# 2. Introducing Fundamental Types of Data

In this chapter, we’ll explain the fundamental data types that are built into C++. We’ll need these in every program. All of the object-oriented capabilities are founded on these fundamental data types because all the data types that you create are ultimately defined in terms of the basic numerical data our computer works with. By the end of the chapter, we’ll be able to write a simple C++ program of the traditional form: input – process – output.

In this chapter, we’ll learn

* What a fundamental data type is in C++
* How you declare, initialize, and reassign variables
* How you can fix the value of a variable
* What integer literals are and how you define them
* How calculations work
* How to work with variables that contain floating-point values
* Which elementary mathematical functions and constants you have at our disposal
* How to convert variables from one type to another
* How to control the format that is used when converting a variable into a string
* How to work with variables that store characters
* What the auto keyword does

## Variables, Data, and Data Types

A *variable* is a named piece of memory that we define. Each variable stores data only of a particular type. Every variable has a *type* that defines the kind of data it can store. Each fundamental type is identified by a unique type name that consists of one or more *keywords*. Keywords are reserved words in C++ that you cannot use for anything else.

The compiler makes extensive checks to ensure that we use the right data type in any given context. It will also ensure that when we combine different types in an operation such as adding two values, for example, either they are of the same type or they can be made to be compatible by converting one value to the type of the other. The compiler detects and reports attempts to combine data of different types that are incompatible.

Numerical values fall into two broad categories: integers, which are whole numbers, and floating-point values, which can be nonintegral. There are several fundamental C++ types in each category, each of which can store a specific range of values. We’ll start with integer types.

### Defining Integer Variables

Here’s a statement that defines an integer variable:

int apple\_count;

This defines a variable of type int with the name apple\_count. The variable will contain some arbitrary junk value. You can and should specify an initial value when you define the variable, like this:

int apple\_count {15}; // Number of apples

The initial value for apple\_count appears between the braces following the name so it has the value 15. The braces enclosing the initial value are called a *braced initializer*. We’ll meet situations later in the book where a braced initializer will have several values between the braces. You don’t have to initialize variables when you define them, but it’s a good idea to do so. Ensuring variables start out with known values makes it easier to work out what is wrong when the code doesn’t work as you expect.

The size of variables of type int is typically four bytes, so they can store integers from –2,147,483,648 to

+2,147,483,647. This covers most situations, which is why int is the integer type that is used most frequently.

Here are definitions for three variables of type int:

int apple\_count {15}; // Number of apples

int orange\_count {5}; // Number of oranges

int total\_fruit {apple\_count + orange\_count}; // Total number of fruit

The initial value for total\_fruit is the sum of the values of two variables defined previously. This demonstrates that the initial value for a variable can be an expression. The statements that define the two variables in the expression for the initial value for total\_fruit must appear earlier in the source file; otherwise, the definition for total\_fruit won’t compile.

There are two other ways for initializing a variable: *functional notation* and *assignment notation*. These look like this (yes, tomato’s a fruit as well):

int lemon\_count(4); // Functional notation int tomato\_count = 12; // Assignment notation

Most of the time, these three notations—curly braces, functional, and assignment notation—are equivalent. The braced initializer form, however, is slightly safer when it comes to so-called *narrowing conversions*. A narrowing conversion changes a value to a type with a more limited range of values. Any such conversion thus has the potential to lose information. Here is an example:

int banana\_count(7.5); // Typically compiles without warning int tangerine\_count = 5.3; // Typically compiles without warning

Normally, the initial value you provide will be of the same type as the variable you are defining. If it isn’t, though, the compiler will try to convert it to the required type. In our previous example, we specified non-integer initial values for two integer variables. We’ll have more to say about floating-point to integer conversions later, but for now believe us when we say that after these variable definitions banana\_count and tangerine\_count will contain the integer values 7 and 5, respectively. It’s unlikely that this is what the author had in mind.

Nevertheless, as far as the C++ standard is concerned, these two definitions are perfectly legal. They are allowed to compile without even the slightest warning. While some compilers do issue a warning about such flagrant narrowing conversions, definitely not all of them do. If you use the braced initializer form, however, a conforming compiler is required to at least issue a diagnostic message. For instance:

int papaya\_count{0.3}; // At least a compiler warning, often an error

If this statement compiles, papaya\_count will be initialized to the integer value 0. But at least the compiler will have given you a warning that something may be amiss. Some compilers will even issue an error and refuse to compile such definitions altogether.

We believe inadvertent narrowing conversions do not deserve to go unnoticed, as they often are a mistake. In this book we’ll therefore embrace the braced initializer syntax. This is the most recent syntax that was introduced in C++11 specifically to standardize initialization. Besides providing better safety guarantees when it comes to narrowing conversions, its main advantage is that it enables you to initialize just about everything in the same way—which is why it is also commonly referred to as *uniform initialization*.

**Note** To represent fractional numbers, you typically use floating-point variables rather than integers. We’ll describe these later in this chapter.

We can define and initialize more than one variable of a given type in a single statement. Here’s an example:

int foot\_count {2}, toe\_count {10}, head\_count {1};

While this is legal, it’s often considered best to define each variable in a separate statement. This makes the code more readable, and you can explain the purpose of each variable in a comment.

You can write the value of any variable of a fundamental type to the standard output stream. Here’s a program that does that with a couple of integers:

// Ex2\_01.cpp

// Writing values of variables to the screen

import <iostream>; // For user input and output through std::cin / cout

int main()

{

int apple\_count {15}; // Number of apples

int orange\_count {5}; // Number of oranges

int total\_fruit {apple\_count + orange\_count}; // Total number of fruit

std::cout << "The value of apple\_count is " << apple\_count << std::endl; std::cout << "The value of orange\_count is " << orange\_count << std::endl; std::cout << "The value of total\_fruit is " << total\_fruit << std::endl;

}

If you compile and execute this, we’ll see that it outputs the values of the three variables following some text explaining what they are. The integer values are automatically converted to a character representation for output by the insertion operator, <<. This works for values of any of the fundamental types.

**Tip** The three variables in Ex2\_01.cpp, of course, do not really need any comments explaining what they represent. Their variable names already make that crystal clear—as they should! In contrast, a lesser programmer might have produced the following, for instance:

int n {15};

int m {5};

int t {n + m};

Without extra context or explanation, no one would ever be able to guess this code is about counting fruit. We should therefore always choose our variable names as self-descriptive as possible. Properly named variables and functions mostly need no additional explanation in the form of a comment. Of course, this does not mean we should never add comments to declarations. We cannot always capture everything in a single name. A few words or, if need be, a little paragraph of comments can then do wonders in helping someone understand our code. A little extra effort at the time of writing can considerably speed up future development!

#### Signed Integer Types

Table [2-1](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark0) shows the complete set of fundamental types that store signed integers—that is, both positive and negative values. The memory allocated for each type, and hence the range of values it can store, may vary between different compilers. Table [2-1](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark0) shows the sizes and ranges used by compilers for all common platforms and computer architectures.

***Table 2-1.*** *Signed Integer Types*

|  |  |  |  |
| --- | --- | --- | --- |
| **Type Name** | **Typical Size(Bytes)** | **Typical Range of Values** | **Theoretical Minimum Size** |
| signed char | 1 | –128 to +127 | 1 |
| short short int signed short  signed short int | 2 | –32,768 to +32,767 | 2 |
| int signed signed int | 4 | –2,147,483,648 to  +2,147,483,647 | 2 |
| long long int signed long  signed long int | 4 or 8 | Same as int or long long | 4 |
| long long long long int signed long long  singed long long int | 8 | –9,223,372,036,854,775,808 to  +9,223,372,036,854,775,807 | 8 |

Type signed char is always one byte (which in turn nearly always is eight bits); the number of bytes occupied by the other types depends on the compiler. Each type will always have at least as much memory as the one that precedes it in the list, though, and (as of C++20) at least as much as shown in the last column of Table [2-1](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark0).

Where two type names appear in the first column of Table [2-1](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark0), the abbreviated name that comes first is more commonly used. That is, you will usually see long used rather than long int or signed long int.

The signed modifier is mostly optional; if omitted, our type will be signed by default. The only exception to this rule is char. While the unmodified type char does exist, it is compiler-dependent whether it is signed or unsigned. We’ll discuss this further in the next subsection. For all integer types other than char, however, you are free to choose whether you add the signed modifier. Personally, we normally do so only when we really want to stress that a particular variable is signed.

#### Unsigned Integer Types

Of course, there are circumstances where you don’t need to store negative numbers. The number of students in a class or the number of parts in an assembly is always a positive integer. You can specify integer types that only store non-negative values by prefixing any of the names of the signed integer types with the unsigned keyword—types unsigned char or unsigned short or unsigned long long, for example. Each unsigned type is a different type from the signed type but occupies the same amount of memory.

Unlike other integer types, type char is a different integer type from both signed char and

unsigned char. The char type is intended only for variables that store character codes and can be a signed or unsigned type depending on our compiler.[1](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark1) We’ll have more to say about variables that store characters later in this chapter.

Tip Only use variables of the unmodified char type to store letter characters. To store numbers, you should use either signed char or unsigned char. And if you ever find yourself having to store, refer to, or manipulate raw binary data (an advanced topic not discussed in this book), you should use C++20’s std::byte type over the more traditional char or unsigned char.

With the exception of unsigned char (with possible values 0 to 255), and possibly unsigned short (typically 0 to 65,535), increasing the range of representable numbers is rarely the main motivator for adding the unsigned modifier. It rarely matters, for instance, whether you can represent numbers up to

2,147,483,647 or up to 4,294,967,295. Instead, you mostly add the unsigned modifier to make our code more self-documenting, to make it more predictable what values a given variable will or should contain.

**Note** We can also use the keywords signed and unsigned on their own. As Table [2-1](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark0) shows, the type signed is considered shorthand for signed int. So naturally, unsigned is short for unsigned int.

## Zero Initialization

The following statement defines an integer variable with an initial value equal to zero:

int counter {0}; // counter starts at zero

We could omit the 0 in the braced initializer here, and the effect would be the same. The statement that defines counter could thus be written like this:

int counter {}; // counter starts at zero

The empty curly braces somewhat resemble the number zero, which makes this syntax easy to remember. *Zero initialization* works for any fundamental type. For all fundamental numeric types, for instance, an empty braced initializer is always assumed to contain the number zero.

## Defining Variables with Fixed Values

Sometimes we’ll want to define variables with values that are fixed and must not be changed. You use the const keyword in the definition of a variable that must not be changed. Such variables are often referred to as *constants*. Here’s an example:

const unsigned toe\_count {10}; // An unsigned integer with fixed value 10

The const keyword tells the compiler that the value of toe\_count must not be changed. Any statement that attempts to modify this value will be flagged as an error during compilation; cutting off someone’s toe is a definite no-no! You can use the const keyword to fix the value of variables of any type.

