;
```

A common beginner's mistake is use commas to separate the variables, as in

```
int num1, num2, num3;
std::cin >> num1, num2, num3;
```

The compiler will not generate an error message, because it is legal C++ code. The statement, however, will not assign the three variables from user input as desired. The comma operator in C++ has different meanings in different contexts, and here it is treated like a statement separator; thus, the variables `num2` and `num3` are not involved with the `std::cin` input stream object. We will have no need to use the comma operator in this way, but you should be aware of this potential pitfall.

`std::cin` is a object that can be used to read input from the user. The `>>` operator—as used here in the context of the `std::cin` object—is known as the *extraction operator*. Notice that it is "backwards" from the `<<` operator used with the `std::cout` object. The `std::cin` object represents the input stream—information flowing into the program from user input from the keyboard. The `>>` operator extracts the data from the input stream `std::cin` and assigns the pieces of the data, in order, to the various variables on its right.

- `sum = value1 + value2;`

  This is an assignment statement because it contains the assignment operator (`=`). The variable `sum` appears to the left of the assignment operator, so `sum` will receive a value when this statement executes. To the right of the assignment operator is an arithmetic expression involving two variables and the addition operator. The expression is *evaluated* by adding together the values of the two variables. Once the expression's value has been determined, that value can be assigned to the `sum` variable.

All expressions have a value. The process of determining the expression's value is called *evaluation*. Evaluating simple expressions is easy. The literal value 54 evaluates to 54. The value of a variable named `x` is the value stored in the memory location reserved for `x`. The value of a more complex expression is found by evaluating the smaller expressions that make it up and combining them with operators to form potentially new values.

Table 4.1 lists the main C++ arithmetic operators. Table 4.1. The common arithmetic operations, addition, subtraction, and multiplication, behave in the expected way. All these operators are classified as *binary* operators because they operate on two operands. In the statement

```
x = y + z;
```

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	modulus

Table 4.1: The simple C++ arithmetic operators

the right side is an addition expression `y + z`. The two operands of the + operator are `y` and `z`.

Two of the operators above, + and -, serve also as *unary* operators. A unary operator has only one operand. The - unary operator expects a single numeric expression (literal number, variable, or complex numeric expression within parentheses) immediately to its right; it computes the *additive inverse* of its operand. If the operand is positive (greater than zero), the result is a negative value of the same magnitude; if the operand is negative (less than zero), the result is a positive value of the same magnitude. Zero is unaffected. For example, the following code sequence

```
int x = 3;
int y = -4;
int z = 0;
std::cout << -x << " " << -y << " " -z << '\n';
```

within a program would print

```
-3 4 0
```

The following statement

```
std::cout << -(4 - 5) << '\n';
```

within a program would print

```
1
```

The unary + operator is present only for completeness; when applied to a numeric value, variable, or expression, the resulting value is no different from the original value of its operand. Omitting the unary + operator from the following statement

```
x = +y;
```

does not change the statement's behavior.

All the arithmetic operators are subject to the limitations of the data types on which they operate; for example, on a system in which the largest `int` is 2,147,483,647, the expression

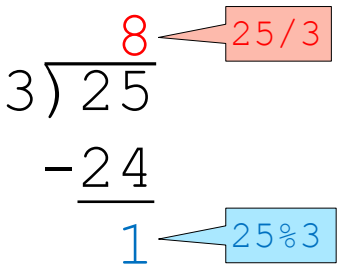
```
2147483647 + 1
```

will not evaluate to the correct answer since the correct answer falls outside the range of `int`s.

If you add, subtract, multiply, is divide two `int`s, the result is an integer. As long as the operation does not exceed the range of `int`s, the arithmetic works as expected. Division, however, is another matter. The statement

```
std::cout << 10/3 << " " << 3/10 << '\n';
```


Figure 4.1 Integer division vs. integer modulus. Integer division produces the quotient, and modulus produces the remainder. In this example, `25/3` is 8, and `25%3` is 1.



prints

```
3 0
```

because in the first case 10 divided by 3 is 3 with a remainder of 1, and in the second case 3 divided by 10 is 0 with a remainder of 3. Since integers are whole numbers, any fractional part of the answer must be discarded. The process of discarding the fractional part leaving only the whole number part is called *truncation*. 10 divided by 3 should be 3.3333..., but that value is truncated to 3. Truncation is not rounding; for example, 11 divided by 3 is 3.6666..., but it also truncates to 3.

Truncation simply removes any fractional part of the value. It does not round. Both 10.01 and 10.999 truncate to 10.

The modulus operator (%) computes the remainder of integer division; thus,

```
std::cout << 10%3 << " " << 3%10 << '\n';
```

prints

```
1 3
```

since 10 divided by 3 is 3 with a remainder of 1, and 3 divided by 10 is 0 with a remainder of 3. Figure 4.1 uses long division for a more hands on illustration of how the integer division and modulus operators work.

The modulus operator is more useful than it may first appear. Listing 4.11 (timeconv.cpp) shows how we can use it to convert a given number of seconds to hours, minutes, and seconds.

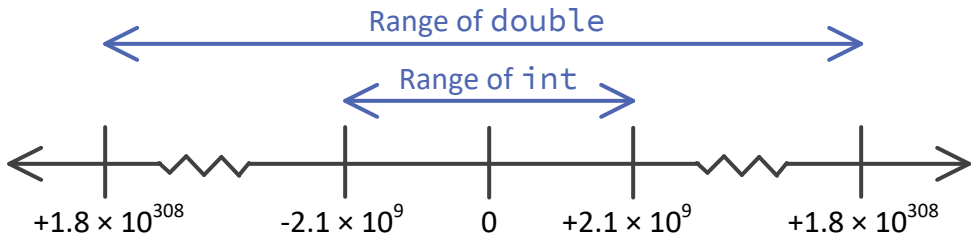
In contrast to integer arithmetic, floating-point arithmetic with `doubles` behaves as expected:

```
std::cout << 10.0/3.0 << " " << 3.0/10.0 << '\n';
```

prints

```
3.33333 0.3
```

Figure 4.2 Range of ints vs. range of doubles



Since a `char` is stored internally as a number (see Section 3.8), we can perform arithmetic on characters. We will have little need to apply mathematics to characters, but sometimes it is useful. As an example, the lower-case letters of the alphabet a–z occupy ASCII values 97–123, with a = 97, b = 98, etc. The upper-case letters A–Z are coded as 65–91, with A = 65, B = 66, etc. To capitalize any lower-case letter, you need only subtract 32, as in

```
char lower = 'd', upper = lower - 32;
std::cout << upper << '\n';
```

This section of code would print D. If you do not remember the offset of 32 between upper- and lower-case letter, you can compute it with the letters themselves:

```
upper = lower - ('a' - 'A');
```

In this case, if `lower` has been assigned any value in the range `'a'` to `'z'`, the statement will assign to `upper` the capitalized version of `lower`. On the other hand, if `lower`'s value is outside of that range, `upper` will not receive a meaningful value.

4.2 Mixed Type Expressions

Expressions may contain mixed elements; for example, the following program fragment

```
int x = 4;
double y = 10.2, sum;
sum = x + y;
```

adds an `int` to a `double`, and the result is being assigned to a `double`. How is the arithmetic performed?

As shown in Figure 4.2, the range of `int`s falls completely within the range of `double`s; thus, any `int` value can represented by a `double`. The `int` 4 also can be expressed as the `double` 4.0. In fact, since the largest `int` on most systems is 2,147,483,647, the minimum 15 digits of `double` precision are more than adequate to represent all integers exactly. This means that any `int` value can be represented by a `double`. The converse is not true, however. 2,200,000,000 can be represented by a `double` but it is too big for the `int` type. We say that the `double` type is *wider* than the `int` type and that the `int` type is *narrower* than the `double` type.

It would be reasonable, then, to be able to assign `int` values to `double` variables. The process is called *widening*, and it is always safe to widen an `int` to a `double`. The following code fragment

```
double d1;
int i1 = 500;
d1 = i1;
std::cout << "d1 = " << d1 << '\n';
```

is legal C++ code, and when part of a complete program it would display

```
d1 = 500
```

Assigning a `double` to an `int` variable is not always possible, however, since the `double` value may not be in the range of `int`s. Furthermore, if the `double` variable falls within the range of `int`s but is not a whole number, the `int` variable is unable to the manage fractional part. Consider the following code fragment:

```
double d = 1.6;
int i = d;
```

The second line assigns 1 to `i`. Truncation loses the 0.6 fractional part (see Section 4.1). Note that proper rounding is not done. The Visual C++ compiler will warn us of a potential problem:

warning C4244: '=' : conversion from 'double' to 'int', possible loss of data

This warning reminds us that some information may be lost in the assignment. While the compiler and linker will generate an executable program when warnings are present, you should carefully scrutinize all warnings. This warning is particularly useful, since it is easy for errors due to the truncation of floating-point numbers to creep into calculations.

Converting from a wider type to a narrower type (like `double` to `int`) is called *narrowing*. It often is necessary to assign a floating-point value to an integer variable. If we know the value to assign is within the range of `int`s, and the value has no fractional parts or its truncation would do no harm, the assignment is safe. To perform the assignment without a warning from the compiler, we use a procedure called a *cast*, also called a *type cast*. The cast forces the compiler to accept the assignment without a issuing a warning. The following statement convinces the compiler to accept the `double`-to-`int` assignment without a warning:

```
i = static_cast<int>(d);
```

The reserved word `static_cast` performs the narrowing conversion and silences the compiler warning. The item to convert (in this case the variable `d`) is placed in the parentheses, and the desired type (in this case the type `int`) appears in the angle brackets. The statement

```
i = static_cast<int>(d);
```

does not change the type of the variable `d`; `d` is declared to be a `double` and so must remain a `double` variable. The statement makes a copy of `d`'s value in a temporary memory location, converting it to its integer representation during the process.

We also can cast literal values and expressions:

```
i = static_cast<int>(1.6);
i = static_cast<int>(x + 2.1);
```


Narrowing a floating-point value to an integer discards any fractional part. Narrowing truncates; it does not round. For example, the `double` value 1.7 narrows to the `int` value 1.

The widening conversion is always safe, so a type cast is not required. Narrowing is a potentially dangerous operation, and using an explicit cast does not remove the danger—it simply silences the compiler. For example, consider Listing 4.2 (badnarrow.cpp).

Listing 4.2: `badnarrow.cpp`

```
#include <iostream>

int main() {
    double d = 2200000000.0;
    int i = d;
    std::cout << "d = " << d << ", i = " << i << '\n';
}
```

The Visual C++ compiler issues a warning about the possible loss of precision when assigning `d` to `i`. Silencing the warning with a type cast in this case is a bad idea; the program's output indicates that the warning should be heeded:

```
d = 2.2e+009, i = -2147483648
```

The printed values of `i` and `d` are not even close, nor can they be because it is impossible to represent the value 2,200,000,000 as an `int` on a system that uses 32-bit integers. When assigning a value of a wider type to a variable of a narrower type, the programmer must assume the responsibility to ensure that the actual value to be narrowed is indeed within the range of the narrower type. The compiler cannot ensure the safety of the assignment.

Casts should be used sparingly and with great care because a cast creates a spot in the program that is immune to the compiler's type checking. A careless assignment can produce a garbage result introducing an error into the program.

When we must perform mixed arithmetic—such as adding an `int` to a `double`—the compiler automatically produces machine language code that copies the `int` value to a temporary memory location and transforms it into its `double` equivalent. It then performs double-precision floating-point arithmetic to compute the result.

Integer arithmetic occurs only when both operands are `int`s. `1/3` thus evaluates to 0, but `1.0/3.0`, `1/3.0`, and `1.0/3` all evaluate to 0.33333.

Since `double` is wider than `int`, we say that `double` *dominates* `int`. In a mixed type arithmetic expression, the less dominant type is coerced into the more dominant type in order to perform the arithmetic operation.

Section 3.9 introduced enumerated types. Behind the scenes, the compiler translates enumerated values into integers. The first value in the enumeration is 0, the second value is 1, etc. Even though the underlying implementation of enumerated types is integer, the compiler does not allow the free exchange between integers and enumerated types. The following code will not compile:

```
enum class Color { Red, Orange, Yellow, Green, Blue, Violet };
std::cout << Color::Orange << " " << Color::Green << '\n';
```

The `std::cout` printing object knows how to print integers, but it does not know anything about our `Color` class and its values. If we really want to treat an enumerated type value as its underlying integer, we must use a type cast. Listing 4.3 (enumcast.cpp) shows how to extract the underlying integer value from an enumerated type.

Listing 4.3: `enumcast.cpp`

```
#include <iostream>

int main() {
    enum class Color { Red, Orange, Yellow, Green, Blue, Violet };
    std::cout << static_cast<int>(Color::Orange) << " "
              << static_cast<int>(Color::Green) << '\n';
}
```

Listing 4.3 (enumcast.cpp) prints prints

```
1 3
```

This is the expected output because `Color::Red` is 0, `Color::Orange` is 1, `Color::Yellow` is 2, `Color::Green` is 3, etc.

Even though enumerated types are encoded as integers internally, programmers may not perform arithmetic on enumerated types without involving casts. Such opportunities should be very rare; if you need to perform arithmetic on a variable, it really should be a numerical type, not an enumerated type.

4.3 Operator Precedence and Associativity

When different operators are used in the same expression, the normal rules of arithmetic apply. All C++ operators have a *precedence* and *associativity*:

- **Precedence**—when an expression contains two different kinds of operators, which should be applied first?
- **Associativity**—when an expression contains two operators with the same precedence, which should be applied first?

To see how precedence works, consider the expression

```
2 + 3 * 4
```

Should it be interpreted as

```
(2 + 3) * 4
```

(that is, 20), or rather is

```
2 + (3 * 4)
```

(that is, 14) the correct interpretation? As in normal arithmetic, in C++ multiplication and division have equal importance and are performed before addition and subtraction. We say multiplication and division have precedence over addition and subtraction. In the expression

```
2 + 3 * 4
```

the multiplication is performed before addition, since multiplication has precedence over addition. The result is 14. The multiplicative operators (`*`, `/`, and `%`) have equal precedence with each other, and the additive operators (binary `+` and `-`) have equal precedence with each other. The multiplicative operators have precedence over the additive operators.

As in standard arithmetic, in C++ if the addition is to be performed first, parentheses can override the precedence rules. The expression

```
(2 + 3) * 4
```

evaluates to 20. Multiple sets of parentheses can be arranged and nested in any ways that are acceptable in standard arithmetic.

To see how associativity works, consider the expression

```
2 - 3 - 4
```

The two operators are the same, so they have equal precedence. Should the first subtraction operator be applied before the second, as in

```
(2 - 3) - 4
```

(that is, -5), or rather is

```
2 - (3 - 4)
```

(that is, 3) the correct interpretation? The former (-5) is the correct interpretation. We say that the subtraction operator is *left associative*, and the evaluation is left to right. This interpretation agrees with standard arithmetic rules. All binary operators except assignment are left associative. Assignment is an exception; it is *right associative*. To see why associativity is an issue with assignment, consider the statement

```
w = x = y = z;
```

This is legal C++ and is called *chained assignment*. Assignment can be used as both a statement and an expression. The *statement*

```
x = 2;
```

assigns the value 2 to the variable `x`. The *expression*

```
x = 2
```

assigns the value 2 to the variable `x` and evaluates to the value that was assigned; that is, 2. Since assignment is right associative, the chained assignment example should be interpreted as

```
w = (x = (y = z));
```

which behaves as follows:

- The expression `y = z` is evaluated first. `z`'s value is assigned to `y`, and the value of the expression `y = z` is `z`'s value.

Arity	Operators	Associativity
Unary	+, −	
Binary	*, /, %	Left
Binary	+, −	Left
Binary	=	Right

Table 4.2: Operator precedence and associativity. The operators in each row have a higher precedence than the operators below it. Operators within a row have the same precedence.

- The expression `x = (y = z)` is evaluated. The value of `y = z`, that is `z`, is assigned to `x`. The overall value of the expression `x = y = z` is thus the value of `z`. Now the values of `x`, `y`, and `z` are all equal (to `z`).
- The expression `w = (x = y = z)` is evaluated. The value of the expression `x = y = z` is equal to `z`'s value, so `z`'s value is assigned to `w`. The overall value of the expression `w = x = y = z` is equal to `z`, and the variables `w`, `x`, `y`, and `z` are all equal (to `z`).

As in the case of precedence, we can use parentheses to override the natural associativity within an expression.

The unary operators have a higher precedence than the binary operators, and the unary operators are right associative. This means the statements

```
std::cout << -3 + 2 << '\n';
std::cout << -(3 + 2) << '\n';
```

which display

```
-1
-5
```

behave as expected.

Table 4.2 shows the precedence and associativity rules for some C++ operators. The * operator also has a unary form that has nothing to do with mathematics; it is covered in Section 10.7.

4.4 Comments

Good programmers annotate their code by inserting remarks that explain the purpose of a section of code or why they chose to write a section of code the way they did. These notes are meant for human readers, not the compiler. It is common in industry for programs to be reviewed for correctness by other programmers or technical managers. Well-chosen identifiers (see Section 3.3) and comments can aid this assessment process. Also, in practice, teams of programmers develop software. A different programmer may be required to finish or fix a part of the program written by someone else. Well-written comments can help others understand new code quicker and increase their productivity modifying old or unfinished code. While it may seem difficult to believe, even the same programmer working on her own code months later can have a difficult time remembering what various parts do. Comments can help greatly.

Any text contained within comments is ignored by the compiler. C++ supports two types of comments: *single line comments* and *block comments*:

- **Single line comment**—the first type of comment is useful for writing a single line remark:

```
// Compute the average of the values
avg = sum / number;
```

The first line here is a comment that comment explains what the statement that follows it is supposed to do. The comment begins with the double forward slash symbols (`//`) and continues until the end of that line. The compiler will ignore the `//` symbols and the contents of the rest of the line. This type of comment is also useful for appending a short comment to the end of a statement:

```
avg = sum / number;  // Compute the average of the values
```

Here, an executable statement and the comment appear on the same line. The compiler will read the assignment statement here, but it will ignore the comment. The compiler generates the same machine code for this example as it does for the preceding example, but this example uses one line of source code instead of two.

- **Block comment**—the second type of comment begins with the symbols `/*` and is in effect until the `*/` symbols are encountered. The `/*. . . */` symbols delimit the comment like parentheses delimit a parenthetical expression. Unlike parentheses, however, these block comments cannot be nested within other block comments.

  The block comment is handy for multi-line comments:

```
/*  After the computation is completed
    the result is displayed.  */
std::cout << result << '\n';
```

What should be commented? Avoid making a remark about the obvious; for example:

```
result = 0;  // Assign the value zero to the variable named result
```

The effect of this statement is clear to anyone with even minimal C++ programming experience. Thus, the audience of the comments should be taken into account; generally, "routine" activities require no remarks. Even though the *effect* of the above statement is clear, its *purpose* may need a comment. For example:

```
result = 0;  // Ensures 'result' has a well-defined minimum value
```

This remark may be crucial for readers to completely understand how a particular part of a program works. In general, programmers are not prone to providing too many comments. When in doubt, add a remark. The extra time it takes to write good comments is well worth the effort.

4.5 Formatting

Program comments are helpful to human readers but ignored by the compiler. Another aspect of source code that is largely irrelevant to the compiler but that people find valuable is its formatting. Imagine the difficulty of reading a book in which its text has no indentation or spacing to separate one paragraph from another. In comparison to the source code for a computer program, a book's organization is quite simple. Over decades of software construction programmers have established a small collection of source code formatting styles that the industry finds acceptable.

The compiler allows a lot of leeway for source code formatting. Consider Listing 4.4 (reformattedvariable.cpp) which is a reformatted version of Listing 3.4 (variable.cpp).

Listing 4.4: reformattedvariable.cpp

```
#include <iostream>
int
main
(
)
{
int
x
;
x
=
10
;
std
::
cout
<<
x
<<
'\n'
;
}
```

Listing 4.5 (reformattedvariable2.cpp) is another reformatted version of Listing 3.4 (variable.cpp).

Listing 4.5: reformattedvariable2.cpp

```
#include <iostream>
int main(){int x;x=10;std::cout<<x<<'\n';}
```

Both reformatted programs are valid C++ and compile to the same machine language code as the original version. Most would argue that the original version is easier to read and understand more quickly than either of the reformatted versions. The elements in Listing 3.4 (variable.cpp) are organized better. Experienced C++ programmers would find both Listing 4.4 (reformattedvariable.cpp) and Listing 4.5 (reformattedvariable2.cpp) visually painful.

What are some distinguishing characteristics of Listing 3.4 (variable.cpp)?

- Each statement appears on its own line. A statement is not unnecessarily split between two lines of text. Visually, one line of text implies one action (statement) to perform.
- The close curly brace aligns vertically with the line above that contains the corresponding open curly brace. This makes it easier to determine if the curly braces match and nest properly. It also better portrays the logical structure of the program. The ability to accurately communicate the logical structure of a program becomes very important as write more complex programs. Programs with complex logic frequently use multiple nested curly braces (for example, see Listing 5.11 (troubleshoot.cpp)). Without a consistent, organized arrangement of curly braces it can difficult to determine which opening brace goes with a particular closing brace.
- The statements that constitute the body of `main` are indented several spaces. This visually emphasizes the fact that the elements are indeed logically enclosed. As with curly brace alignment, indentation to emphasize logical enclosure becomes more important as more complex programs are considered.

- Spaces are used to spread out statements and group pieces of the statement. Space around the operators (=) makes it easier to visually separate the operands from the operators and comprehend the details of the expression. Most people find the statement

  ```
  total_sale = subtotal + tax;
  ```

  much easier to read than

  ```
  total_sale=subtotal+tax;
  ```

  since the lack of space in the second version makes it more difficult to pick out the individual pieces of the statement. In the first version with extra space, it is clearer where operators and variable names begin and end.

  In a natural language like English, a book is divided into distinct chapters, and chapters are composed of paragraphs. One paragraph can be distinguished from another because the first line is indented or an extra space appears between two paragraphs. Space is used to separate words in each sentence. Consider how hard it would be to read a book if all the sentences were printed like this one:

  > Theboyranquicklytothetreetoseethestrandedcat.

  Judiciously placed open space in a C++ program can greatly enhance its readability.

C++ gives the programmer a large amount of freedom in formatting source code. The compiler reads the characters that make up the source code one symbol at a time left to right within a line before moving to the next line. While extra space helps readability, spaces are not allowed in some places:

- Variable names and reserved words must appear as unbroken units.
- Multi-symbol operators like `<<` cannot be separated (`< <` is illegal).

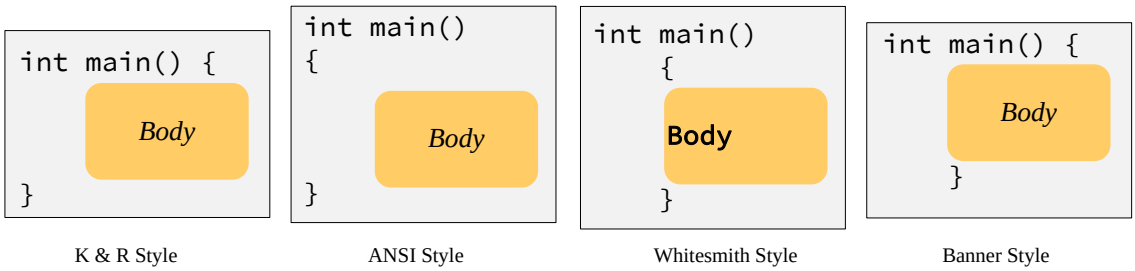
One common coding convention that is universal in C++ programming is demonstrated in Listing 3.10 (const.cpp). While programmers usually use lower-case letters in variable names, they usually express constant names with all capital letters; for example, `PI` is used for the mathematical constant π instead of `pi`. C++ does not require constants to be capitalized, but capitalizing them aids humans reading the source code so they can quickly distinguish between variables and constants.

Figure 4.3 shows the four most common ways programmers use indentation and place curly braces in C++ source code.

The K&R and ANSI styles are the most popular in published C++ source code. The Whitesmith and Banner styles appear much less frequently. `http://en.wikipedia.org/wiki/Indent_style` reviews the various ways to format C++ code. Observe that all the accepted formatting styles indent the block of statements contained in the `main` function.

Most software development organizations adopt a set of *style guidelines*, sometimes called *code conventions*. These guidelines dictate where to indent and by how many spaces, where to place curly braces, how to assign names to identifiers, etc. Programmers working for the organization are required to follow these style guidelines for the code they produce. This better enables any member of the development team to read and understand more quickly code written by someone else. This is necessary when code is reviewed for correctness or when code must be repaired or extended, and the original programmer is no longer with the development team.

Figure 4.3 The most common C++ coding styles.



K & R Style ANSI Style Whitesmith Style Banner Style

Even if you are not forced to use a particular style, it is important to use a consistent style throughout the code you write. As our programs become more complex we will need to use additional curly braces and various levels of indentation to organize the code we write. A consistent style (especially one of the standard styles shown in Figure 4.3) makes it easier to read and verify that the code actually expresses our intent. It also makes it easier to find and fix errors. Said another way, haphazard formatting increases the time it takes to develop correct software because programmer's mistakes hide better in poorly formatted code.

Good software development tools can boost programmer productivity, and many programming editors have the ability to automatically format source code according to a standard style. Some of these editors can correct the code's style as the programmer types in the text. A standalone program known as a *pretty printer* can transform an arbitrarily formatted C++ source file into a properly formatted one.

4.6 Errors and Warnings

Beginning programmers make mistakes writing programs because of inexperience in programming in general or because of unfamiliarity with a programming language. Seasoned programmers make mistakes due to carelessness or because the proposed solution to a problem is faulty and the correct implementation of an incorrect solution will not produce a correct program. Regardless of the reason, a programming error falls under one of three categories:

- compile-time error
- run-time error
- logic error

4.6.1 Compile-time Errors

A *compile-time error* results from the programmer's misuse of the language. A *syntax error* is a common compile-time error. For example, in English one can say

> The boy walks quickly.

This sentence uses correct syntax. However, the sentence

The boy walk quickly.

is not correct syntactically: the number of the subject (singular form) disagrees with the number of the verb (plural form). It contains a syntax error. It violates a grammatical rule of the English language. Similarly, the C++ statement