**Tip** If nothing else, knowing which variables can and cannot change their values along the way makes our code easier to follow. So, we recommend you add the const specifier whenever applicable.

## Integer Literals

Constant values of any kind, such as 42, 2.71828, 'Z', or "Mark Twain", are referred to as *literals*. These examples are, in sequence, an *integer literal*, a *floating-point literal*, a *character literal*, and a *string literal*. Every literal will be of some type. We’ll first explain integer literals and introduce the other kinds of literals in context later.

### Decimal Integer Literals

You can write integer literals in a very straightforward way. Here are some examples of decimal integers:

-123L +123 123 22333 98U -1234LL 12345ULL

Unsigned integer literals have u or U appended. Literals of types long and type long long have L or LL appended, respectively, and if they are unsigned, they also have u or U appended. If there is no suffix, an integer constant is of type int. The U and L or LL can be in either sequence. You can use lowercase for the L and LL suffixes, but we recommend that you don’t because lowercase l is easily confused with the digit 1.

We could omit the + in the second example, as it’s implied by default, but if putting it in makes things clearer, that’s not a problem. The literal +123 is the same as 123 and is of type int because there is no suffix.

The fourth example, 22333, is the number that we, depending on local conventions, might write as either 22,333; 22 333; or 22.333 (though other formatting conventions exist as well). We must not use

commas or spaces in a C++ integer literal, though, and adding a dot would turn it into a floating-point literal (as discussed later). Ever since C++14, however, we can use the single quote character, ', to make numeric literals more readable. Here’s an example:

22'333 -1'234LL 12'345ULL

Here are some statements using some of these literals:

unsigned long age {99UL}; // 99ul or 99LU would be OK too

unsigned short price {10u}; // There is no specific literal type for short long long distance {15'000'000LL}; // Common digit grouping of the number 15 million

Note that there are no restrictions on how to group the digits. Most Western conventions group digits per three, but this is not universal. Natives of the exotic subcontinent of India, for instance, would typically write the literal for 15 million as follows (using groups of two digits except for the rightmost group of three digits):

1'50'00'000LL

So far we have been very diligent in adding our literal suffixes—u or U for unsigned literals, L for literals of type long, and so on. In practice, however, we’ll rarely add these in variable initializers of this form. The reason is that no compiler will ever complain if you simply type this:

unsigned long age {99};

unsigned short price {10}; // There is no specific literal type for short long long distance {15'000'000}; // Common digit grouping of the number 15 million

While all these literals are technically of type (signed) int, our compiler will happily convert them to the correct type for you. As long as the target type can represent the given values without loss of information, there’s no need to issue a warning.

Note While mostly optional, there are situations where you do need to add the correct literal suffixes, such as when you initialize a variable with type auto (as explained near the end of this chapter) or when calling overloaded functions with literal arguments .

An initializing value should always be within the permitted range for the type of variable, as well as from the correct type. The following two statements violate these restrictions. They require, in other words, narrowing conversions:

unsigned char high\_score { 513U }; // The valid range for unsigned char is [0,255] unsigned int high\_score { -1 }; // -1 is a literal of type signed int

As we explained earlier, both these braced initialization statements will result in at least a compiler warning, if not a compilation error, depending on which compiler we use.

### Hexadecimal Literals

We can write integer literals as hexadecimal values. We prefix a hexadecimal literal with 0x or 0X, so 0x999 is a hexadecimal number of type int with three hexadecimal digits. Plain old 999, on the other hand, is a decimal value of type int with decimal digits, so the value will be completely different. Here are some more examples of hexadecimal literals:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Hexadecimal literals: | 0x1AF | 0x123U | 0xAL | 0xcad | 0xFF |
| Decimal literals: | 431 | 291U | 10L | 3245 | 255 |

A major use for hexadecimal literals is to define particular patterns of bits. Each hexadecimal digit corresponds to four bits, so it’s easy to express a pattern of bits as a hexadecimal literal. The red, blue, and green components (RGB values) of a pixel color, for instance, are often expressed as three bytes packed into a 32-bit word. The color white can be specified as 0xFFFFFF because the intensity of each of the three

components in white have the same maximum value of 255, which is 0xFF. The color red would be 0xff0000. Here are some examples:

unsigned int color {0x0ff1ce}; // Unsigned int hexadecimal constant - decimal 1,044,942 int mask {0XFF00FF00}; // Four bytes specified as FF, 00, FF, 00

unsigned long value {0xDEADlu}; // Unsigned long hexadecimal literal - decimal 57,005

### Octal Literals

We can also write integer literals as octal values—that is, using base 8. You identify a number as octal by writing it with a leading zero.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Octal literals: | 0657 | 0443U | 012L | 06255 | 0377 |
| Decimal literals: | 431 | 291U | 10L | 3245 | 255 |

**Caution** Don’t write decimal integer values with a leading zero. The compiler will interpret such values as octal (base 8), so a value written as 065 will be the equivalent of 53 in decimal notation.

### Binary Literals

Binary literals were introduced by the C++14 standard. You write a binary integer literal as a sequence of binary digits (0 or 1) prefixed by either 0b or 0B. As always, a binary literal can have L or LL as a suffix to indicate it is type long or long long, and u or U if it is an unsigned literal. Here are some examples:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Binary literals: | 0B110101111 | 0b100100011U | 0b1010L | 0B110010101101 | 0b11111111 |
| Decimal literals: | 431 | 291U | 10L | 3245 | 255 |

We have illustrated in the code fragments how you can write various combinations for the prefixes and suffixes such as 0x or 0X and UL, LU, or Lu, but of course it’s best to stick to a consistent way of writing integer literals.

As far as our compiler is concerned, it doesn’t matter which number base you choose when you write an integer value. Ultimately it will be stored as a binary number. The different ways for writing an integer are there just for our convenience. You choose one or other of the possible representations to suit the context.

**Note** You can use a single quote as a separator in any integer literal to make it easier to read. This includes hexadecimal or binary literals. For instance: 0xFF00'00FFu or 0b11001010'11011001.

## Calculations with Integers

To begin with, let’s get some bits of terminology out of the way. An operation such as addition or multiplication is defined by an *operator*—the operators for addition and multiplication are + and \*, respectively. The values that an operator acts upon are called *operands*, so in an expression such as 2 \* 3, the operands are 2 and 3. Operators such as multiplication that require two operands are called *binary operators*. Operators that require one operand are called *unary operators*. An example of a unary operator is the minus sign in the expression -width. The minus sign negates the value of width, so the result of the expression is a value with the opposite sign to that of its operand. This contrasts with the binary multiplication operator in expressions such as width \* height, which acts on two operands, width and height.

Table [2-2](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark2) shows the basic arithmetic operations that you can carry out on integers.

***Table 2-2.*** *Basic Arithmetic Operations*

Operator Operation

+ Addition

- Subtraction

\* Multiplication

/ Division

% Modulus (the remainder after division)

The operators in Table [2-2](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark2) are all binary operators and work largely in the way you would expect. There are two operators that may need a little word of explanation, though: the somewhat lesser-known modulus operator, of course, but also the division operator. Integer division is slightly idiosyncratic in C++. When applied to two integer operands, the result of a division operation is always again an integer. Suppose, for instance, that you write the following:

int numerator = 11;

int quotient = numerator / 4;

Mathematically speaking, the result of the division 11/4 is of course 2.75 or 2¾, that is, two and three quarters. But 2.75 is clearly no integer, so what to do? Any sane mathematician would suggest that you round the quotient to the nearest integer, so 3. But, alas, that is *not* what our computer will do. Instead, our computer will simply discard the fractional part, 0.75, altogether. No doubt this is because proper rounding would require more complicated circuitry and hence also more time to evaluate. This means that, in C++, 11/4 will always give the integer value 2. Figure [2-1](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark3) illustrates the effects of the division and modulus operators on our example.

**Integer Divide Operator**

11/4 2 times 4 Remainder 3

Result = **2** Discarded

**Modulus Operator**

11%4 2 times 4 Remainder 3

Discarded Result = **3**

***Figure 2-1.*** *Contrasting the division and modulus operators*

Integer division returns the number of times that the denominator divides into the numerator. Any remainder is discarded. The modulus operator, %, complements the division operator in that it produces the *remainder after integer division*. It is defined such that, for all integers x and y, (x / y) \* y + (x % y)

equals x. Using this formula, you can easily deduce what the modulus operand will do for negative operands.

The result of both the division and modulus operator is undefined when the right operand is zero— what’ll happen depends, in other words, on our compiler and computer architecture.

### Compound Arithmetic Expressions

If multiple operators appear in the same expression, multiplication, division, and modulus operations always execute before addition and subtraction. Here’s an example of such a case:

long width {4};

long length {5};

long area { width \* length }; // Result is 20 long perimeter {2\*width + 2\*length}; // Result is 18

We can control the order in which more complicated expressions are executed using parentheses. We could write the statement that calculates a value for perimeter as follows:

long perimeter{ (width + length) \* 2 }; // Result is 18

The subexpression within the parentheses is evaluated first. The result then is multiplied by two, which produces the same end result as before. If we omit the parentheses here, however, the result would no longer be 18. The result, instead, would become 14:

long perimeter{ width + length \* 2 }; // Result is 14

The reason is that multiplication is always evaluated before addition. So, the previous statement is actually equivalent to the following one:

long perimeter{ width + (length \* 2) };

Parentheses can be nested, in which case subexpressions between parentheses are executed in sequence from the innermost pair of parentheses to the outermost. This example of an expression with nested parentheses will show how it works:

2\*(a + 3\*(b + 4\*(c + 5\*d)))

The expression 5\*d is evaluated first, and c is added to the result. That result is multiplied by 4, and b is added. That result is multiplied by 3, and a is added. Finally, that result is multiplied by 2 to produce the result of the complete expression.

We will have more to say about the order in which such *compound expressions* are evaluated in the next chapter. The main thing to remember is that whatever the default evaluation order is, you can always override it by adding parentheses. And even if the default order happens to be what you want, it never hurts to add some extra parentheses just for the sake of clarity:

long perimeter{ (2\*width) + (2\*length) }; // Result is 18

## Assignment Operations

In C++, the value of a variable is fixed only if we use the const qualifier. In all other cases, the value of a variable can always be overwritten with a new value:

long perimeter {};

// ...

perimeter = 2 \* (width + length);

This last line is an *assignment statement*, and the = is the *assignment operator*. The arithmetic expression on the right of the assignment operator is evaluated, and the result is stored in the variable on the left.

Initializing the perimeter variable upon declaration may not be strictly necessary—as long as the variable is not read prior to the assignment, that is—but it’s considered good practice to always initialize our variables nevertheless. And zero is often as good a value as any.

We can assign a value to more than one variable in a single statement. Here’s an example:

int a {}, b {}, c {5}, d{4}; a = b = c\*c - d\*d

The second statement calculates the value of the expression c\*c - d\*d and stores the result in b, so b will be set to 9. Next the value of b is stored in a, so a will also be set to 9. You can have as many repeated assignments like this as you want.

It’s important to appreciate that an assignment operator is quite different from an = sign in an algebraic equation. The latter implies equality, whereas the former is specifying an action—specifically, the act of overwriting a given memory location. A variable can be overwritten as many times as you want, each time with different, mathematically nonequal values. Consider the assignment statement in the following:

int y {5}; y = y + 1;

The variable y is initialized with 5, so the expression y + 1 produces 6. This result is stored back in y, so the effect is to increment y by 1. This last line makes no sense in common math: as any mathematician will tell you, y can never equal y + 1 (except of course when y equals infinity…). But in programming languages such as C++ repeatedly incrementing a variable with one is actually extremely common. In Chapter 5, we’ll find that equivalent expressions are, for instance, ubiquitous in loops.

Let’s see some of the arithmetic operators in action in an example. This program converts distances that you enter from the keyboard and in the process illustrates using the arithmetic operators:

// Ex2\_02.cpp

// Converting distances

import <iostream>; // For user input and output through std::cin / cout

int main()

{

unsigned int yards {}, feet {}, inches {};

// Convert a distance in yards, feet, and inches to inches std::cout << "Enter a distance as yards, feet, and inches "

<< "with the three values separated by spaces: "; std::cin >> yards >> feet >> inches;

const unsigned feet\_per\_yard {3}; const unsigned inches\_per\_foot {12};

unsigned total\_inches {};

total\_inches = inches + inches\_per\_foot \* (yards\*feet\_per\_yard + feet); std::cout << "This distance corresponds to " << total\_inches << " inches.\n";

// Convert a distance in inches to yards, feet, and inches std::cout << "Enter a distance in inches: ";

std::cin >> total\_inches;

feet = total\_inches / inches\_per\_foot; inches = total\_inches % inches\_per\_foot; yards = feet / feet\_per\_yard;

feet = feet % feet\_per\_yard;

std::cout << "This distance corresponds to "

<< yards << " yards "

<< feet << " feet "

<< inches << " inches." << std::endl;

}

The following is an example of typical output from this program:

Enter a distance as yards, feet, and inches with the three values separated by spaces: 9 2 11 This distance corresponds to 359 inches.

Enter a distance in inches: 359

This distance corresponds to 9 yards 2 feet 11 inches.

The first statement in main() defines three integer variables and initializes them with zero. They are type unsigned int because in this example the distance values cannot be negative. This is an instance where defining three variables in a single statement is reasonable because they are closely related.

The next statement outputs a prompt to std::cout for the input. We used a single statement spread over two lines, but it could be written as two separate statements as well:

std::cout << "Enter a distance as yards, feet, and inches "; std::cout << "with the three values separated by spaces: ";

When you have a sequence of << operators as in the original statement, they execute from left to right so the output from the previous two statements will be the same as the original.

The next statement reads values from cin and stores them in the variables yards, feet, and inches. The type of value that the >> operator expects to read is determined by the type of variable in which the value is to be stored. So, in this case, unsigned integers are expected to be entered. The >> operator ignores spaces, and the first space following a value terminates the operation. This implies that you cannot read and store spaces using the >> operator for a stream, even when you store them in variables that store characters. The input statement in the example could again also be written as three separate statements:

std::cin >> yards; std::cin >> feet; std::cin >> inches;

The effect of these statements is the same as the original.You define two variables, inches\_per\_foot and feet\_per\_yard, that you need to convert from yards, feet, and inches to just inches, and vice versa. The values for these are fixed, so you specify the variables as const. You could use explicit values for conversion factors in the code, but using const variables is much better because it is then clearer what you are doing. The const variables are also positive values, so you define them as type unsigned int. You could add U modifiers to the integer literals if you prefer, but there’s no need. The conversion to inches is done in a single assignment statement:

total\_inches = inches + inches\_per\_foot \* (yards\*feet\_per\_yard + feet);

The expression between parentheses executes first. This converts the yards value to feet and adds the feet value to produce the total number of feet. Multiplying this result by inches\_per\_foot obtains the total number of inches for the values of yards and feet. Adding inches to that produces the final total number of inches, which you output using this statement:

std::cout << "This distance corresponds to " << total\_inches << "inches.\n";

The first string is transferred to the standard output stream, std::cout, followed by the value of total\_inches. The string that is transferred to cout next has \n as the last character, which will cause the next output to start on the next line.

Converting a value from inches to yards, feet, and inches requires four statements:

feet = total\_inches / inches\_per\_foot; inches = total\_inches % inches\_per\_foot; yards = feet / feet\_per\_yard;

feet = feet % feet\_per\_yard;

We reuse the variables that stored the input for the previous conversion to store the results of this conversion. Dividing the value of total\_inches by inches\_per\_foot produces the number of whole feet, which you store in feet. The % operator produces the remainder after division, so the next statement calculates the number of residual inches, which is stored in inches. The same process is used to calculate the number of yards and the final number of feet.

Notice the use of whitespace to nicely outline these assignment statements. We could’ve written the same statements without spaces as well, but that simply does not read very fluently:

feet=total\_inches/inches\_per\_foot; inches=total\_inches%inches\_per\_foot; yards=feet/feet\_per\_yard; feet=feet%feet\_per\_yard;

We generally add a single space before and after each binary operator, as it promotes code readability.

Adding extra spaces to outline related assignments in a semitabular form doesn’t harm either.

There’s no return statement after this final output statement because it isn’t necessary. When the execution sequence runs beyond the end of main(), it is equivalent to executing return 0.

### The op= Assignment Operators

In Ex2\_02.cpp, there was a statement that you could write more economically:

feet = feet % feet\_per\_yard;

This statement could be written using an op= assignment operator. The op*= assignment operators*, or also *compound assignment operators*, are so called because they’re composed of an operator and an assignment operator =. You could use one to write the previous statement as follows:

feet %= feet\_per\_yard;

This is the same operation as the previous statement. In general, an op= assignment is of the following form:

lhs op= rhs;

lhs represents a variable of some kind that is the destination for the result of the operator. rhs is any expression. This is equivalent to the following statement:

lhs = lhs op (rhs);

The parentheses are important because you can write statements such as the following:

x \*= y + 1;

This is equivalent to the following:

x = x \* (y + 1);

Without the implied parentheses, the value stored in x would be the result of x \* y + 1, which is quite different. You can use a range of operators for op in the op= form of assignment. Table [2-3](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark4) shows the complete set, including some operators we’ll meet in Chapter 3.

***Table 2-3.*** *op= Assignment Operators*

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Operator** | **Operation** | **Operator** |
| Addition | += | Bitwise AND | &= |
| Subtraction | -= | Bitwise OR | |= |
| Multiplication | \*= | Bitwise exclusive OR | ^= |
| Division | /= | Shift left | <<= |
| Modulus | %= | Shift right | >>= |

Note that there can be no spaces between op and the =. If you include a space, it will be flagged as an error. You can use += when you want to increment a variable by some amount. For example, the following two statements have the same effect:

y = y + 1;

y += 1;

The shift operators that appear in the table, << and >>, look the same as the insertion and extraction operators that you have been using with streams. The compiler can figure out what << or >> means in a statement from the context. We’ll understand how it is possible that the same operator can mean different things in different situations later in the book.

## The sizeof Operator

We use the sizeof operator to obtain the number of bytes occupied by a type, by a variable, or by the result of an expression. Here are some examples of its use:

int height {74};

std::cout << "The height variable occupies " << sizeof height << " bytes." << std::endl; std::cout << "Type \"long long\" occupies " << sizeof(long long) << " bytes." << std::endl; std::cout << "The result of the expression height \* height/2 occupies "

<< sizeof(height \* height/2) << " bytes." << std::endl;

These statements show how we can output the size of a variable, the size of a type, and the size of the result of an expression. To use sizeof to obtain the memory occupied by a type, the type name must be between parentheses. We also need parentheses around an expression with sizeof. We don’t need parentheses around a variable name, but there’s no harm in putting them in. Thus, if we always use parentheses with sizeof, we can’t go wrong.

We can apply sizeof to any fundamental type, class type, or pointer type (we’ll learn about pointers in Chapter 5). The result that sizeof produces is of type size\_t, which is an unsigned integer type that is defined in the <cstddef> module of the Standard Library. Type size\_t is implementation defined, but if you use size\_t, our code will work with any compiler.

Now you should be able to create our own program to list the sizes of the fundamental integer types with our compiler.

## Incrementing and Decrementing Integers

You’ve seen how you can increment a variable with the += operator and we’re sure you’ve deduced that you can decrement a variable with -=. There are two other operators that can perform the same tasks. They’re called the *increment operator* and the *decrement operator*, ++ and --, respectively.

These operators are more than just other options. We’ll see a lot more of them, and we’ll find them to be quite an asset once you get further into C++. In particular, we’ll use them all the time when working with arrays and loops in Chapter 5. The increment and decrement operators are unary operators that you can apply to an integer variable. The following three statements that modify count have exactly the same effect:

int count {5};

count = count + 1; count += 1;

++count;

Each statement increments count by 1. Using the increment operator is clearly the most concise. The action of this operator is different from other operators that you’ve seen in that it directly modifies the value of its operand. The effect in an expression is to increment the value of the variable and then to use the incremented value in the expression. For example, suppose count has the value 5 and you execute this statement:

total = ++count + 6;

The increment and decrement operators execute before any other binary arithmetic operators in an expression. Thus, count will be incremented to 6, and then this value will be used in the evaluation of the expression on the right of the assignment. total will therefore be assigned the value 12.

We use the decrement operator in the same way:

total = --count + 6;

Assuming count is 6 before this statement, the -- operator will decrement it to 5, and then this value will be used to calculate the value to be stored in total, which will be 11.

We’ve seen how you place a ++ or -- operator before the variable to which it applies. This is called the *prefix form* of these operators. We can also place them after a variable, which is called the *postfix form*. The effect is a little different.

### Postfix Increment and Decrement Operations

The postfix form of ++ increments the variable to which it applies after its value is used in context. For example, you can rewrite the earlier example as follows:

total = count++ + 6;

With an initial value of 5 for count, total is assigned the value 11. In this case, count will be incremented to 6 only *after* being used in the surrounding expression. The preceding statement is thus equivalent to the following two statements:

total = count + 6;

++count;

In an expression such as a++ + b, or even a+++b, it’s less than obvious what you mean, or indeed what the compiler will do. These two expressions are actually the same, but in the second case you might have meant a + ++b, which is different—it evaluates to one more than the other two expressions. It would be clearer to write the preceding statement as follows:

total = 6 + count++;

Alternatively, you can use parentheses:

total = (count++) + 6;

The rules that we’ve discussed in relation to the increment operator also apply to the decrement operator. For example, suppose count has the initial value 5 and you write this statement:

total = --count + 6;

This results in total having the value 10 assigned. However, consider this statement:

total = 6 + count--;

In this instance, total is set to 11.