```
x = y + 2;
```

is syntactically correct because it obeys the rules for the structure of an assignment statement described in Section 3.2. However, consider replacing this assignment statement with a slightly modified version:

```
y + 2 = x;
```

If a statement like this one appears in a program and the variables `x` and `y` have been properly declared, the compiler will issue an error message; for example, the Visual C++ compiler reports (among other things):

error C2106: '=' : left operand must be l-value

The syntax of C++ does not allow an expression like `y + 2` to appear on the left side of the assignment operator.

(The term *l-value* in the error message refers to the left side of the assignment operator; the `l` is an "elle," not a "one.".)

The compiler may generate an error for a syntactically correct statement like

```
x = y + 2;
```

if either of the variables `x` or `y` has not been declared; for example, if `y` has not been declared, Visual C++ reports:

error C2065: 'y' : undeclared identifier

Other common compile-time errors include missing semicolons at the end of statements, mismatched curly braces and parentheses, and simple typographical errors.

Compile-time errors usually are the easiest to repair. The compiler pinpoints the exact location of the problem, and the error does not depend on the circumstances under which the program executes. The exact error can be reproduced by simply recompiling the same source code.

Compilers have the reputation for generating cryptic error messages. They seem to provide little help as far as novice programmers are concerned. Sometimes a combination of errors can lead to messages that indicate errors on lines that follow the line that contains the actual error. Once you encounter the same error several times and the compiler messages become more familiar, you become better able to deduce the actual problem from the reported message. Unfortunately C++ is such a complex language that sometimes a simple compile-time error can result in a message that is incomprehensible to beginning C++ programmers.

4.6.2 Run-time Errors

The compiler ensures that the structural rules of the C++ language are not violated. It can detect, for example, the malformed assignment statement and the use of a variable before its declaration. Some violations of the language cannot be detected at compile time, however. A program may not run to completion but instead terminate with an error. We commonly say the program "crashed." Consider Listing 4.6 (dividedanger.cpp) which under certain circumstances will crash.

Listing 4.6: `dividedanger.cpp`

```
// File dividedanger.cpp

#include <iostream>

int main() {
    int dividend, divisor;

    // Get two integers from the user
    std::cout << "Please enter two integers to divide:";
    std::cin >> dividend >> divisor;
    // Divide them and report the result
    std::cout << dividend << "/" << divisor << " = "
              << dividend/divisor << '\n';
}
```

The expression

```
dividend/divisor
```

is potentially dangerous. If the user enters, for example, 32 and 4, the program works nicely

```
Please enter two integers to divide: 32 4
32/4 = 8
```

and displays the answer of 8. If the user instead types the numbers 32 and 0, the program reports an error and terminates. Division by zero is undefined in mathematics, and integer division by zero in C++ is illegal. When the program attempts the division at run time, the system detects the attempt and terminates the program.

This particular program can fail in other ways as well; for example, outside of the C++ world, 32.0 looks like a respectable integer. If the user types in 32.0 and 8, however, the program crashes because 32.0 is not a valid way to represent an integer in C++. When the compiler compiles the source line

```
std::cin >> dividend >> divisor;
```

given that `dividend` has been declared to be an `int`, it generates slightly different machine language code than it would if `dividend` has been declared to be a `double` instead. The compiled code expects the text entered by the user to be digits with no extra decoration. Any deviation from this expectation results in a run-time error. Similar results occur if the user enters text that does not represent an integer, like *fred*.

Observe that in either case—entry of a valid but inappropriate integer (zero) or entry of a non-integer (32.0 or *fred*)—it is impossible for the compiler to check for these problems at compile time. The compiler cannot predict what the user will enter when the program is run. This means it is up to the programmer to write code that can handle bad input that the user may provide. As we continue our exploration of programming in C++, we will discover ways to make our programs more robust against user input (see Listing 5.2 (betterdivision.cpp) in Chapter 5, for example). The solution involves changing the way the program runs depending on the actual input provided by the user.

4.6.3 Logic Errors

Consider the effects of replacing the expression

```
dividend/divisor;
```

in Listing 4.6 (dividedanger.cpp) with the expression:

```
divisor/dividend;
```

The program compiles with no errors. It runs, and unless a value of zero is entered for the dividend, no run-time errors arise. However, the answer it computes is not correct in general. The only time the correct answer is printed is when `dividend = divisor`. The program contains an error, but neither the compiler nor the run-time system is able detect the problem. An error of this type is known as a *logic error*.

Listing 4.20 (faultytempconv.cpp) is an example of a program that contains a logic error. Listing 4.20 (faultytempconv.cpp) compiles and does not generate any run-time errors, but it produces incorrect results.

Beginning programmers tend to struggle early on with compile-time errors due to their unfamiliarity with the language. The compiler and its error messages are actually the programmer's best friend. As the programmer gains experience with the language and the programs written become more complicated, the number of compile-time errors decrease or are trivially fixed and the number of logic errors increase. Unfortunately, both the compiler and run-time environment are powerless to provide any insight into the nature and sometimes location of logic errors. Logic errors, therefore, tend to be the most difficult to find and repair. Tools such as debuggers are frequently used to help locate and fix logic errors, but these tools are far from automatic in their operation.

Errors that escape compiler detection (run-time errors and logic errors) are commonly called *bugs*. Since the compiler is unable to detect these problems, such bugs are the major source of frustration for developers. The frustration often arises because in complex programs the bugs sometimes only reveal themselves in certain situations that are difficult to reproduce exactly during testing. You will discover this frustration as your programs become more complicated. The good news is that programming experience and the disciplined application of good programming techniques can help reduce the number logic errors. The bad news is that since software development in an inherently human intellectual pursuit, logic errors are inevitable. Accidentally introducing and later finding and eliminating logic errors is an integral part of the programming process.

4.6.4 Compiler Warnings

A warning issued by the compiler does mark a violation of the rules in the C++ language, but it is an notification to the programmer that the program contains a construct that is a potential problem. In Listing 4.10 (tempconv.cpp) the programmer is attempting to print the value of a variable before it has been given a known value.

Listing 4.7: `uninitialized.cpp`

```
// uninitialized.cpp

#include <iostream>

int main() {
    int n;
    std::cout << n << '\n';
}
```

An attempt to build Listing 4.7 (uninitialized.cpp) yields the following message from the Visual C++ compiler:

warning C4700: uninitialized local variable 'n' used

The compiler issued a warning but still generated the executable file. When run, the program produces a random result because it prints the value in memory associated with the variable, but the program does not initialize that memory location.

Listing 4.8 (narrow.cpp) assigns a `double` value to an `int` variable, which we know from Section 4.1 truncates the result.

Listing 4.8: `narrow.cpp`

```
#include <iostream>

int main() {
    int n;
    double d = 1.6;
    n = d;
    std::cout << n << '\n';
}
```

When compiled we see

warning C4244: '=' : conversion from 'double' to 'int', possible loss of data

Since it is a warning and not an error, the compiler generates the executable, but the warning should prompt us to stop and reflect about the correctness of the code. The enhanced warning level prevents the programmer from being oblivious to the situation.

The default Visual C++ warning level is 3 when compiling in the IDE and level 1 on the command line (that is why we use the `/W3` option on the command line); the highest warning level is 4. You can reduce the level to 1 or 2 or disable warnings altogether, but that is not recommended. The only reason you might want to reduce the warning level is to compile older existing C++ source code that does meet newer C++ standards. When developing new code, higher warning levels are preferred since they provide more help to the programmer. Unless otherwise noted, all the complete program examples in this book compile cleanly under Visual C++ set at warning level 3. Level 3 is helpful for detecting many common logic errors.

We can avoid most warnings by a simple addition to the code. Section 4.2 showed how we can use `static_cast` to coerce a wider type to a narrower type. At Visual C++ warning Level 3, the compiler issues a warning if the cast is not used. The little code that must be added should cause the programmer to stop and reflect about the correctness of the construct. The enhanced warning level prevents the programmer from being oblivious to the situation.

Use the strongest level of warnings available to your compiler. Treat all warnings as problems that must be corrected. Do not accept as completed a program that compiles with warnings.

We may assign a `double` literal to a `float` variable without any special type casting. The compiler automatically narrows the `double` to a `float` as Listing 4.9 (assignfloat.cpp) shows:

Listing 4.9: assignfloat.cpp

```
#include <iostream>

int main() {
    float number;
    number = 10.0;  // OK, double literal assignable to a float
    std::cout << "number = " << number << '\n';
}
```

The statement

```
number = 10.0;
```

assigns a `double` literal (10.0) to a `float` variable. You instead may explicitly use a `float` literal as:

```
number = 10.0f;
```

4.7 Arithmetic Examples

The kind of arithmetic to perform in a complex expression is determined on an operator by operator basis. For example, consider Listing 4.10 (tempconv.cpp) that attempts to convert a temperature from degrees Fahrenheit to degrees Celsius using the formula

$$^{\circ}C = \frac{5}{9} \times (^{\circ}F - 32)$$

Listing 4.10: tempconv.cpp

```
// File tempconv.cpp

#include <iostream>

int main() {
    double degreesF, degreesC;
    // Prompt user for temperature to convert
    std::cout << "Enter the temperature in degrees F: ";
    // Read in the user's input
    std::cin >> degreesF;
    // Perform the conversion
    degreesC = 5/9*(degreesF - 32);
    // Report the result
    std::cout << degreesC << '\n';
}
```

Listing 4.10 (tempconv.cpp) contains comments that document each step explaining the code's purpose. An initial test is promising:

```
Enter the temperature in degrees F: 32
Degrees C = 0
```

Water freezes at 32 degrees Fahrenheit and 0 degrees Celsius, so the program's behavior is correct for this test. Several other attempts are less favorable—consider

```
Enter the temperature in degrees F: 212
Degrees C = 0
```

Water boils at 212 degrees Fahrenheit which is 100 degrees Celsius, so this answer is not correct.