You should take care applying these operators to a given variable more than once in an expression.

Suppose count has the value 5 and you write this:

total = ++count \* 3 + count++ \* 5;

The result of this statement is undefined because the statement modifies the value of count more than once using increment operators. Even though this expression is undefined according to the C++

standard, this doesn’t mean that compilers won’t compile them. It just means that there is no guarantee for consistency in the results.

Now consider the following statement:

k = k++ + 5;

In the expression on the right, you’re incrementing the value of the variable that also appears on the left of the assignment operator. So you’re again modifying the value of k twice. Prior to C++17, the effects of such statements used to be undefined as well. With k equal to 10 prior to this statement, we have seen both 15 and 16 as possible outcomes with older compilers.

Informally, the C++17 standard added the rule that all side effects of the right side of an assignment (including compound assignments, increments, and decrements) are fully committed before evaluating the left side and the actual assignment. Meaning that with C++17, our statement should take k from 10 to 15.

Nevertheless, the precise rules of when precisely an expression is defined or undefined remain subtle, even in C++17 and beyond. Also, in our experience, not all compilers fully respect the new evaluation order rules of C++17 yet. So our advice remains unchanged:

Tip Modify a variable only once within a statement and access the prior value of the variable only to determine its new value—that is, do not attempt to read a variable again after it has been modified in the same statement.

The increment and decrement operators are usually applied to integers, particularly in the context of loops, as we’ll see in Chapter 5. We’ll see later in this chapter that you can apply them to floating-point variables too. In later chapters, we’ll explore how they can also be applied to certain other data types, in some cases with rather specialized (but very useful) effects.

## Defining Floating-Point Variables

We use floating-point variables whenever you want to work with values that are not integral. There are three floating-point data types, as shown in Table [2-4](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark5).

***Table 2-4.*** *Floating-Point Data Types*

Data Type Description

float Single precision floating-point values

double Double precision floating-point values

long double Double-extended precision floating-point values

Note You cannot use the unsigned or signed modifiers with floating-point types; floating-point types are always signed.

As explained in Chapter 1, the term *precision* refers to the number of significant digits in the mantissa. The types are in order of increasing precision, with float providing the lowest number of digits in the mantissa and long double the highest. The precision only determines the number of digits in the mantissa. The range of numbers that can be represented by a particular type is determined by the range of possible exponents.

The precision and range of values aren’t prescribed by the C++ standard, so what you get with each type depends on our compiler. And this, in turn, will depend on what kind of processor is used by our computer and the floating-point representation it uses. The standard does guarantee that type long double will provide a precision that’s no less than that of type double, and type double will provide a precision that is no less than that of type float.

Virtually all compilers and computer architectures today, however, use floating-point numbers and arithmetic as specified by the IEEE standard we introduced in Chapter 1. Normally, float thus provides seven decimal digits of precision (with a mantissa of 23 bits), double nearly 16 digits (52 bit mantissa). For long double, it depends on our compiler: with most major compilers it provides about 18 to 19 digits of precision (64-bit mantissa), but with others (most notably Microsoft Visual C++) long double is only as precise as double. Table [2-5](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark6) shows typical ranges of values that you can represent with the floating-point types on an Intel processor.

***Table 2-5.*** *Floating-Point Type Ranges*

|  |  |  |
| --- | --- | --- |
| **Type** | **Precision (Decimal Digits)** | **Range (+ or –)** |
| float | 7 | ±1.18 × 10-38 to ±3.4 × 1038 |
| double | 15 (nearly 16) | ±2.22 × 10-308 to ±1.8 × 10308 |
| long double | 18-19 | ±3.65 × 10-4932 to ±1.18 × 104932 |

The numbers of digits of precision in Table [2-5](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark6) are approximate. Zero can be represented exactly with each type, but values between zero and the lower limit in the positive or negative range can’t be represented, so the lower limits are the smallest possible nonzero values.

Here are some statements that define floating-point variables:

float pi {3.1415926f}; // Ratio of circle circumference to diameter double inches\_to\_mm {25.4};

long double root2 {1.4142135623730950488L}; // Square root of 2

As you see, you define floating-point variables just like integer variables. Type double is more than adequate in the majority of circumstances. You typically use float only when speed or data size is truly of the essence. If you do use float, though, you always need to remain vigilant that the loss of precision is acceptable for our application.

## Floating-Point Literals

We can see from the code fragment in the previous section that float literals have f (or F) appended and long double literals have L (or l) appended. Floating-point literals without a suffix are of type double. A floating-point literal includes either a decimal point or an exponent, or both; a numeric literal with neither is an integer.

An exponent is optional in a floating-point literal and represents a power of 10 that multiplies the value.

An exponent must be prefixed with e or E and follows the value. Here are some floating-point literals that include an exponent:

5E3 (5000.0) 100.5E2 (10050.0) 2.5e-3 (0.0025) -0.1E-3L (-0.0001L) .345e1F (3.45F)

The value between parentheses following each literal with an exponent is the equivalent literal without the exponent. Exponents are particularly useful when you need to express very small or very large values.

As always, most compilers will happily initialize floating-point variables with literals that lack a proper F or L suffix, or even with integer literals. If the literal value falls outside the representable range of the variable’s type, though, our compiler should at least issue a warning regarding a narrowing conversion.

## Floating-Point Calculations

We write floating-point calculations in the same way as integer calculations. Here’s an example:

const double pi {3.141592653589793}; // Circumference of a pizza divided by its diameter double a {0.2}; // Thickness of proper New York-style pizza (in inches)

double z {9}; // Radius of large New York-style pizza (in inches) double volume {}; // Volume of pizza - to be calculated

volume = pi\*z\*z\*a;

The modulus operator, %, can’t be used with floating-point operands, but all the other binary arithmetic operators that you have seen, +, -, \*, and /, can be. We can also apply the prefix and postfix increment and decrement operators, ++ and --, to a floating-point variable with essentially the same effect as for an integer; the variable will be incremented or decremented by 1.0.

### Mathematical Constants

In the previous example, we computed the volume of a New York-style pizza using a self-defined constant pi: const double pi {3.141592653589793}; // Circumference of a pizza divided by its diameter

But this number has other applications as well. You can use it to compute the volume of Neapolitan- style pizzas, California-style pizzas, Chicago-style pizzas, Greek pizzas, and so on—though not that of Sicilian- or Detroit-style pizzas. In fact, some even use this number outside of Italian cuisine as well. It is known among mathematicians as Archimedes' constant, and generally denoted by the Greek letter π.

Given its many uses in Italian cuisine and other scientific computations alike, it would therefore be a shame if every developer had to reinvent this wheel over and over again (or, rather, reinvent the ratio of this wheel’s circumference to its diameter over and over again). In the C++20 Standard Library, they therefore (finally) provide <numbers>, a module that defines this and several other common mathematical constants. Table [2-6](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark7) lists some of the most well-known examples. You can consult our Standard Library reference for the complete list.

***Table 2-6.*** *Examples of Numerical Constants in the* <numbers> *Module*

|  |  |  |
| --- | --- | --- |
| **Constant** | **Description** | **Approximate Value** |
| std::numbers::e | The base of the natural logarithm | 2.71828… |
| std::numbers::pi | π | 3.14159… |
| std::numbers::sqrt2 | Square root of 2 | 1.41421… |
| std::numbers::phi | The golden ratio constant φ | 1.618… |

All these constants have type double and are as accurate as possible for double precision floating-point numbers (so up to about 17 decimal digits). If we need these constants in computations with float or long double precision—especially the latter!—you should use expressions of the form std::numbers::pi\_ v<float> or std::numbers::sqrt2\_v<long double> instead. That is: you append \_v<T> to the constant’s name, substituting T with the desired floating-point type.[2](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark8)

Tip Prefer these pre-defined constants over self-defined ones. And if you do need to define some new constant, make sure to use the appropriate precision. All too often have we encountered legacy code that defines π as, say, 3.14159, with needlessly inaccurate results as a consequence!

### Mathematical Functions

The <cmath> Standard Library header defines a large selection of trigonometric and numerical functions that you can use in our programs. In this section, we’ll only discuss some of the functions that you are likely to use on a regular basis, but there are many, many more. The functions defined by <cmath> today truly range from the very basic to some of the most advanced mathematical functions (in the latter category, the C++17 standard, for instance, has recently added beauties such as cylindrical Neumann functions, associated Laguerre polynomials, and, our favorite, the Riemann zeta function).

Table [2-7](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark9) presents some of the most useful functions from this header. As always, all the function names defined are in the std namespace. All functions of <cmath> accept arguments that can be of any floating- point or integral type. Unless otherwise noted, the outcome has the same type as that of the (largest) input for floating-point arguments, and type double for integer arguments.

***Table 2-7.*** *Numerical Functions of the <cmath> Header*

Function Description

abs(arg) Computes the absolute value of arg. Unlike most <cmath> functions, abs() returns an integer type if arg is an integer.

ceil(arg) Computes a floating-point value that is the smallest integer greater than or equal to

arg, so std::ceil(2.5) produces 3.0 and std::ceil(-2.5) produces -2.0.

floor(arg) Computes a floating-point value that is the largest integer less than or equal to arg, so

std::floor(2.5) results in 2.0 and std::floor(-2.5) results in -3.0.

exp(arg) Computes the value of earg.

log(arg) Computes the natural logarithm (to base e) of arg. log10(arg) Computes the logarithm to base 10 of arg.

pow(arg1, arg2) Computes the value of arg1 raised to the power arg2, or arg1arg2. arg1 and arg2 can be integer or floating-point types. The result of std::pow(2, 3) is 8.0, std::pow(1.5f, 3) equals 3.375f, and std::pow(4, 0.5) is equal to 2.0.

sqrt(arg) Computes the square root of arg.

round(arg) lround(arg) llround(arg)

Rounds arg to the nearest integer. The result of round() is a floating-point number, even for integer inputs, whereas the results of lround() and llround() are of type long and long long, respectively. Halfway cases are rounded away from zero. In other words, std::lround(0.5) gives 1L, whereas std::round(-1.5f) gives -2.0f.

Besides these, <cmath> also provides all basic trigonometric functions (std::cos(), sin(), and tan()), as well as their inverse functions (std::acos(), asin(), and atan()). Angles are always expressed in radians.