```
Enter the temperature in degrees F: -40
Degrees C = 0
```

The value -40 is the point where the Fahrenheit and Celsius curves cross, so the result should be -40, not zero. The first test was only *coincidentally correct*.

Unfortunately, the printed result is always zero regardless of the input. The problem is the division `5/9` in the statement

```
degreesC = 5/9*(degreesF - 32);
```

Division and multiplication have equal precedence, and both are left associative; therefore, the division is performed first. Since both operands are integers, integer division is performed and the quotient is zero (5 divided by 9 is 0, remainder 5). Of course zero times any number is zero, thus the result. The fact that a floating-point value is involved in the expression (`degreesF`) and the overall result is being assigned to a floating-point variable, is irrelevant. The decision about the exact type of operation to perform is made on an operator-by-operator basis, not globally over the entire expression. Since the division is performed first and it involves two integer values, integer division is used before the other floating-point pieces become involved.

One solution simply uses a floating-point literal for either the five or the nine, as in

```
degreesC = 5.0/9*(degreesF - 32);
```

This forces a double-precision floating-point division (recall that the literal `5.0` is a `double`). The correct result, subject to rounding instead of truncation, is finally computed.

Listing 4.11 (timeconv.cpp) uses integer division and modulus to split up a given number of seconds to hours, minutes, and seconds.

Listing 4.11: `timeconv.cpp`

```
// File timeconv.cpp

#include <iostream>

int main() {
    int hours, minutes, seconds;
    std::cout << "Please enter the number of seconds:";
    std::cin >> seconds;
    // First, compute the number of hours in the given number
    // of seconds
    hours = seconds / 3600;  // 3600 seconds = 1 hours
    // Compute the remaining seconds after the hours are
    // accounted for
    seconds = seconds % 3600;
    // Next, compute the number of minutes in the remaining
    // number of seconds
    minutes = seconds / 60;  // 60 seconds = 1 minute
    // Compute the remaining seconds after the minutes are
```

```
    // accounted for
    seconds = seconds % 60;
    // Report the results
    std::cout << hours << " hr, " << minutes << " min, "
              << seconds << " sec\n";
}
```

If the user enters `10000`, the program prints `2 hr, 46 min, 40 sec`. Notice the assignments to the `seconds` variable, such as

```
seconds = seconds % 3600
```

The right side of the assignment operator (`=`) is first evaluated. The remainder of `seconds` divided by 3,600 is assigned back to `seconds`. This statement can alter the value of `seconds` if the current value of `seconds` is greater than 3,600. A similar statement that occurs frequently in programs is one like

```
x = x + 1;
```

This statement increments the variable `x` to make it one bigger. A statement like this one provides further evidence that the C++ assignment operator does not mean mathematical equality. The following statement from mathematics

$$x = x+1$$

is surely never true; a number cannot be equal to one more than itself. If that were the case, I would deposit one dollar in the bank and then insist that I really had two dollars in the bank, since a number is equal to one more than itself. That two dollars would become 3.00, then 4.00, etc., and soon I would be rich. In C++, however, this statement simply means "add one to `x` and assign the result back to `x`."

A variation on Listing 4.11 (timeconv.cpp), Listing 4.12 (enhancedtimeconv.cpp) performs the same logic to compute the time pieces (hours, minutes, and seconds), but it uses more simple arithmetic to produce a slightly different output—instead of printing 11,045 seconds as `3 hr, 4 min, 5 sec`, Listing 4.12 (enhancedtimeconv.cpp) displays it as `3:04:05`. It is trivial to modify Listing 4.11 (timeconv.cpp) so that it would print `3:4:5`, but Listing 4.12 (enhancedtimeconv.cpp) includes some extra arithmetic to put leading zeroes in front of single-digit values for minutes and seconds as is done on digital clock displays.

Listing 4.12: `enhancedtimeconv.cpp`

```
// File enhancedtimeconv.cpp

#include <iostream>

int main() {
    int hours, minutes, seconds;
    std::cout << "Please enter the number of seconds:";
    std::cin >> seconds;
    // First, compute the number of hours in the given number
    // of seconds
    hours = seconds / 3600;  // 3600 seconds = 1 hours
    // Compute the remaining seconds after the hours are
    // accounted for
    seconds = seconds % 3600;
    // Next, compute the number of minutes in the remaining
    // number of seconds
    minutes = seconds / 60;  // 60 seconds = 1 minute
    // Compute the remaining seconds after the minutes are
```

```
    // accounted for
    seconds = seconds % 60;
    // Report the results
    std::cout << hours << ":";
    // Compute tens digit of minutes
    int tens = minutes / 10;
    std::cout << tens;
    // Compute ones digit of minutes
    int ones = minutes % 10;
    std::cout << ones << ":";
    // Compute tens digit of seconds
    tens = seconds / 10;
    std::cout << tens;
    // Compute ones digit of seconds
    ones = seconds % 10;
    std::cout << ones << '\n';
}
```

Listing 4.12 (enhancedtimeconv.cpp) uses the fact that if `x` is a one- or two-digit number, `x / 10` is the tens digit of `x`. If `x / 10` is zero, `x` is necessarily a one-digit number.

4.8 Integers vs. Floating-point Numbers

Floating-point numbers offer some distinct advantages over integers. Floating-point numbers, especially `double`s have a much greater range of values than any integer type. Floating-point numbers can have fractional parts and integers cannot. Integers, however, offer one big advantage that floating-point numbers cannot—exactness. To see why integers are exact and floating-point numbers are not, we will explore the way computers store and manipulate the integer and floating-point types.

Computers store all data internally in binary form. The binary (base 2) number system is much simpler than the familiar decimal (base 10) number system because it uses only two digits: 0 and 1. The decimal system uses 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Despite the lack of digits, every decimal integer has an equivalent binary representation. Binary numbers use a place value system not unlike the decimal system. Figure 4.4 shows how the familiar base 10 place value system works.

Figure 4.4 The base 10 place value system

$$
\begin{array}{cccccccc}
\cdots & \boxed{4} & \boxed{7} & \boxed{3} & \boxed{4} & \boxed{0} & \boxed{6} \\
\cdots & 10^5 & 10^4 & 10^3 & 10^2 & 10^1 & 10^0 \\
\cdots & 100{,}000 & 10{,}000 & 1{,}000 & 100 & 10 & 1
\end{array}
$$

$$
\begin{aligned}
473{,}406 &= 4\times10^5 + 7\times10^4 + 3\times10^3 + 4\times10^2 + 0\times10^1 + 6\times10^0 \\
&= 400{,}000 + 70{,}000 + 3{,}000 + 400 + 0 + 6 \\
&= 473{,}406
\end{aligned}
$$

With 10 digits to work with, the decimal number system distinguishes place values with powers of 10. Compare the base 10 system to the base 2 place value system shown in Figure 4.5.

Figure 4.5 The base 2 place value system

$$\begin{array}{cccccccc} \cdots & \boxed{1} & \boxed{0} & \boxed{0} & \boxed{1} & \boxed{1} & \boxed{1} \\ \cdots & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \cdots & 32 & 16 & 8 & 4 & 2 & 1 \end{array}$$

$$\begin{aligned} 100111_2 &= 1\times 2^5 + 0\times 2^4 + 0\times 2^3 + 1\times 2^2 + 1\times 2^1 + 1\times 2^0 \\ &= 32+0+0+4+2+1 \\ &= 39 \end{aligned}$$

With only two digits to work with, the binary number system distinguishes place values by powers of two. Since both binary and decimal numbers share the digits 0 and 1, we will use the subscript 2 to indicate a binary number; therefore, 100 represents the decimal value *one hundred*, while 100_2 is the binary number *four*. Sometimes to be very clear we will attach a subscript of 10 to a decimal number, as in 100_{10}.

In the decimal system, it is easy to add $3+5$:

$$\begin{array}{rr} & 3 \\ + & 5 \\ \hline & 8 \end{array}$$

The sum $3+9$ is a little more complicated, as early elementary students soon discover:

$$\begin{array}{rr} & 3 \\ + & 9 \\ \hline \end{array}$$

The answer, of course, is 12, but there is no single *digit* that means 12—it takes two digits, 1 and 2. The sum is

$$\begin{array}{rr} & 1 \\ & 03 \\ + & 09 \\ \hline & 12 \end{array}$$

We can say $3+9$ is 2, *carry the* 1. The rules for adding binary numbers are shorter and simpler than decimal numbers:

$$\begin{aligned} 0_2+0_2 &= 0_2 \\ 0_2+1_2 &= 1_2 \\ 1_2+0_2 &= 1_2 \\ 1_2+1_2 &= 10_2 \end{aligned}$$

We can say the sum 1_2+1_2 is 0_2, *carry the* 1_2. A typical larger sum would be

$$\begin{array}{rrr} & & 11 \\ & 9_{10} & = 1001_2 \\ + & 3_{10} & = 11_2 \\ \hline & 12_{10} & = 1100_2 \end{array}$$

4.8.1 Integer Implementation

Mathematical integers are whole numbers (no fractional parts), both positive and negative. Standard C++ supports multiple integer types: `int`, `short`, `long`, and `long long`, `unsigned`, `unsigned short`,

`unsigned long`, and `unsigned long long`. These are distinguished by the number of bits required to store the type, and, consequently, the range of values they can represent. Mathematical integers are infinite, but all of C++'s integer types correspond to finite subsets of the mathematical integers. The most commonly used integer type in C++ is `int`. All `int`s, regardless of their values, occupy the same amount of memory and, therefore use the same number of bits. The exact number of bits in an `int` is processor specific. A 32-bit processor, for example, is built to manipulate 32-bit integers very efficiently. A C++ compiler for such a system most likely would use 32-bit `int`s, while a compiler for a 64-bit machine might represent `int`s with 64 bits. On a 32-bit computer, the numbers 4 and 1,320,002,912 both occupy 32 bits of memory.

For simplicity, we will focus on unsigned integers, particularly the `unsigned` type. The `unsigned` type in Visual C++ occupies 32 bits. With 32 bits we can represent $4,294,967,296$ different values, and so Visual C++'s `unsigned` type represents the integers $0 \ldots 4,294,967,295$. The hardware in many computer systems in the 1990s provided only 16-bit integer types, so it was common then for C++ compilers to support 16-bit `unsigned` values with a range $0 \ldots 65,535$. To simplify our exploration into the properties of computer-based integers, we will consider an even smaller, mythical unsigned integer type that we will call **`unsigned tiny`**. C++ has no such **`unsigned tiny`** type as it has a very small range of values—too small to be useful as an actual type in real programs. Our **`unsigned tiny`** type uses only five bits of storage, and Table 4.3 shows all the values that a variable of type **`unsigned tiny`** can assume.

Binary Bit String	Decimal Value
00000	0
00001	1
00010	2
00011	3
00100	4
00101	5
00110	6
00111	7
01000	8
01001	9
01010	10
01011	11
01100	12
01101	13
01110	14
01111	15
10000	16
10001	17
10010	18
10011	19
10100	20
10101	21
10110	22
10111	23
11000	24
11001	25
11010	26
11011	27
11100	28
11101	29
11110	30
11111	31

Table 4.3: The **`unsigned tiny`** values

Table 4.3 shows that the **`unsigned tiny`** type uses all the combinations of 0s and 1s in five bits. We can derive the decimal number 6 directly from its bit pattern:

$$00110 \implies \frac{0}{16}\frac{0}{8}\frac{1}{4}\frac{1}{2}\frac{0}{1} \implies 0\times 16+0\times 8+1\times 4+1\times 2+0\times 1=6$$

To see that arithmetic works, try adding $4+3$:

$$\begin{array}{rcl} & 4_{10} & = 00100_2 \\ + & 3_{10} & = 00011_2 \\ \hline & 7_{10} & = 00111_2 \end{array}$$

That was easy since involved no carries. Next we will try $3+1$:

$$\begin{array}{rcl} & & 11 \\ & 3_{10} & = 00011_2 \\ + & 1_{10} & = 00001_2 \\ \hline & 4_{10} & = 00100_2 \end{array}$$

In the ones column (rightmost column), $1_2+1_2=10_2$, so write a 0 and carry the 1 to the top of the next column to the left (that is, the twos column). In the twos column, $1_2+1_2+0_2=10_2$, so we must carry a 1 into the fours column as well.

The next example illustrates a limitation of our finite representation. Consider the sum $8+28$:

$$\begin{array}{rcl} & & 11 \\ & 8_{10} & = 01000_2 \\ + & 28_{10} & = 11100_2 \\ \hline & 4_{10} & = 1\,00100_2 \end{array}$$

In the this sum we have a carry of 1 from the eights column to the 16s column, and we have a carry from the 16s column to nowhere. We need a sixth column (a 32s column), another place value, but our **`unsigned tiny`** type is limited to five bits. That carry out from the 16s place is lost. The largest **`unsigned tiny`** value is 31, but $28+8=36$. It is not possible to store the value 36 in an **`unsigned tiny`** just as it is impossible to store the value $5,000,000,000$ in a C++ `unsigned` variable.

Consider exceeding the capacity of the **`unsigned tiny`** type by just one:

$$\begin{array}{rcl} & & 11111 \\ & 31_{10} & = 11111_2 \\ + & 1_{10} & = 00001_2 \\ \hline & 0_{10} & = 1\,00000_2 \end{array}$$

Adding one to the largest possible **`unsigned tiny`**, 31, results in the smallest possible value, 0! This mirrors the behavior of the actual C++ `unsigned` type, as Listing 4.13 (unsignedoverflow.cpp) demonstrates.

Listing 4.13: `unsignedoverflow.cpp`

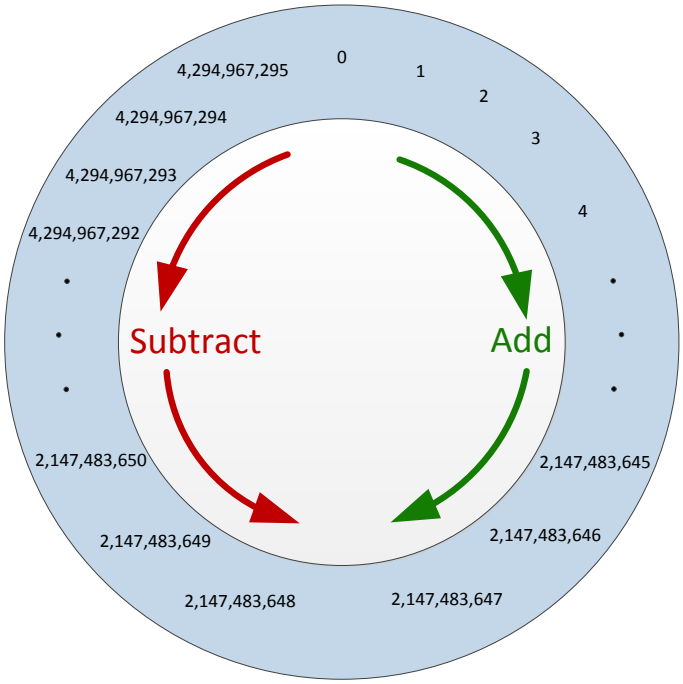
```
#include <iostream>

int main() {
    unsigned x = 4294967293;  // Almost the largest possible unsigned value
    std::cout << x << " + 1 = " << x + 1 << '\n';
    std::cout << x << " + 2 = " << x + 2 << '\n';
    std::cout << x << " + 3 = " << x + 3 << '\n';

}
```

Listing 4.13 (unsignedoverflow.cpp) prints

Figure 4.6 The cyclic nature of 32-bit unsigned integers. Adding 1 to 4,294,967,295 produces 0, one position clockwise from 4,294,967295. Subtracting 4 from 2 yields 4,294,967,294, four places counterclockwise from 2.




```
4294967293 + 1 = 4294967294
4294967293 + 2 = 4294967295
4294967293 + 3 = 0
```

In fact, Visual C++'s 32-bit `unsigned`s follow the cyclic pattern shown in Figure 4.6.

In the figure, an addition moves a value clockwise around the circle, while a subtraction moves a value counterclockwise around the circle. When the numeric limit is reached, the value rolls over like an automobile odometer. Signed integers exibit a similar cyclic pattern as shown in Figure 4.7.

In the case of signed integers, as Figure 4.7 shows, adding one to the largest representable value produces the smallest negative value. Listing 4.14 (integeroverflow.cpp) demonstrates.

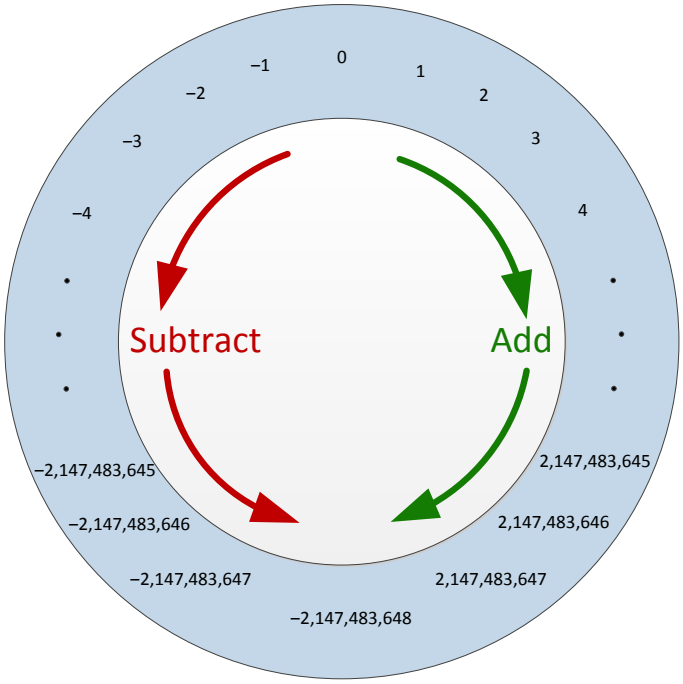
Listing 4.14: `integeroverflow.cpp`

```
#include <iostream>

int main() {
    int x = 2147483645; // Almost the largest possible int value
    std::cout << x << " + 1 = " << x + 1 << '\n';
    std::cout << x << " + 2 = " << x + 2 << '\n';
    std::cout << x << " + 3 = " << x + 3 << '\n';

}
```


Figure 4.7 The cyclic nature of 32-bit signed integers. Adding 1 to $2,147,483,647$ produces $-2,147,483,648$, one position clockwise from $2,147,483,647$. Subtracting 5 from $-2,147,483,645$ yields $2,147,483,646$, five places counterclockwise from $-2,147,483,645$.



Listing 4.14 (integeroverflow.cpp) prints

```
2147483645 + 1 = 2147483646
2147483645 + 2 = 2147483647
2147483645 + 3 = -2147483648
```

Attempting to exceed the maximum limit of a numeric type results in *overflow*, and attempting to exceed the minimum limit is called *underflow*. Integer arithmetic that overflow or underflow produces a valid, yet incorrect integer result. The compiler does not check that a computation will result in exceeding the limit of a type because it is impossible to do so in general (consider adding two integer variables whose values are determined at run time). Also significantly, an overflow or underflow situation does not generate a run-time error. It is, therefore, a logic error if a program performs an integral computation that, either as a final result or an intermediate value, is outside the range of the integer type being used.

4.8.2 Floating-point Implementation

The standard C++ floating point types consist of `float`, `double`, and `long double`. Floating point numbers can have fractional parts (decimal places), and the term floating point refers to the fact the decimal point in a number can float left or right as necessary as the result of a calculation (for example, $2.5 \times 3.3 = 8.25$, two one-decimal place values produce a two-decimal place result). As with the integer types, the different floating-point types may be distinguished by the number of bits of storage required and corresponding range of values. The type `float` stands for *single-precision floating-point*, and `double` stands for *double-precision floating-point*. Floating point numbers serve as rough approximations of mathematical *real numbers*, but as we shall see, they have some severe limitations compared to actual real numbers.

On most modern computer systems floating-point numbers are stored internally in exponential form according to the standard adopted by the Institute for Electrical and Electronic Engineers (IEEE 754). In the decimal system, *scientific notation* is the most familiar form of exponential notation:

> One mole contains 6.023×10^{23} molecules.

Here 6.023 is called the mantissa, and 23 is the exponent.

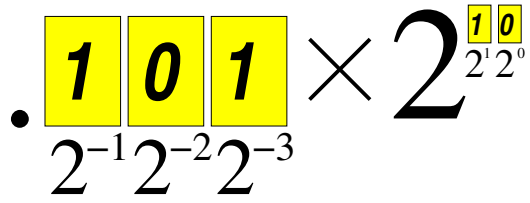
The IEEE 754 standard uses binary exponential notation; that is, the mantissa and exponent are binary numbers. Single-precision floating-point numbers (type `float`) occupy 32 bits, distributed as follows:

Mantissa	24 bits
Exponent	7 bits
Sign	1 bit
Total	32 bits

Double-precision floating-point numbers (type `double`) require 64 bits:

Mantissa	52 bits
Exponent	11 bits
Sign	1 bit
Total	64 bits

Figure 4.8 A **`tiny float`** simplified binary exponential value



The details of the IEEE 754 implementation are beyond the scope of this book, but a simplified example serves to highlight the limitations of floating-point types in general. Recall the fractional place values in the decimal system. The place values, from left to right, are

...	10,000	1,000	100	10	1	•	$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{1000}$	$\frac{1}{10,000}$	...
...	10^4	10^3	10^2	10^1	10^0	•	10^{-1}	10^{-2}	10^{-3}	10^{-4}	...

Each place value is one-tenth the place value to its left. Move to the right, divide by ten; move to the left, multiply by ten. In the binary system, the factor is two instead of ten:

...	16	8	4	2	1	•	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	...
...	2^4	2^3	2^2	2^1	2^0	•	2^{-1}	2^{-2}	2^{-3}	2^{-4}	...

As in our **`unsigned tiny`** example (see Section 4.8.1), consider a a binary exponential number that consists of only five bits—far fewer bits than either `float`s or `double`s in C++. We will call our mythical floating-point type **`tiny float`**. The first three bits of our 5-bit **`tiny float`** type will represent the mantissa, and the remaining two bits store the exponent. The three bits of the mantissa all appear to the right of the binary point. The base of the 2-bit exponent is, of course, two. Figure 4.8 illustrates such a value.

To simplify matters even more, neither the mantissa nor the exponent can be negative. Thus, with three bits, the mantissa may assume one of eight possible values. Since two bits constitute the exponent of **`tiny float`**s, the exponent may assume one of four possible values. Table 4.4 lists all the possible values that **`tiny float`** mantissas and exponents may assume. The number shown in Figure 4.8 is thus

$$\begin{aligned}(1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3})\times 2^{(1\times 2^1+0\times 2^0)} &= \left(\frac{1}{2}+\frac{1}{8}\right)\times 2^2\\ &= \frac{5}{8}\times 4\\ &= 2.5\end{aligned}$$

Table 4.5 combines the mantissas and exponents to reveal all possible **`tiny float`** values that we can represent with the 32 different bit strings made up of five bits. The results are interesting.

The range of our **`tiny float`** numbers is $0\ldots7$. Just stating the range is misleading, however, since it might give the impression that we may represent any value in between 0 and 7 down to the $\frac{1}{8}$th place. This, in fact, is not true. We *can* represent 2.5 with this scheme, but we have no way of expressing 2.25. Figure 4.9 plots all the possible **`tiny float`** values on the real number line.

3-bit Mantissas		
Bit String	Binary Value	Decimal Value
000	0.000_2	$\frac{0}{2}+\frac{0}{4}+\frac{0}{8} = 0.000$
001	0.001_2	$\frac{0}{2}+\frac{0}{4}+\frac{1}{8} = 0.125$
010	0.010_2	$\frac{0}{2}+\frac{1}{4}+\frac{0}{8} = 0.250$
011	0.011_2	$\frac{0}{2}+\frac{1}{4}+\frac{1}{8} = 0.375$
100	0.100_2	$\frac{1}{2}+\frac{0}{4}+\frac{0}{8} = 0.500$
101	0.101_2	$\frac{1}{2}+\frac{0}{4}+\frac{1}{8} = 0.625$
110	0.110_2	$\frac{1}{2}+\frac{1}{4}+\frac{0}{8} = 0.750$
111	0.111_2	$\frac{1}{2}+\frac{1}{4}+\frac{1}{8} = 0.875$

2-bit Exponents		
Bit String	Binary Value	Decimal Value
00	2^{00_2}	$2^0 = 1$
01	2^{01_2}	$2^1 = 2$
10	2^{10_2}	$2^2 = 4$
11	2^{11_2}	$2^3 = 8$

Table 4.4: The eight possible mantissas and four possible exponents that make up all **`tiny float`** values

Figure 4.9 A plot of all the possible **`tiny float`** numbers on the real number line. Note that the numbers are more dense near zero and become more sparse moving to the right. The precision in the range $0\ldots1$ is one-eighth. The precision in the range $1\ldots2$ is only one-fourth, and over the range $2\ldots4$ it drops to one-half. In the range $4\ldots7$ our **`tiny float`** type can represent only whole numbers.

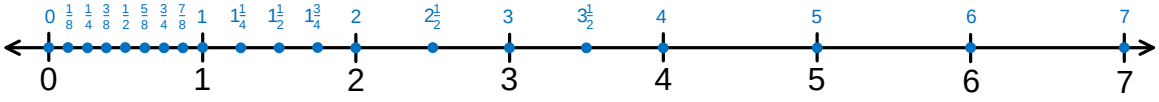


Table 4.5 and Figure 4.9 reveal several troubling issues about our **`tiny float`** type:

1. There are many gaps; for example, the value 2.4 is missing and thus cannot be represented exactly (2.5 is the closest approximation). As another example, 0.75 and 1.75 both appear, but 2.75 is missing.

2. The scheme duplicates some numbers; for example, three different bit patterns represent the decimal value 0.5:

$$0.100 \times 2^{00} \;=\; 0.010 \times 2^{01} \;=\; 0.001 \times 2^{10} \;=\; 0.5_{10}$$

   This duplication limits the number of different values that can be represented by a given number of bits. In our **`tiny float`** example 12 of the 32 bit strings (37.5%) are redundant.

3. The numbers are not uniformly dense. There are more values nearer to zero, and the numbers become more sparse farther away from zero.

Our **`unsigned tiny`** type discussed in Section 4.8.1 exhibits none of these weaknesses. All integers in a given range $(0\ldots31)$ are present, no two bit strings represent the same value, and the integers are uniformly distributed across their specified range. While the standard integer types provided by C++ have much greater ranges than our **`unsigned tiny`** type, they all share these same qualities: all values in their ranges are present, and all bit strings represent unique integer values. The standard floating-point types provided by C++ use many more bits than our **`tiny float`** type, yet they exhibit the same problems

Bit String	Interpretation	Decimal Equivalent	Value
00000	$.000_2 \times 2^{00_2}$	0.000×1	0.000
00001	$.000_2 \times 2^{01_2}$	0.000×2	0.000
00010	$.000_2 \times 2^{10_2}$	0.000×4	0.000
00011	$.000_2 \times 2^{11_2}$	0.000×8	0.000
00100	$.001_2 \times 2^{00_2}$	0.125×1	0.125
00101	$.001_2 \times 2^{01_2}$	0.125×2	0.250
00110	$.001_2 \times 2^{10_2}$	0.125×4	0.500
00111	$.001_2 \times 2^{11_2}$	0.125×8	1.000
01000	$.010_2 \times 2^{00_2}$	0.250×1	0.250
01001	$.010_2 \times 2^{01_2}$	0.250×2	0.500
01010	$.010_2 \times 2^{10_2}$	0.250×4	1.000
01011	$.010_2 \times 2^{11_2}$	0.250×8	2.000
01100	$.011_2 \times 2^{00_2}$	0.375×1	0.375
01101	$.011_2 \times 2^{01_2}$	0.375×2	0.750
01110	$.011_2 \times 2^{10_2}$	0.375×4	1.500
01111	$.011_2 \times 2^{11_2}$	0.375×8	3.000
10000	$.100_2 \times 2^{00_2}$	0.500×1	0.500
10001	$.100_2 \times 2^{01_2}$	0.500×2	1.000
10010	$.100_2 \times 2^{10_2}$	0.500×4	2.000
10011	$.100_2 \times 2^{11_2}$	0.500×8	4.000
10100	$.101_2 \times 2^{00_2}$	0.625×1	0.625
10101	$.101_2 \times 2^{01_2}$	0.625×2	1.250
10110	$.101_2 \times 2^{10_2}$	0.625×4	2.500
10111	$.101_2 \times 2^{11_2}$	0.625×8	5.000
11000	$.110_2 \times 2^{00_2}$	0.750×1	0.750
11001	$.110_2 \times 2^{01_2}$	0.750×2	1.500
11010	$.110_2 \times 2^{10_2}$	0.750×4	3.000
11011	$.110_2 \times 2^{11_2}$	0.750×8	6.000
11100	$.111_2 \times 2^{00_2}$	0.875×1	0.875
11101	$.111_2 \times 2^{01_2}$	0.875×2	1.750
11110	$.111_2 \times 2^{10_2}$	0.875×4	3.500
11111	$.111_2 \times 2^{11_2}$	0.875×8	7.000

Table 4.5: The **`tiny float`** values. The first three bits of the bit string constitute the mantissa, and the last two bits represent the exponent. Given five bits we can produce 32 different bit strings. Notice that due to the ways different mantissas and exponents can combine to produce identical values, the 32 different bit strings yield only 20 unique **`tiny float`** values.

shown to a much smaller degree: missing values, multiple bit patterns representing the same values, and uneven distribution of values across their ranges. This is not solely a problem of C++'s implementation of floating-point numbers; all computer languages and hardware that adhere to the IEEE 754 standard exhibit these problems. To overcome these problems and truly represent and compute with mathematical real numbers we would need a computer with an infinite amount of memory along with an infinitely fast processor.

Listing 4.15 (imprecisedifference.cpp) demonstrates the inexactness of floating-point arithmetic.

Listing 4.15: `imprecisedifference.cpp`