Let’s look at some examples of how these are used. Here’s how you can calculate the cosine of an angle in radians:

double angle {1.5}; // In radians double cosine\_value {std::cos(angle)};

If the angle is in degrees, you can calculate the tangent by using a value for π to convert to radians:

const double pi\_degrees {180}; // Equivalent of pi radians in degrees double angle\_deg {60.0}; // Angle in degrees

double tangent {std::tan(std::numbers::pi \* angle\_deg / pi\_degrees)};

If we know the height of a church steeple is 100 feet and you’re standing 50 feet from its base, we can calculate the angle in radians of the top of the steeple like this:

double height {100.0}; // Steeple height in feet

double distance {50.0}; // Distance from base in feet double angle {std::atan(distance / height)}; // Result in radians

You can use this value in angle and the value of distance to calculate the distance from our toe to the top of the steeple:

double toe\_to\_tip {distance / std::sin(angle)};

Of course, fans of Pythagoras of Samos could obtain the result much more easily, like this:

double toe\_to\_tip {std::sqrt(std::pow(distance, 2) + std::pow(height, 2))};

Tip The problem with an expression of form std::atan(a / b) is that by evaluating the division a / b, you lose information about the sign of a and b. In our example this does not matter much, as both distance and height are positive, but in general you may be better off calling std::atan2(a, b) (also defined by <cmath>). Because atan2() knows the signs of both a and b, it is capable of properly reflecting this in the resulting angle. You can consult a Standard Library reference for the detailed specification.

One important caveat with <cmath> is that, because it has its origins in the C Standard Library, there is no guarantee that you can import this so-called *C header* as a module. To make its functionality available in our source files you should therefore use the following #include preprocessor directive instead:

#include <cmath>

Caution Unlike an import declaration, an #include directive must not be followed by a semicolon. Also, do not forget the leading number sign, #. Most preprocessor directives begin with a # sign.

Refer to online Appendix A for more information regarding #include and other preprocessor directives.

For now, you can recognize Standard Library headers that need to be included rather than imported by the “c” prefix in their name. Other examples of C headers that you may still use in C++ code today include

<cassert>, <cstddef>, and <cstdlib>.

Let’s try a floating-point example. Suppose that you want to construct a circular pond in which you will keep fish. Having looked into the matter, you know that you must allow two square feet of pond surface

area for every six inches of fish length. You need to figure out the diameter of the pond that will keep the fish happy. Here’s how you can do it:

// Ex2\_03.cpp

// Sizing a pond for happy fish import <iostream>;

import <numbers>; // For the pi constant

#include <cmath> // For the square root function

int main()

{

// 2 square feet pond surface for every 6 inches of fish

const double fish\_factor { 2.0/0.5 }; // Area per unit length of fish const double inches\_per\_foot { 12.0 };

double fish\_count {}; // Number of fish

double fish\_length {}; // Average length of fish

std::cout << "Enter the number of fish you want to keep: "; std::cin >> fish\_count;

std::cout << "Enter the average fish length in inches: "; std::cin >> fish\_length;

fish\_length /= inches\_per\_foot; // Convert to feet std::cout << '\n';

// Calculate the required surface area

const double pond\_area {fish\_count \* fish\_length \* fish\_factor};

// Calculate the pond diameter from the area

const double pond\_diameter {2.0 \* std::sqrt(pond\_area / std::numbers::pi)};

std::cout << "Pond diameter required for " << fish\_count << " fish is "

<< pond\_diameter << " feet.\n";

}

With input values of 20 fish with an average length of 9 inches, this example produces the following output:

Enter the number of fish you want to keep: 20 Enter the average fish length in inches: 9

Pond diameter required for 20 fish is 8.74039 feet.

We first define two const variables in main() that we’ll use in the calculation. Notice the use of a constant expression to specify the initial value for fish\_factor. We can use any expression for an initial value that produces a result of the appropriate type. We specify fish\_factor and inches\_per\_foot as const because their values are fixed and should not be altered.

Next, we define the fish\_count and fish\_length variables in which we’ll store the user input. Both have an initial value of zero. The input for the fish length is in inches, so we convert it to feet before we use it in the calculation for the pond. We use the /= operator to convert the original value to feet.

We define a variable for the area for the pond and initialize it with an expression that produces the required value:

const double pond\_area {fish\_count \* fish\_length \* fish\_factor};

The product of fish\_count and fish\_length gives the total length of all the fish in feet, and multiplying this by fish\_factor gives the required area for the pond in square feet. Once computed and initialized, the value of pond\_area will and should not be changed anymore, so we might as well declare the variable const to make that clear.

The area of a circle is given by the formula πr2, where r is the radius. We can therefore calculate the radius of the circular pond by dividing the area by π and calculating the square root of the result. The diameter is twice the radius, so the whole calculation is carried out by this statement:

const double pond\_diameter {2.0 \* std::sqrt(pond\_area / std::numbers::pi)};

We obtain the square root using the sqrt() function from the <cmath> header. Of course, we could calculate the pond diameter in a single statement like this:

const double pond\_diameter

{2.0 \* std::sqrt(fish\_count \* fish\_length \* fish\_factor / std::numbers::pi)};

This eliminates the need for the pond\_area variable so the program will be smaller and shorter. It’s debatable whether this is better than the original, though, because it’s far less obvious what is going on.

The last statement in main() outputs the result. Unless you’re an exceptionally meticulous pond enthusiast, however, the pond diameter has more decimal places than you need. We’ll look into how you can fix that later in this chapter.

### Invalid Floating-Point Results

So far as the C++ standard is concerned, the result of division by zero is undefined. Nevertheless, floating- point operations in most computers are implemented according to the IEEE 754 standard (also known as IEC 559). So in practice, compilers generally behave quite similarly when dividing floating-point numbers by zero. Details may differ across specific compilers, so consult our product documentation.

The IEEE floating-point standard defines special values having a binary mantissa of all zeroes and an exponent of all ones to represent +infinity or -infinity, depending on the sign. When you divide a

positive nonzero value by zero, the result will be +infinity, and dividing a negative value by zero will result in -infinity.

Another special floating-point value defined by this standard is called *not-a-number*, usually abbreviated to NaN. This represents a result that isn't mathematically defined, such as when you divide zero by zero or infinity by infinity. Any operation in which either or both operands are NaN results in NaN. Once an operation results in ±infinity, this will pollute all subsequent operations in which it participates as well.

Table [2-8](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark10) summarizes all the possibilities.

***Table 2-8.*** *Floating-Point Operations with NaN and ±infinity Operands*

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Result** | **Operation** | **Result** |
| ±value / 0 | ±infinity | 0 / 0 | NaN |
| ±infinity ± value | ±infinity | ±infinity / ±infinity | NaN |
| ±infinity \* value | ±infinity | infinity - infinity | NaN |
| ±infinity / value | ±infinity | infinity \* 0 | NaN |

value in the table is any nonzero value. You can discover how our compiler presents these values by plugging the following code into main():

double a{ 1.5 }, b{}, c{}; double result { a / b };

std::cout << a << "/" << b << " = " << result << std::endl;

std::cout << result << " + " << a << " = " << result + a << std::endl; result = b / c;

std::cout << b << "/" << c << " = " << result << std::endl;

We’ll see from the output when you run this how ±infinity and NaN look. One possible outcome is

this:

1.5/0 = inf

inf + 1.5 = inf 0/0 = -nan

**Tip** The easiest way to obtain a floating-point value that represents either infinity or NaN is using the facilities of the <limits> module of the Standard Library, which we discuss later in this chapter. That way you do not really have to remember the rules of how to obtain them through divisions by zero. To check whether a given number is either infinity or NaN, you should use the std::isinf() and std::isnan() functions provided by <cmath>—what to do with the Boolean-typed results of these functions will only become clear in Chapter 4, though.

### Pitfalls

We need to be aware of the limitations of working with floating-point values. It’s not difficult for the unwary to produce results that may be inaccurate or even incorrect. As we’ll recall from Chapter 1, common sources of errors when using floating-point values include the following:

* + - Many decimal values don’t convert exactly to binary floating-point values. The small errors that occur can easily be amplified in our calculations to produce large errors.
    - Taking the difference between two nearly identical values will lose precision. If we take the difference between two values of type float that differ in the sixth significant digit, we’ll produce a result that will have only one or two digits of accuracy. The other digits in the mantissa will be garbage. We already named this phenomenon *catastrophic cancellation*.
    - Working with values that differ by several orders of magnitude can lead to errors. An elementary example of this is adding two values stored as type float with seven digits of precision where one value is 108 times larger than the other. We can add the smaller value to the larger as many times as we like, and the larger value will be unchanged.

## Mixed Expressions and Type Conversion

We can write expressions involving operands of different types. For example, you could have defined the variable to store the number of fish in Ex2\_03 like this:

unsigned int fish\_count {}; // Number of fish

The number of fish is certainly an integer, so this makes sense. The number of inches in a foot is also integral, so you would want to define the variable like this:

const unsigned int inches\_per\_foot {12};

The calculation would still work OK in spite of the variables now being of differing types. Here’s an example (available in Ex2\_03A):

fish\_length /= inches\_per\_foot; // Convert to feet double pond\_area{fish\_count \* fish\_length \* fish\_factor};

Technically, all binary arithmetic operands require both operands to be of the same type. Where this is not the case, however, the compiler will arrange to convert one of the operand values to the same type as the other. These are called *implicit conversions*. The way this works is that the variable of a type with the more limited range is converted to the type of the other. The fish\_length variable in the first statement is of type double. Type double has a greater range than type unsigned int, so the compiler will insert a conversion for the value of inches\_per\_foot to type double to allow the division to be carried out. In the second statement, the value of fish\_count will be converted to type double to make it the same type as fish\_length before the multiply operation executes.

With each operation with operands of different types, the compiler chooses the operand with the type that has the more limited range of values as the one to be converted to the type of the other. In effect, it ranks the types in the following sequence, from high to low:

|  |  |  |
| --- | --- | --- |
| 1. long double | 4. unsigned long long | 7. long |
| 2. double | 5. long long | 8. unsigned int |
| 3. float | 6. unsigned long | 9. int |

The operand to be converted will be the one with the lower rank. Thus, in an operation with operands of type long long and type unsigned int, the latter will be converted to type long long. An operand of type char, signed char, unsigned char, short, or unsigned short is always converted to at least type int (remember this: it’ll become relevant in the next chapter!).

Implicit conversions can produce unexpected results. Consider these statements:

unsigned int x {20u}; int y {30};

std::cout << x - y << std::endl;

You might expect the output to be -10, but it isn’t. The output will be 4294967286! This is because the value of y is converted to unsigned int to match the type of x, so the result of the subtraction is an unsigned integer value. And -10 cannot be represented by an unsigned type. For unsigned integer types, going below

zero always wraps around to the largest possible integer value. That is, for a 32-bit unsigned int type, -1

becomes 232 - 1 or 4294967295, -2 becomes 232 - 2 or 4294967293, and so on. This then of course means

that -10 indeed becomes 232 - 10, or 4294967286.

Note The phenomenon where the result of a subtraction of unsigned integers wraps around to very large positive numbers is sometimes called *underflow*. In general, underflow is something to watch out for (we’ll encounter examples of this in later chapters). Naturally, the converse phenomenon exists as well, and is called *overflow*. Adding the unsigned char values 253 and 5, for instance, will not give 258—the largest value a variable of type unsigned char can hold is 255! Instead, the result will be 2, or 258 modulo 256.

**Caution** The outcome of over- and underflow is only defined for unsigned integers. With variables of signed integer types, the outcome of going beyond the bounds of what their type can represent is undefined—that is, it depends on our compiler and its target computer architecture.

The compiler will also insert an implicit conversion when the expression on the right of an assignment produces a value that is of a different type from the variable on the left. Here’s an example:

int y {};

double z {5.0};

y = z; // Requires an implicit narrowing conversion

The last statement requires a conversion of the value of the expression on the right of the assignment to allow it to be stored as type int. The compiler will insert a conversion to do this, but since this is a narrowing conversion, it may issue a warning message about possible loss of data.

You need to take care when writing integer operations with operands of different types. Don’t rely on implicit type conversion to produce the result you want unless you are certain it will do so. If you are not sure, what you need is an *explicit type conversion*, also called an *explicit cast*.

## Explicit Type Conversion

To explicitly convert the value of an expression to a given type, you write the following:

static\_cast<type\_to\_convert\_to>(expression)

The static\_cast keyword reflects the fact that the cast is checked statically, that is, when the code is compiled. Later, when you get to deal with classes, we’ll meet *dynamic casts*, where the conversion is

checked dynamically, that is, when the program is executing. The effect of the cast is to convert the value that results from evaluating expression to the type that you specify between the angle brackets. The expression can be anything from a single variable to a complex expression involving lots of nested parentheses. You could eliminate the warning that arises from the assignment in the previous section by writing it as follows:

y = static\_cast<int>(z); // Never a compiler warning this time

By adding an explicit cast, you signal the compiler that a narrowing conversion is intentional.

If the conversion is not narrowing, you’d rarely add an explicit cast. Here’s another example of the use of static\_cast<>():

double value1 {10.9};

double value2 {15.9};

int whole\_number {static\_cast<int>(value1) + static\_cast<int>(value2)}; // 25

The initializing value for whole\_number is the sum of the integral parts of value1 and value2, so they’re each explicitly cast to type int. whole\_number will therefore have the initial value 25. Note that as with integer division, casting from a floating-point type to an integral type uses *truncation*. That is, it simply discards the entire fractional part of the floating-point number.

Tip The std::round(), lround(), and llround() functions from <cmath> allow you to round floating- point numbers to the nearest integer. In many cases this is better than the truncation used by (implicit or explicit) casting.

The casts in our previous example do not affect the values stored in value1 and value2, which will remain as 10.9 and 15.9, respectively. The values 10 and 15 produced by the casts are just stored

temporarily for use in the calculation and then discarded. Although both casts cause a loss of information, the compiler always assumes you know what you’re doing when you explicitly specify a cast.

Of course, the value of whole\_number would be different if you wrote this:

int whole\_number {static\_cast<int>(value1 + value2)}; // 26

The result of adding value1 and value2 will be 26.8, which results in 26 when converted to type int.

This is again different from what you would obtain if you instead wrote this:

int whole\_number {static\_cast<int>(std::round(value1 + value2))}; // 27

As always with braced initializers, without the explicit type conversion in this statement, the compiler will either refuse to insert or at least warn about inserting implicit narrowing conversions.

Generally, the need for explicit casts should be rare, particularly with basic types of data. If you have to include a lot of explicit conversions in our code, it’s often a sign that you could choose more suitable types for our variables. Still, there are circumstances when casting is necessary, so let’s look at a simple example. This example converts a length in yards as a decimal value to yards, feet, and inches:

// Ex2\_04.cpp

// Using explicit type conversions import <iostream>;

int main()

{

const unsigned feet\_per\_yard {3}; const unsigned inches\_per\_foot {12};

const unsigned inches\_per\_yard { feet\_per\_yard \* inches\_per\_foot };

double length {}; // Length as decimal yards unsigned int yards{}; // Whole yards

unsigned int feet {}; // Whole feet unsigned int inches {}; // Whole inches

std::cout << "Enter a length in yards as a decimal: ";

std::cin >> length;

// Get the length as yards, feet, and inches

yards = static\_cast<unsigned int>(length);

feet = static\_cast<unsigned int>((length - yards) \* feet\_per\_yard);

inches = static\_cast<unsigned int>(length \* inches\_per\_yard) % inches\_per\_foot;

std::cout << length << " yards converts to "

<< yards << " yards "

<< feet << " feet "

<< inches << " inches." << std:: endl;

}

This is typical output from this program:

Enter a length in yards as a decimal: 2.75

2.75 yards converts to 2 yards 2 feet 3 inches.

The first three statements in main() define unsigned integer constants to convert between yards, feet, and inches. You declare these as const to prevent them from being modified accidentally. The variables that will store the results of converting the input to yards, feet, and inches are of type unsigned int and initialized with zero.

The statement that computes the whole number of yards from the input value is as follows:

yards = static\_cast<unsigned int>(length);

The cast discards the fractional part of the value in length and stores the integral result in yards. You could omit the explicit cast here and leave it to the compiler to take care of, but it’s always better to write an explicit cast in such cases. If you don’t, it’s not obvious that you realized the need for the conversion and the potential loss of data. Many compilers will then issue a warning as well.

You obtain the number of whole feet with this statement:

feet = static\_cast<unsigned int>((length - yards) \* feet\_per\_yard);

Subtracting yards from length produces the fraction of a yard in the length as a double value. The compiler will arrange for the value in yards to be converted to type double for the subtraction. The value of feet\_per\_yard will then be converted to double to allow the multiplication to take place, and finally the explicit cast converts the result from type double to type unsigned int.

The final part of the calculation obtains the residual number of whole inches:

inches = static\_cast<unsigned int>(length \* inches\_per\_yard) % inches\_per\_foot;

The explicit cast applies to the total number of inches in length, which results from the product of length and inches\_per\_yard. Because length is type double, the const value will be converted implicitly to type double to allow the product to be calculated. The remainder after dividing the integral number of inches in length by the number of inches in a foot is the number of residual inches.

### Old-Style Casts

Prior to the introduction of static\_cast<> into C++ around 1998—so a very, very long time ago—explicit casts were written like this:

(type\_to\_convert\_to)expression

The result of expression is cast to the type between the parentheses. For example, the statement to calculate inches in the previous example could be written like this:

inches = (unsigned int)(length \* inches\_per\_yard) % inches\_per\_foot;

This type of cast is a remnant of the C language and is therefore also referred to as a *C-style cast*. There are several kinds of casts in C++ that are now differentiated, but the old-style casting syntax covers them all. Because of this, code using the old-style casts is more prone to errors. It isn’t always clear what you

intended, and you may not get the result you expected. Also, the round parentheses blend in too much with the surrounding (compound) expressions—the static\_cast<>() operator is far easier to spot visually.

Therefore:

* **Tip** We’ll still see old-style casts at times because it’s still part of the language, but we strongly recommend you do not use them in new code. One should never use C-style casts in C++ code anymore. Period. That is why this is also the last time we mention this syntax in this book….

## Formatting Strings

Earlier in this chapter we wrote you a highly scientific program to recommend a suitable diameter for our new fish pond. For 20 fish with an average length of nine inches, Ex2\_03A produced the following output:

Pond diameter required for 20 fish is 8.74039 feet.

Extremely insightful. Only: we do not know about you but whenever we dig ponds we rarely measure their size up to one thousandth of an inch (one hundredth of a millimeter for our metric readers). Most fish really aren’t that fussy. So why output the diameter which such a high precision?

Of course you could attempt to round each number up to the desired number of decimals yourself, before sending it to the output stream (that would make for an interesting exercise, actually). But surely there are better solutions? Let’s explore.

### Formatting Stream Output

You can change the way an output stream formats data using *stream manipulators*. You apply a stream manipulator by inserting it into the stream using its << operator together with the data itself. With the setprecision() manipulator of the <iomanip> module, for instance, you can tweak the number of decimal digits the stream uses to format floating-point numbers. Here is an example.

std::cout << "Pond diameter required for " << fish\_count << " fish is "

<< std::setprecision(2) // Use two significant digits

<< pond\_diameter << " feet.\n"; // Output value is 8.7

The standard <ios> and <iomanip> modules define many more stream manipulators. Examples include std::hex (produces hexadecimal numbers), std::scientific (enables exponent notation for floating- point numbers), and std::setw() (used to format tabular data). We do not discuss stream manipulators in detail here, however, because in this book we will use std::format() instead. The reason is that, compared to stream manipulators, this C++20 function is slightly more powerful, tends to result in more compact and readable code, and is often faster in execution. You can consult a Standard Library reference to find out more about stream manipulators. Doing so should be a walk in the park once you know std::format(), though, as formatting with manipulators is based on the exact same concepts as formatting with std::format() (width, precision, fill characters, and so on).

### String Formatting with std::format()

After importing C++20’s <format> module, you can replace the output statement of Ex2\_03A with the following one (note that in Ex2\_03A the type of fish\_count was changed from double to unsigned int):

std::cout << std::format("Pond diameter required for {} fish is {} feet.\n", fish\_count, pond\_diameter);

The first argument to std::format() is always the *format string*. This string contains any number of *replacement fields*, each surrounded with a pair of curly braces, {}. The format string is followed by zero or more additional *arguments*, generally one per replacement field. In our example there are two: fish\_count and pond\_diameter. The result of std::format() then is a copy of the format string where each field is replaced with a textual representation of one of these arguments.

In our initial example both replacement fields are empty. This means that std::format() will match them with the other two arguments, fish\_count and pond\_diameter, in left-to-right order,and that it will use its default formatting rules to convert these values into text. Here is the expected output (you can find the complete program in Ex2\_03B.cpp*):*

Pond diameter required for 20 fish is 8.740387444736633 feet.

A good start. Only: we seem to have made matters worse. We’re now digging ponds with sub- femptometer precision—a precision that is about a million times smaller than our average atom. The reason is that the default precision of std::format() is such that it ensures so-called *lossless round-trips*. This means that if you convert any formatted number back into a value of the same type, you by default obtain the exact same value as the one you started with. For a double this can result in strings with up to 16 decimal digits[3](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark11), as in our example.

3 Not all doubles will be formatted using 16 digits, though. The default formatting of the double value 1.5, for instance, would just be "1.5".

But of course you can override the default formatting. You do so by adding a *format specifier* between the curly braces of a replacement field. We discuss this in the next subsection. After that, we’ll explain you how to override the default left-to-right order, and how to output the same value more than once.