```
#include <iostream>
#include <iomanip>

int main() {
    double d1 = 2000.5;
    double d2 = 2000.0;
    std::cout << std::setprecision(16) << (d1 - d2) << '\n';
    double d3 = 2000.58;
    double d4 = 2000.0;
    std::cout << std::setprecision(16) << (d3 - d4) << '\n';
}
```

The output of Listing 4.15 (imprecisedifference.cpp) is:

```
0.5
0.5799999999999272
```

The program uses an additional `#include` directive:

```
#include <iomanip>
```

This preprocessor directive allows us to use the `std::setprecision` output stream manipulator that directs the `std::cout` output stream object to print more decimal places in floating-point values. During the program's execution, the first subtraction yields the correct answer. We now know that some floating-point numbers (like 0.5) have exact internal representations while others are only approximations. The exact answer for the second subtraction should be 0.58, and if we round the reported result to 12 decimal places, the answer matches. Floating-point arithmetic often produces results that are close approximations of the true answer.

Listing 4.16 (precise8th.cpp) computes zero in a roundabout way:

$$1-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8}-\frac{1}{8} = 0$$

Listing 4.16: `precise8th.cpp`

```
#include <iostream>

int main() {
    double one = 1.0,
           one_eighth = 1.0/8.0,
           zero = one - one_eighth - one_eighth - one_eighth
                      - one_eighth - one_eighth - one_eighth
                      - one_eighth - one_eighth;

    std::cout << "one = " << one << ", one_eighth = " << one_eighth
              << ", zero = " << zero << '\n';

}
```

Listing 4.16 (precise8th.cpp) prints

```
one = 1, one_eighth = 0.125, zero = 0
```

The number $\frac{1}{8}$ has an exact decimal representation, 0.625. It also has an exact binary representation, 0.001_2. Consider, however, $\frac{1}{5}$. While $\frac{1}{5} = 0.2$ has a finite representation in base 10, it has no finite representation in base 2:

$$\frac{1}{5} = 0.2 = 0.0011001100110\overline{0011}_2$$

In the binary representation the 0011_2 bit sequence repeats without end. This means $\frac{1}{5}$ does not have an exact floating-point representation. Listing 4.17 (imprecise5th.cpp) illustrates with arithmetic involving $\frac{1}{5}$.

Listing 4.17: `imprecise5th.cpp`

```
#include <iostream>

int main() {
    double one = 1.0,
           one_fifth = 1.0/5.0,
           zero = one - one_fifth - one_fifth - one_fifth
                      - one_fifth - one_fifth;

    std::cout << "one = " << one << ", one_fifth = " << one_fifth
              << ", zero = " << zero << '\n';

}
```

```
one = 1, one_fifth = 0.2, zero = 5.55112e-017
```

Surely the reported answer ($5.551122 \times 10^{-17} = 0.00000000000000005551122$) is close to the correct answer (zero). If you round it to the one-quadrillionth place (15 places behind the decimal point), it is correct.

What are the ramifications for programmers of this inexactness of floating-point numbers? Section 9.4.6 shows how the misuse of floating-point values can lead to logic errors in programs.

Being careful to avoid overflow and underflow, integer arithmetic is exact and, on most computer systems, faster than floating-point arithmetic. If an application demands the absolute correct answer and integers are appropriate for the computation, you should choose integers. For example, in financial calculations it is important to keep track of every cent. The exact nature of integer arithmetic makes integers an attractive option. When dealing with numbers, an integer type should be the first choice of programmers.

The limitations of floating-point numbers are unavoidable since computers have finite resources. Compromise is inevitable even when we do our best to approximate values with infinite characteristics in a finite way. Despite their inexactness, double-precision floating-point numbers are used every day throughout the world to solve sophisticated scientific and engineering problems; for example, the appropriate use of floating-point numbers have enabled space probes to reach distant planets. In the example C++ programs above that demonstrate the inexactness of floating-point numbers, the problems largely go away if we agree that we must compute with the most digits possible and then round the result to fewer digits. Floating-point numbers provide a good trade-off of precision for practicality.

4.9 More Arithmetic Operators

As Listing 4.12 (enhancedtimeconv.cpp) demonstrates, an executing program can alter a variable's value by performing some arithmetic on its current value. A variable may increase by one or decrease by five. The statement

```
x = x + 1;
```

increments `x` by one, making it one bigger than it was before this statement was executed. C++ has a shorter statement that accomplishes the same effect:

```
x++;
```

This is the *increment* statement. A similar *decrement* statement is available:

```
x--;    // Same as x = x - 1;
```

These statements are more precisely *post-increment* and *post-decrement* operators. There are also *pre-increment* and *pre-decrement* forms, as in

```
--x;    // Same as x = x - 1;
++y;    // Same as y = y + 1;
```

When they appear alone in a statement, the pre- and post- versions of the increment and decrement operators work identically. Their behavior is different when they are embedded within a more complex statement. Listing 4.18 (prevspost.cpp) demonstrates how the pre- and post- increment operators work slightly differently.

Listing 4.18: `prevspost.cpp`

```
#include <iostream>

int main() {
    int x1 = 1, y1 = 10, x2 = 100, y2 = 1000;
    std::cout << "x1=" << x1 << ", y1=" << y1
              << ", x2=" << x2 << ", y2=" << y2 << '\n';
    y1 = x1++;
    std::cout << "x1=" << x1 << ", y1=" << y1
              << ", x2=" << x2 << ", y2=" << y2 << '\n';
    y2 = ++x2;
    std::cout << "x1=" << x1 << ", y1=" << y1
              << ", x2=" << x2 << ", y2=" << y2 << '\n';
}
```

Listing 4.18 (prevspost.cpp) prints

```
x1=1, y1=10, x2=100, y2=1000
x1=2, y1=1, x2=100, y2=1000
x1=2, y1=1, x2=101, y2=101
```

If `x1` has the value 1 just before the statement

```
y1 = x1++;
```

then immediately after the statement executes `x1` is 2 and `y1` is 1.

If `x1` has the value 1 just before the statement

```
y1 = ++x1;
```

then immediately after the statement executes `x1` is 2 and `y1` is also 2.

As you can see, the pre-increment operator uses the new value of the incremented variable when evaluating the overall expression. In contrast, the post-increment operator uses the original value of the incremented variable when evaluating the overall expression. The pre- and post-decrement operator behaves similarly.

For beginning programmers it is best to avoid using the increment and decrement operators within more complex expressions. We will use them frequently as standalone statements since there is no danger of misinterpreting their behavior when they are not part of a more complex expression.

C++ provides a more general way of simplifying a statement that modifies a variable through simple arithmetic. For example, the statement

```
x = x + 5;
```

can be shorted to

```
x += 5;
```

This statement means "increase x by five." Any statement of the form

x *op*= *exp*;

where

- x is a variable.
- *op*= is an arithmetic operator combined with the assignment operator; for our purposes, the ones most useful to us are +=, -=, *=, /=, and %=.
- *exp* is an expression compatible with the variable x.

Arithmetic reassignment statements of this form are equivalent to

x = x *op exp*;

This means the statement

```
x *= y + z;
```

is equivalent to

```
x = x * (y + z);
```

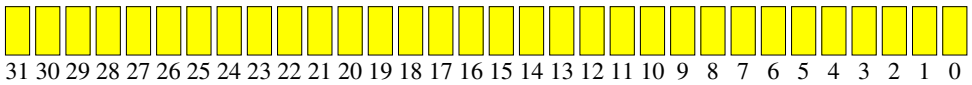
The version using the arithmetic assignment does not require parentheses. The arithmetic assignment is especially handy if a variable with a long name is to be modified; consider

```
temporary_filename_length = temporary_filename_length / (y + z);
```

versus

```
temporary_filename_length /= y + z;
```


Figure 4.10 The bit positions of a 32-bit C++ unsigned integer



Do not accidentally reverse the order of the symbols for the arithmetic assignment operators, like in the statement

```
x =+ 5;
```

Notice that the + and = symbols have been reversed. The compiler interprets this statement as if it had been written

```
x = +5;
```

that is, assignment and the unary operator. This assigns `x` to exactly five instead of increasing it by five.

Similarly,

```
x =- 3;
```

would assign -3 to `x` instead of decreasing `x` by three.

Section 4.10 examines some additional operators available in C++.

4.10 Bitwise Operators

In addition to the common arithmetic operators introduced in Section 4.1, C++ provides a few other special-purpose arithmetic operators. These special operators allow programmers to examine or manipulate the individual bits that make up data values. They are known as the *bitwise operators*. These operators consist of &, |, ^, $\sim$, >>, and <<. Applications programmers generally do not need to use bitwise operators very often, but bit manipulation is essential in many systems programming tasks.

Consider 32-bit unsigned integers. The bit positions usually are numbered right to left, starting with zero. Figure 4.10 shows how the individual bit positions often are numbered.

The bitwise *and* operator, &, takes two integer subexpressions and computes an integer result. The expression e_1 & e_2 is evaluated as follows:

If bit 0 in both e_1 and e_2 is 1, then bit 0 in the result is 1; otherwise, bit 0 in the result is 0.

If bit 1 in both e_1 and e_2 is 1, then bit 1 in the result is 1; otherwise, bit 1 in the result is 0.

If bit 2 in both e_1 and e_2 is 1, then bit 2 in the result is 1; otherwise, bit 2 in the result is 0.

If bit 31 in both e_1 and e_2 is 1, then bit 31 in the result is 1; otherwise, bit 31 in the result is 0.

For example, the expression `13 & 14` evaluates to 12, since:

$$\begin{array}{rcl} 13_{10} & = & 00000000000000000000000000001101_2 \\ \& \;\; 14_{10} & = & 00000000000000000000000000001110_2 \\ \hline 12_{10} & = & 00000000000000000000000000001100_2 \end{array}$$

Bits 2 and 3 are one for both 13 and 14; thus, bits 2 and 3 in the result must be one.

The bitwise *or* operator, `|`, takes two integer subexpressions and computes an integer result. The expression e_1 `|` e_2 is evaluated as follows:

If bit 0 in both e_1 and e_2 is 0, then bit 0 in the result is 0; otherwise, bit 0 in the result is 1.

If bit 1 in both e_1 and e_2 is 0, then bit 1 in the result is 0; otherwise, bit 1 in the result is 1.

If bit 2 in both e_1 and e_2 is 0, then bit 2 in the result is 0; otherwise, bit 2 in the result is 1.

$\vdots$

If bit 31 in both e_1 and e_2 is 0, then bit 31 in the result is 0; otherwise, bit 31 in the result is 1.

For example, the expression `13 | 14` evaluates to 15, since:

$$\begin{array}{rcl} 13_{10} & = & 00000000000000000000000000001101_2 \\ | \;\; 14_{10} & = & 00000000000000000000000000001110_2 \\ \hline 15_{10} & = & 00000000000000000000000000001111_2 \end{array}$$

Bits 4–31 are zero in both 13 and 14. In bits 0–3 either 13 has a one or 14 has a one; therefore, the result has ones in bits 0–3 and zeroes everywhere else.

The bitwise *exclusive or* (often refered to as *xor*) operator (`^`) takes two integer subexpressions and computes an integer result. The expression e_1 `^` e_2 is evaluated as follows:

If bit 0 in e_1 is the same as bit 0 in e_2, then bit 0 in the result is 0; otherwise, bit 0 in the result is 1.

If bit 1 in e_1 is the same as bit 1 in e_2, then bit 1 in the result is 0; otherwise, bit 1 in the result is 1.

If bit 2 in e_1 is the same as bit 2 in e_2, then bit 2 in the result is 0; otherwise, bit 2 in the result is 1.

$\vdots$

If bit 31 in e_1 is the same as bit 31 in e_2, then bit 31 in the result is 0; otherwise, bit 31 in the result is 1.

For example, the expression `13 ^ 14` evaluates to 3, since:

$$\begin{array}{rcl} 13_{10} & = & 00000000000000000000000000001101_2 \\ \hat{} \;\; 14_{10} & = & 00000000000000000000000000001110_2 \\ \hline 3_{10} & = & 00000000000000000000000000000011_2 \end{array}$$

Bits 0 and 1 differ in 13 and 14, so these bits are one in the result. The bits match in all the other positions, so these positions must be set to zero in the result.

The bitwise *negation* operator ($\sim$) is a unary operator that inverts all the bits of its expression. The expression $\sim e$ is evaluated as follows:

If bit 0 in e is 0, then bit 0 in the result is 1; otherwise, bit 0 in the result is 0.

If bit 1 in e is 0, then bit 1 in the result is 1; otherwise, bit 1 in the result is 0.

If bit 2 in e is 0, then bit 2 in the result is 1; otherwise, bit 2 in the result is 0.

$\vdots$

If bit 31 in e is 0, then bit 31 in the result is 1; otherwise, bit 31 in the result is 0.

For example, the `unsigned` expression $\sim$ `13u` evaluates to 4,294,967,282, since

$$\begin{array}{rcl} 13_{10} & = & 00000000000000000000000000001101_2 \\ & & \text{negate} \downarrow \\ \hline -14_{10} & = & 11111111111111111111111111110010_2 \end{array}$$

For signed integers the 31st bit stores the number's sign. Signed integers use a representation called *two's complement* binary, a slight variation of the standard binary layout. Suffice it to say that the `int` expression $\sim$ `13` evaluates to the same bit pattern as $\sim$ `13u`, but as a signed integer it represents -14.

The shift operators move all the bits in an integer to the left or right:

- **Shift left** (`<<`). The expression x `<<` y, where x and y are integer types, shifts all the bits in `x` to the left y places. Zeros fill vacated positions. The bits shifted off the left side are discarded. The expression `5 << 2` evaluates to 20, since $5_{10} = 101_2$ shifted two places to the left yields $10100_2 = 20_{10}$. Observe that x `<<` y is equal to $x \times 2^y$.

- **Shift right** (`>>`). The expression x `>>` y, where x and y are integer types, shifts all the bits in `x` to the right y places. What fills the vacated bits on the left depends on whether the integer is signed or unsigned (for example, `int` vs. `unsigned`):

  - For signed values the vacated bit positions are filled with the sign bit (the original leftmost bit).
  - For unsigned values the vacated bit positions are filled with zeros.

  The bits shifted off the right side are discarded. The expression `5 >> 2` evaluates to 1, since $5_{10} = 101_2$ shifted two places to the left yields $001_2 = 20_{10}$ (the original bits in positions 1 and 0 are shifted off the end and lost). Observe that x `>>` y is equal to $x \div 2^y$.

Do not confuse the left shift operator (`<<`) with the output stream insertion operator (`<<`). The operators are identical, but the context differentiates them. If the left operand is an integer type, `<<` means left shift; if the left operand is a stream output object like `std::cout`, `<<` means send the right-hand operand to the output stream object for display.
Similarly, the input stream object `std::cin` uses the `>>` for a different purpose from the right shift operator used with integers.

Listing 4.19 (bitwiseoperators.cpp) experiments with bitwise operators.

Listing 4.19: bitwiseoperators.cpp

```
#include <iostream>

int main() {
    int x, y;
    std::cout << "Please enter two integers: ";
    std::cin >> x >> y;
    std::cout << x << " & " << y << " = " << (x & y) << '\n';
    std::cout << x << " | " << y << " = " << (x | y) << '\n';
    std::cout << x << " ^ " << y << " = " << (x ^ y) << '\n';
    std::cout << "~" << x << " = " << ~x << '\n';
    std::cout << x << " << " << 2 << " = " << (x << 2) << '\n';
    std::cout << x << " >> " << 2 << " = " << (x >> 2) << '\n';
}
```

Developers use bitwise operations for a variety of systems-level programming tasks. For example, in a graphical user interface (GUI), the user generates *events* by interacting with an application using the mouse and a keyboard. One event might be clicking a mouse button over a particular graphical element (like a button) within a window. Multiple pieces of information about this event can be stored in a single integer. For example, bit 0 may indicate whether or not the ↑ Shift key was held down when the mouse button was clicked. Bit 1 may be responsible for the Alt key, bit 3 for the Ctrl key, etc. Thus the integer value 5, which in binary is

`00000000000000000000000000000101`

means that when the mouse button was clicked both the ↑ Shift and Alt keys were being held down. This might require a different action on the part of the program than if some other combination of keys (or none) were being pressed. For example, suppose the `int` variable `key_status` holds information about which keys the user was depressing during the most recent mouse click. Consider the expression `key_status & 1`. The bit string

`00000000000000000000000000000001`

represents the expression 1, and the value of `key_status` is unknown, so `key_status & 1` is

$$\begin{array}{rrcl} & \texttt{key\_status} & = & \texttt{????????????????????????????????}_2 \\ \& & 1_{10} & = & \texttt{00000000000000000000000000000001}_2 \\ \hline & 0_{10} \text{ or } 1_{10} & = & \texttt{0000000000000000000000000000000?}_2 \end{array}$$

If the answer is zero, this means bit 0 is 0 in `key_status`, and so the ↑ Shift key is not depressed. On the other hand, if the answer is one, this means bit 0 is 1, and so the ↑ Shift key *is* depressed. In the expression

```
key_status & 1
```

the 1 is called a *mask*, since it serves to mask off, or "remove" the first 31 bits of `key_status`.

Usually the GUI library will define constants that help programmers examine or modify particular bits; for example, given the following constants:

```
const int SHIFT_DOWN = 1;                // This is 0...0001
const int CTRL_DOWN = SHIFT_DOWN << 1;   // This is 0...0010
const int ALT_DOWN = CTRL_DOWN << 1;     // This is 0...0100
```


the expression

```
key_status & 1
```

is better written

```
key_status & SHIFT_DOWN
```

The expression

```
key_status & SHIFT_DOWN | ALT_DOWN
```

can test for both the ↑ Shift and Alt keys being down during the mouse event. Do you see how the expression `SHIFT_DOWN | ALT_DOWN` means both keys are down simultaneously?

We can use masks to ensure that certain bits are on or off. To see how, consider the mask 5, which is

```
00000000000000000000000000000101
```

If `x` is a 32-bit integer variable, we can selectively turn on its bits 0 and 2 with the statement

```
x = x | 5;
```

Next, consider the unsigned value `4294967290u`, which is $\sim$ `5u`, or

```
11111111111111111111111111111010
```

If `x` is a 32-bit integer variable, we can selectively turn off its bits 0 and 2 with the statement

```
x = x & 4294967290u;
```

We cannot fully appreciate the utility of using bitwise operators for masking purposes until we consider conditional execution in Chapter 5. Even then, since we concentrate on applications programming instead of systems programming in this book, we will have little use for the bitwise operators except for a few isolated situations. It is good to be aware of their presence, though, since their accidental use may lead to difficult to diagnose compiler messages and logic errors.

Like the other C++ arithmetic operators that work on two operands, we may combine the bitwise binary operators `&`, `|`, `^`, `<<`, and `>>` with assignment to simplify the modification of a variable; for example, the following statement