#### Format Specifiers

A format specifier is a seemingly cryptic sequence of numbers and characters, introduced by a colon, telling std::format() how you would like the corresponding data to be formatted. To tune floating-point precision, for instance, you add, after the mandatory colon, a dot followed by an integer number:

std::cout << std::format("Pond diameter required for {} fish is {:.2} feet.\n", fish\_count, pond\_diameter);

By default, this integer specifies the *total number of significant digits* (2 in our example), counting digits both before and after the decimal point. The result thus becomes (see Ex2\_03C.cpp):

Pond diameter required for 20 fish is 8.7 feet.

You can instead make the precision specify the number of digits *after* the decimal point—the *number of decimal places* in other words—by enabling so-called “fixed-point” formatting of floating-point numbers. You do so by appending the letter f to the format specifier. If you replace {:.2} with {:.2f} in our running example, it produces the following output:

Pond diameter required for 20 fish is 8.74 feet.

**Note** You obtain the equivalent with stream manipulators by inserting ‘<< std::precision(2) << std::fixed’ into an output stream. See how much more compact ‘{:.2f}’ is?

Here is the (slightly simplified) general form of the format specifiers for fields of fundamental and string types:

[[fill]align][sign][#][0][width][.precision][type]

The square brackets are for illustration purposes only and mark optional *formatting options*. Not all options are applicable to all field types. precision, for instance, is only applicable to floating-point numbers (as you already know) and strings (where it defines how many characters will be used from the string).

Caution If you specify an unsupported formatting option, or if you make any syntactical mistake in one of the format specifiers, std::format() will fail. To report this failure, std::format() raises what is known as an *exception*. If such an exception occurs, and there is no error handling code in place; our entire program

instantly terminates with an error. You can give this a try by replacing the format string in Ex2\_03C.cpp with the following (we added a precision to the first format specifier):

std::cout << std::format("Pond diameter required for {:.2} fish is {:.2} feet.\n",

fish\_count, pond\_diameter);

Because fish\_count is an integer, you are not allowed to specify a precision for that field. And so this invocation of std::format(), and by extension our entire program, will fail when executed.

Even though we only explain how to handle exceptions in Chapter 16, you need a means to debug failing std::format() statements. After all: you will be using std::format() in examples and exercises long before Chapter 16, and finding out which tiny little mistake you made in a format specifier can be tricky at times. Hence, the following tip:

**Tip** To debug a failing std::format() expression, you can put the corresponding statement inside a so-called try-catch block as follows (this code is available in Ex2\_03D.cpp):

try

{

std::cout << std::format("Pond diameter required for {:.2} fish is {:.2} feet.\n",

fish\_count, pond\_diameter);

}

catch (const std::format\_error& error)

{

std::cout << error.what(); // Outputs "precision not allowed for this argument type"

}

The program will now no longer fail and instead will output a diagnostic message that should aid you in fixing the buggy format specifier. This try-catch snippet uses several language elements you do not know yet (references, exceptions, etc.), but you can easily copy-paste it from Ex2\_03D.cpp whenever you need it.

##### Formatting Tabular Data

The following formatting options (highlighted in black) allow you to control the width and alignment of each field. We regularly use them in this book to output text that resembles a table.

[[fill]align][sign][#][0][width][.precision][type]

width is a positive integer that defines the *minimum field width*. If needed, extra characters are inserted into the formatted field to reach this minimum width. Which characters are inserted and where depends both on the field’s type and which other formatting options are present:

* + - For numeric fields, if the width option is preceded with 0 (zero), extra 0 characters are inserted before the number’s digits, but after any sign character (+ or -) or prefix sequence (such as 0x for hexadecimal numbers: see later).
    - Otherwise, the so-called *fill character* is inserted. The default fill character is a space, but you can override this with the fill formatting option. The align option determines where this fill character is inserted. A field can be *left-aligned* (<),

*right-aligned* (>), or *centered* (^). The default alignment depends on the field’s type.

You cannot specify the fill character without specifying an alignment as well. Note also that the fill, align, and 0 options have no effect unless you also specify a width.

Still with us? Or is our head starting to spin? We know: we just threw a lot of formatting options at you all at once. High time we made it all concrete with some code. We recommend that you take our time to analyze this next example (perhaps you can try to predict what the output will look like?), and to play with its format specifiers to see the effects of the different formatting options.

// Ex2\_05.cpp

// The width, alignment, fill, and 0 formatting options of std::format() import <iostream>;

import <format>;

int main()

{

// Default alignment: right for numbers, left otherwise

std::cout << std::format("{:7}|{:7}|{:7}|{:7}|{:7}\n", 1, -.2, "str", 'c', true);

// Left and right alignment + custom fill character

std::cout << std::format("{:\*<7}|{:\*<7}|{:\*>7}|{:\*>7}|{:\*>7}\n", 1,-.2,"str",'c',true);

// Centered alignment + 0 formatting option for numbers

std::cout << std::format("{:^07}|{:^07}|{:^7}|{:^7}|{:^7}\n", 1, -.2, "str", 'c', true);

}

Because we use the same width, 7, for all fields, the result resembles a table. More specifically, the result looks as follows:

1| -0.2|str |c |true 1\*\*\*\*\*\*|-0.2\*\*\*|\*\*\*\*str|\*\*\*\*\*\*c|\*\*\*true 0000001|-0000.2| str | c | true

From the first line, you see that the default alignment of fields is not always the same. By default numeric fields align to the right, whereas string, character, and Boolean fields align to the left.

For the second line, we mirrored all default alignments using the < and > alignment options. We also set the fill character of all fields to \*.

The third line showcases two more options: centered alignment (^) and the special 0 filling option for numeric fields (0 may only be used for numeric fields).

##### Formatting Numbers

These four remaining options are mostly relevant to numeric fields:

[[fill]align][sign][#][0][width][.precision][type]

precision you know already from before, but there is much more to formatting numbers than that. Like in the previous section there is a lot to take in here, and we will make it more concrete with an example after.

* Fixed-point formatting (f; see earlier) is not the only supported type option for formatting floating-point numbers. Others are *scientific formatting* (e), *general formatting* (g), and even *hexadecimal formatting* (a).
* Integer fields are mostly equivalent to floating-point fields, except that the precision option is not allowed. Supported types include *binary formatting* (b) and *hexadecimal formatting* (x).
  + Adding a # character toggles the so-called *alternate form*. For integers, the alternate form adds a *base prefix* (0x or 0b); for floating-point numbers it causes the output to always contain a decimal-point, even if no digit follows it.
  + The E, G, A, B, and X formatting types are equivalent to their lowercase equivalents, except that any letter in the output is capitalized. This includes base prefixes (0X or 0B), hexadecimal digits (A through F), and values for infinity and NaN values (INF and NAN instead of inf and nan).
  + The sign option is a single character that determines what is printed in front of non-negative numbers. Negative numbers are always preceded with a – character. Possible values for sign include + (instructs to put a + in front of non-negative numbers) and a space character (put a space in front of non-negative numbers).

The following example should clarify these points (the different bullets from earlier are illustrated more or less in order):

// Ex2\_06.cpp

// Formatting numeric values with std::format() import <iostream>;

import <format>; import <numbers>;

int main()

{

const double pi = std::numbers::pi;

std::cout << std::format("Default: {:.2}, fixed: {:.2f}, scientific: {:.2e}, "

"general: {:.2g}\n", pi, pi, pi, pi);

std::cout << std::format("Default: {}, binary: {:b}, hex.: {:x}\n", 314, 314, 314);

std::cout << std::format("Default: {}, decimal: {:d}, hex.: {:x}\n", 'c', 'c', 'c');

std::cout << std::format("Alternative hex.: {:#x}, binary: {:#b}, HEX.: {:#X}\n",

314, 314, 314);

std::cout << std::format("Forced sign: {:+}, space sign: {: }\n", 314, 314); std::cout << std::format("All together: {:\*<+10.4f}, {:+#09x}\n", pi, 314);

}

Here is the expected outcome:

Default: 3.1, fixed: 3.14, scientific: 3.14e+00, general: 3.1

Default: 314, binary: 100111010, hex.: 13a

Default: c, decimal: 99, hex.: 63

Alternative hex.: 0x13a, binary: 0b100111010, HEX.: 0X13A Forced sign: +314, space sign: 314

All together: +3.1416\*\*\*, +0x00013a

You should be able to understand most of this yourself. Only the different formatting options for floating-point numbers may warrant some more explanation.

*Scientific formatting* (e or E) always adds an exponent component to the output, precisely like we did in Chapter 1 when explaining floating-point numbers. You can use the same exponent notation also when entering floating-point literals. Exponents are mostly interesting either for very small numbers (such as 2.50e-11, the radius of a hydrogen atom), or for very large numbers (such as 1.99e30, the mass of the sun).

For more regular-sized numbers, always adding an exponent component is typically not ideal, as can be seen from our example (where pi has exponent +00). This is where the *general formatting* option (g or G) comes in handy: by default it is equivalent to fixed-point formatting (f or F), only for very small or large numbers (the exact heuristic is not that important), it switches to scientific formatting (e or E) and adds the exponent. You can give it a run by assigning different values (large or small) to the pi constant in Ex2\_06.

#### Argument Indexes

Being the perceptive, critical reader that you are, you have of course been wondering why we put a colon in front of each format specifier. The reason, of course, is that there is still something that you can add to the left of this colon as well. And that something is an *argument index*. Until now all replacement fields were matched with arguments of std::format() in a left-to-right order. You can override this, however, by adding an argument index to each replacement field. To illustrate, consider the following variation of our favorite format() expression.

std::cout << std::format(

"{1:.2f} feet is the diameter required for a pond with {0} fishes.\n", fish\_count, pond\_diameter

);

By rephrasing our sentence we have swapped the order of the two replacement fields. To then keep the order of the other two arguments, fish\_count and pond\_diameter, the same, we had to add an argument index to both replacement fields. This index is separated from the format specifier by a colon. If no format specifier is required, the index is not followed by a colon.

The semantics of an argument index should be rather straightforward. The only thing to watch out for is that you have to start counting at zero: the first argument, fish\_count, has index 0, the second index 1, and so on. This may seem odd as first, but that is just the way indexes always work in C++. In Chapter 5, for instance, we’ll encounter the same phenomenon again when working with arrays.

Note Of course, in our toy example the argument indexes are not really required. You could instead simply swap the order of the fish\_count and pond\_diameter arguments as well. It would even make the code clearer to do so. But the need for out-of-order formatting does arise in real-life scenarios, where you, as a developer, are not necessarily the one supplying the (final) format string. Instead, a technical writer may be entering user- facing text (where the order may change for grammatical reasons), or a translator may be providing translated messages (where the order may even differ between different languages). It would be a shame if a program had to be reworked each time one of these texts changes.