```
x = x | y;  // Turn on bits in x determined by
```

may be written as

```
x |= y;  // Turn on bits in x determined by
```

Table 4.6 lists the possibilities.

4.11 Algorithms

An *algorithm* is a finite sequence of steps, each step taking a finite length of time, that solves a problem or computes a result. A computer program is one example of an algorithm, as is a recipe to make lasagna. In

Assignment	Short Cut
`x = x & y;`	`x &= y;`
`x = x | y;`	`x |= y;`
`x = x ^ y;`	`x ^= y;`
`x = x << y;`	`x <<= y;`
`x = x >> y;`	`x >>= y;`

Table 4.6: The bitwise assignment operators

both of these examples, the order of the steps matter. In the case of lasagna, the noodles must be cooked in boiling water before they are layered into the filling to be baked. It would be inappropriate to place the raw noodles into the pan with all the other ingredients, bake it, and then later remove the already baked noodles to cook them in boiling water separately. In the same way, the ordering of steps is very important in a computer program. While this point may be obvious, consider the following sound argument:

1. The relationship between degrees Celsius and degrees Fahrenheit can be expressed as

$$^{\circ}C = \frac{5}{9} \times (^{\circ}F - 32)$$

2. Given a temperature in degrees Fahrenheit, the corresponding temperature in degrees Celsius can be computed.

Armed with this knowledge, Listing 4.20 (faultytempconv.cpp) follows directly.

Listing 4.20: `faultytempconv.cpp`

```
// File faultytempconv.cpp

#include <iostream>

int main() {
    double degreesF = 0, degreesC = 0;
    // Define the relationship between F and C
    degreesC = 5.0/9*(degreesF - 32);
    // Prompt user for degrees F
    std::cout << "Enter the temperature in degrees F: ";
    // Read in the user's input
    std::cin >> degreesF;
    // Report the result
    std::cout << degreesC << '\n';
}
```

Unfortunately, the executing program always displays

```
-17.7778
```

regardless of the input provided. The English description provided above is correct. No integer division problems lurk, as in Listing 4.10 (tempconv.cpp). The problem lies simply in statement ordering. The statement

```
degreesC = 5.0/9*(degreesF - 32);
```

is an *assignment* statement, not a definition of a relationship that exists throughout the program. At the point of the assignment, `degreesF` has the value of zero. The executing program computes and assigns the `degreesC` variable *before* receiving `degreesF`'s value from the user.

As another example, suppose `x` and `y` are two integer variables in some program. How would we interchange the values of the two variables? We want `x` to have `y`'s original value and `y` to have `x`'s original value. This code may seem reasonable:

```
x = y;
y = x;
```

The problem with this section of code is that after the first statement is executed, `x` and `y` both have the same value (`y`'s original value). The second assignment is superfluous and does nothing to change the values of `x` or `y`. The solution requires a third variable to remember the original value of one the variables before it is reassigned. The correct code to swap the values is

```
temp = x;
x = y;
y = temp;
```

This small example emphasizes the fact that algorithms must be specified precisely. Informal notions about how to solve a problem can be valuable in the early stages of program design, but the coded program requires a correct detailed description of the solution.

The algorithms we have seen so far have been simple. Statement 1, followed by Statement 2, etc. until every statement in the program has been executed. Chapter 5 and Chapter 6 introduce some language constructs that permit optional and repetitive execution of some statements. These constructs allow us to build programs that do much more interesting things, but more complex algorithms are required to make it happen. We must not lose sight of the fact that a complicated algorithm that is 99% correct is *not* correct. An algorithm's design and implementation can be derailed by inattention to the smallest of details.

4.12 Exercises

1. Is the literal `4` a valid C++ expression?
2. Is the variable `x` a valid C++ expression?
3. Is `x + 4` a valid C++ expression?
4. What affect does the unary `+` operator have when applied to a numeric expression?
5. Sort the following binary operators in order of high to low precedence: `+`, `-`, `*`, `/`, `%`, `=`.
6. Write a C++ program that receives two integer values from the user. The program then should print the sum (addition), difference (subtraction), product (multiplication), quotient (division), and remainder after division (modulus). Your program must use only integers.

   A sample program run would look like (the user enters the 10 and the 2 after the colons, and the program prints the rest):

   ```
   Please enter the first number: 10
   Please enter the second number: 2
   10 + 2 = 12
   10 - 2 = 8
   ```



```
10 * 2 = 20
10 / 2 = 5
10 % 2 = 0
```

Can you explain the results it produces for all of these operations?

7. Write a C++ program that receives two double-precision floating-point values from the user. The program then should print the sum (addition), difference (subtraction), product (multiplication), and quotient (division). Your program should use only integers.

   A sample program run would look like (the user enters the 10 and the 2.5 after the colons, and the program prints the rest):

```
Please enter the first number: 10
Please enter the second number: 2.5
10 + 2.5 = 12.5
10 - 2.5 = 7.5
10 * 2.5 = 25
10 / 2.5 = 4
```

   Can you explain the results it produces for all these operations? What happens if you attempt to compute the remainder after division (modulus) with double-precision floating-point values?

8. Given the following declaration:

```
int x = 2;
```

   Indicate what each of the following C++ statements would print.

   (a) `std::cout << "x"<< '\n';`
   (b) `std::cout << 'x'<< '\n';`
   (c) `std::cout << x << '\n';`
   (d) `std::cout << "x + 1"<< '\n';`
   (e) `std::cout << 'x'+ 1 << '\n';`
   (f) `std::cout << x + 1 << '\n';`

9. Sort the following types in order from narrowest to widest: `int`, `double`, `float`, `long`, `char`.

10. Given the following declarations:

```
int i1 = 2, i2 = 5, i3 = -3;
double d1 = 2.0, d2 = 5.0, d3 = -0.5;
```

    Evaluate each of the following C++ expressions.

    (a) `i1 + i2`
    (b) `i1 / i2`
    (c) `i2 / i1`
    (d) `i1 * i3`
    (e) `d1 + d2`
    (f) `d1 / d2`
    (g) `d2 / d1`

(h) `d3 * d1`

(i) `d1 + i2`

(j) `i1 / d2`

(k) `d2 / i1`

(l) `i2 / d1`

(m) `i1/i2*d1`

(n) `d1*i1/i2`

(o) `d1/d2*i1`

(p) `i1*d1/d2`

(q) `i2/i1*d1`

(r) `d1*i2/i1`

(s) `d2/d1*i1`

(t) `i1*d2/d1`

11. What is printed by the following statement:

```
std::cout << /* 5 */ 3 << '\n';
```

12. Given the following declarations:

```
int i1 = 2, i2 = 5, i3 = -3;
double d1 = 2.0, d2 = 5.0, d3 = -0.5;
```

Evaluate each of the following C++ expressions.

(a) `i1 + (i2 * i3)`

(b) `i1 * (i2 + i3)`

(c) `i1 / (i2 + i3)`

(d) `i1 / i2 + i3`

(e) `3 + 4 + 5 / 3`

(f) `(3 + 4 + 5) / 3`

(g) `d1 + (d2 * d3)`

(h) `d1 + d2 * d3`

(i) `d1 / d2 - d3`

(j) `d1 / (d2 - d3)`

(k) `d1 + d2 + d3 / 3`

(l) `(d1 + d2 + d3) / 3`

(m) `d1 + d2 + (d3 / 3)`

(n) `3 * (d1 + d2) * (d1 - d3)`

13. How are single-line comments different from block comments?

14. Can block comments be nested?

15. Which is better, too many comments or too few comments?

16. What is the purpose of comments?

17. The programs in Listing 3.4 (variable.cpp), Listing 4.4 (reformattedvariable.cpp), and Listing 4.5 (reformattedvariable2.cpp) compile to the same machine code and behave exactly the same. What makes one of the programs clearly better than the others?

18. Why is human readability such an important consideration?

19. Consider the following program which contains some errors. You may assume that the comments within the program accurately describe the program's intended behavior.

```
#include <iostream>

int main() {
    int n1, n2, d1;                          // 1
    // Get two numbers from the user
    cin << n1 << n2;                         // 2
    // Compute sum of the two numbers
    std::cout << n1 + n2 << '\n';            // 3
    // Compute average of the two numbers
    std::cout << n1+n2/2 << '\n';            // 4
    // Assign some variables
    d1 = d2 = 0;                             // 5
    // Compute a quotient
    std::cout << n1/d1 << '\n';              // 6
    // Compute a product
    n1*n2 = d1;                              // 7
    // Print result
    std::cout << d1 << '\n';                 // 8
}
```

    For each line listed in the comments, indicate whether or not a compile-time, run-time, or logic error is present. Not all lines contain an error.

20. What distinguishes a compiler warning from a compiler error? Should you be concerned about warnings? Why or why not?

21. What are the advantages to enhancing the warning reporting capabilities of the compiler?

22. Write the shortest way to express each of the following statements.

    (a) `x = x + 1;`

    (b) `x = x / 2;`

    (c) `x = x - 1;`

    (d) `x = x + y;`

    (e) `x = x - (y + 7);`

    (f) `x = 2*x;`

    (g) `number_of_closed_cases = number_of_closed_cases + 2*ncc;`

23. What is printed by the following code fragment?

```
int x1 = 2, y1, x2 = 2, y2;
y1 = ++x1;
y2 = x2++;
std::cout << x1 << " " << x2 << '\n';
std::cout << y1 << " " << y2 << '\n';
```

Why does the output appear as it does?

24. Consider the following program that attempts to compute the circumference of a circle given the radius entered by the user. Given a circle's radius, r, the circle's circumference, C is given by the formula:

$$C = 2\pi r$$

```
#include <iostream>

int main() {
    double C, r;
    const double PI = 3.14159;
    // Formula for the area of a circle given its radius
    C = 2*PI*r;
    // Get the radius from the user
    cout >> "Please enter the circle's radius: ";
    cin << r;
    // Print the circumference
    std::cout << "Circumference is " << C << '\n';
}
```

(a) The compiler issues a warning. What is the warning?

(b) The program does not produce the intended result. Why?

(c) How can it be repaired so that it not only eliminates the warning but also removes the logic error?

25. In mathematics, the midpoint between the two points (x_1, y_1) and (x_2, y_2) is computed by the formula

$$\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$$

Write a C++ program that receives two mathematical points from the user and computes and prints their midpoint.

A sample run of the program produces

```
Please enter the first point: (0,0)
Please enter the second point: (1,1)
The midpoint of (0,0) and (1,1) is (0.5,0.5)
```

The user literally enters "(0,0)" and "(1,1)" with the parentheses and commas as shown. To see how to do this, suppose you want to allow a user to enter the point $(2.3, 9)$, assigning the x component of the point to a variable named `x` and the y component to a variable named `y`. You can add the following code fragment to your program to achieve the desired effect:

Food	Calories
Bean burrito	357
Salad w/dressing	185
Milkshake	388

Table 4.7: Calorie content of several fast food items

```
double x, y;
char left_paren, comma, right_paren;
std::cin >> left_paren >> x >> comma >> y >> right_paren;
```

If the user literally types `(2.3,9)`, the `std::cin` statement will assign the `(` character to the variable `left_paren`. It next will assign 2.3 to the variable `x`. It assigns the `,` character to the variable named `comma`, the value 9 to the `y` variable, and the `)` character to the `right_paren` variable. The `left_paren`, `comma`, and `right_paren` variables are just placeholders for the user's input and are not used elsewhere within the program. In reality, the user can type in other characters in place of the parentheses and comma as long as the numbers are in the proper location relative to the characters; for example, the user can type `*2.3:9#`, and the program will interpret the input as the point $(2.3, 9)$.

26. Table 4.7 lists the Calorie contents of several foods. Running or walking burns off about 100 Calories per mile. Write a C++ program that requests three values from the user: the number of bean burritos, salads, and shakes consumed (in that order). The program should then display the number of miles that must be run or walked to burn off the Calories represented in that food. The program should run as follows (the user types in the 3 2 1):

```
Number of bean burritos, bowls of salad, and milkshakes eaten?   3 2 1

You ingested 1829 Calories
You will have to run 18.29 miles to expend that much energy
```

Observe that the result is a floating-point value, so you should use floating-point arithmetic to compute the answers for this problem.