Besides reordering fields, you can also use argument indexes to format the same input value more than once. The second output statement of Ex2\_06, for instance, could be written as follows (see Ex2\_06B):

std::cout << std::format("Default: {0}, binary: {0:b}, hex.: {0:x}\n", 314);

## Finding the Limits

You have seen typical examples of the upper and lower limits for various types. The <limits> Standard Library module makes this information available for all the fundamental data types so you can access this for our compiler. Let’s look at an example. To display the maximum value you can store in a variable of type double, you could write this:

std::cout << "Maximum value of type double is " << std::numeric\_limits<double>::max();

The expression std::numeric\_limits<double>::max() produces the value you want. By putting different type names between the angled brackets, you can obtain the maximum values for other data types. You can also replace max() with min() to get the minimum value that can be stored, but the meaning of minimum is different for integer and floating-point types. For an integer type, min() results in the true

minimum, which will be a negative number for a signed integer type. For a floating-point type, min() returns the minimum positive value that can be stored.

**Caution** std::numeric\_limits<double>::min() typically equals 2.225e-308, an extremely tiny *positive* number. So, for floating-point types, min() does not give you the complement of max(). To get the lowest *negative* value a type can represent, you should use lowest() instead. For instance, std::numeric\_ limits<double>::lowest() equals -1.798e+308, a hugely negative number. For integer types, min() and lowest() always evaluate to the same number.

The following program will display the maximums and minimums for some of the numerical data types:

// Ex2\_07.cpp

// Finding maximum and minimum values for data types import <iostream>;

import <limits>; import <format>;

int main()

{

std::cout

<< std::format("The range for type short is from {} to {}\n", std::numeric\_limits<short>::min(), std::numeric\_limits<short>::max())

<< std::format("The range for type unsigned int is from {} to {}\n", std::numeric\_limits<unsigned int>::min(), std::numeric\_limits<unsigned int>::max())

<< std::format("The range for type long is from {} to {}\n", std::numeric\_limits<long>::min(), std::numeric\_limits<long>::max())

<< std::format("The positive range for type float is from {} to {}\n", std::numeric\_limits<float>::min(), std::numeric\_limits<float>::max())

<< std::format("The full range for type float is from {} to {}\n", std::numeric\_limits<float>::lowest(), std::numeric\_limits<float>::max())

<< std::format("The positive range for type double is from {} to {}\n", std::numeric\_limits<double>::min(), std::numeric\_limits<double>::max())

<< std::format("The positive range for type long double is from {} to {}\n", std::numeric\_limits<long double>::min(), std::numeric\_limits<long double>::max());

}

You can easily extend this to include additional numeric types. On our test system, the results of running the program are as follows:

The range for type short is from -32768 to 32767

The range for type unsigned int is from 0 to 4294967295

The range for type long is from -9223372036854775808 to 9223372036854775807 The positive range for type float is from 1.17549e-38 to 3.40282e+38

The full range for type float is from -3.40282e+38 to 3.40282e+38

The positive range for type double is from 2.22507e-308 to 1.79769e+308

The positive range for type long double is from 3.3621e-4932 to 1.18973e+4932

Note In Ex2\_07, we used std::format() even though we didn’t really need any of its powerful text formatting facilities (such as field width, precision, alignment, etc.). We did so to illustrate how std::format(), even then, offers advantages. Just count the " delimeters and << operators you have to type when streaming the same output directly to std::cout:

<< "The range for type short is from " << std::numeric\_limits<short>::min()

<< " to " << std::numeric\_limits<short>::max() << '\n';

More importantly, interleaving text and invocations of << like this also makes our code harder to read. That is: reconstructing the output text becomes harder, as our eyes have to scan the entire statement for snippets of text. With the std::format() version, on the other hand, you can instantly see that the output will be of the form "The range for type short is from ... to ...\n".

### Finding Other Properties of Fundamental Types

You can retrieve many other items of information about various types. The number of binary digits, or bits, for example, is returned by this expression:

std::numeric\_limits<type\_name>::digits

type\_name is the type in which you’re interested. For floating-point types, we’ll get the number of bits in the mantissa. For signed integer types, we’ll get the number of bits in the value, that is, excluding the sign bit. You can also find out what the range of the exponent component of floating-point values is, whether a type is signed or not, and so on. You can consult a Standard Library reference for the complete list.

Before we move on, though, there are two more numeric\_limits<> functions we still want to introduce.

We promised you earlier that we would. To obtain the special floating-point values for infinity and not-a- number (NaN), you should use expressions of the following form:

float positive\_infinity = std::numeric\_limits<float>::infinity(); double negative\_infinity = -std::numeric\_limits<double>::infinity();

long double not\_a\_number = std::numeric\_limits<long double>::quiet\_NaN();

None of these expressions would compile for integer types, nor would they compile in the unlikely event that the floating-point types that our compiler uses do not support these special values. Besides quiet\_NaN(), there’s a function called signaling\_NaN()—so not loud\_NaN() or noisy\_NaN(). The difference between the two is outside the scope of this brief introduction. If you’re interested, you can always consult our Standard Library documentation.

## Working with Character Variables

Variables of type char are used primarily to store a code for a single character and occupy one byte. The C++ standard doesn’t specify the character encoding to be used for the basic character set, so in principle this is down to the particular compiler, but it’s usually ASCII.

You define variables of type char in the same way as variables of the other types that you’ve seen. Here’s an example:

char letter; // Uninitialized - so junk value

char yes {'Y'}, no {'N'}; // Initialized with character literals char ch {33}; // Integer initializer equivalent to '!'

You can initialize a variable of type char with a character literal between single quotes or by an integer. An integer initializer must be within the range of type char—remember, it depends on the compiler whether it is a signed or unsigned type. Of course, you can specify a character as one of the escape sequences you saw in Chapter 1.

There are also escape sequences that specify a character by its code expressed as either an octal or a hexadecimal value. The escape sequence for an octal character code is one to three octal digits preceded by a backslash. The escape sequence for a hexadecimal character code is one or more hexadecimal digits

preceded by \x. You write either form between single quotes when you want to define a character literal. For example, the letter 'A' could be written as hexadecimal '\x41' in ASCII. Obviously, you could write codes that won’t fit within a single byte, in which case the result is implementation defined.

Variables of type char are numeric; after all, they store integer codes that represent characters. They can therefore participate in arithmetic expressions, just like variables of type int or long. Here’s an example:

char ch {'A'};

char letter {ch + 2}; // letter is 'C'

++ch; // ch is now 'B'

ch += 3; // ch is now 'E'

When you write a char variable using cout or format(), it is by default output as a character, not as an integer. If you want to see it as a numerical value with cout, you have no choice but to cast it to another integer type first. With format(), you can instead format the character using binary (b), decimal (d), or hexadecimal (x) formatting. Here’s an example:

std::cout << std::format("ch is '{0}' which is code {0:#x}\n", ch);

This produces the following output:

ch is 'E' which is code 0x45

We used argument indexes (0) to format the same character value twice—once with default formatting, and once with the alternate form (#) of lowercase hexadecimal formatting (x).

When you use >> to read from a stream into a variable of type char, the first nonwhitespace character will be stored. This means you can’t read whitespace characters in this way; they’re simply ignored. Further, you can’t read a numerical value into a variable of type char; if you try, the character code for the first digit will be stored.

### Working with Unicode Characters

ASCII is generally adequate for national language character sets that use Latin characters. However, if you want to work with characters for multiple languages simultaneously or if you want to handle character sets for many non-English languages, 256 character codes doesn’t go nearly far enough, and Unicode is the answer. You can refer to Chapter 1 for an introduction to Unicode and character encodings.

Type wchar\_t is a fundamental type intended for character sets where a single character does not fit into one byte. Hence its name: wchar\_t derives from *w*ide *char*acter, because the character is “wider” than the usual one-byte character. By contrast, type char is referred to as “narrow” because of the limited range of character codes that are available.

You define wide-character literals in a similar way to literals of type char, but you prefix them with L. Here’s an example:

wchar\_t z {L'Z'};

This defines z as type wchar\_t and initializes it to the wide-character representation for Z.

Our keyboard may not have keys for representing other national language characters, but you can still create them using hexadecimal notation. Here’s an example:

wchar\_t cc {L'\x00E7'}; // Initialized with the wide-character encoding of c-cedilla (ç)

The value between the single quotes is an escape sequence that specifies the hexadecimal representation of the character code. The backslash indicates the start of the escape sequence, and x or X after the backslash signifies that the code is hexadecimal (see also Chapter 1).

The problem with type wchar\_t is that its size is highly implementation-specific, as is the encoding used by the compiler for wide-character literals. Both generally correspond to the preferred wide-character encoding of the target platform. For Windows, wchar\_t is therefore typically 16-bit wide and wide-character literals are encoded with UTF-16; for most other platforms, wchar\_t is 32-bit wide and wide-character literals are encoded with UTF-32. While that makes wchar\_t perfectly suited for interacting with native Unicode APIs, it does not lend itself to writing code that is portable across different platforms.

Unless you are interacting with wchar\_t-based APIs, it is therefore recommended to use types char8\_t, char16\_t, or char32\_t instead. Values of these types are intended to store characters encoded as UTF-8, UTF-16, or UTF-32, respectively, and their sizes are the same on all common platforms (we’re sure you can guess what these sizes are…). Here are some example variables of these three types:

char8\_t yen {u8'\x00A5'}; // Initialized with UTF-8 code for the yen sign (¥) char16\_t delta {u'\x0394'}; // Initialized with UTF-16 code for Greek Delta (Δ)

char32\_t ya {U'\x044f'}; // Initialized with UTF-32 code for cyrillic letter ya (я)

The prefixes u8, u, and U to the literals indicate that they are UTF-8, UTF-16, and UTF-32, respectively. If our editor and compiler have the capability to process Unicode characters, and you know how to type such letters on our keyboard, you can define Unicode character variables without the escape sequences as well (see Chapter 1):

wchar\_t cc {L'ç'}; char8\_t yen {u8'¥'}; char16\_t delta {u16'Δ'}; char32\_t ya {U'я'};

Because UTF-8 and UTF-16 are variable-width encodings, not all letters can be represented by a single character. The Greek letter Δ, for instance, needs a sequence of two bytes to be encoded in UTF-8.

char8\_t delta8 {u8'Δ'}; /\* Error: Δ (code point U+0394) encoded as 2 UTF-8 code units \*/

Note The char8\_t type was introduced in C++20. Most legacy code and libraries will therefore still use type char to represent UTF-8 encoded letters. This can be confusing, at times, because the same type is used to store narrow (generally ASCII-encoded) letters. New code should therefore shift toward using char8\_t as much as possible.

The Standard Library provides standard input and output streams wcin and wcout for reading and writing characters of type wchar\_t, but there is no provision with the library for handling char8\_t, char16\_t, or char32\_t character data. We briefly return to processing Unicode characters in Chapter 7, where we discuss strings.

## The auto Keyword

We use the auto keyword to indicate that the compiler should deduce the type of a variable. Here are some examples:

auto m {10}; // m has type int

auto n {200UL}; // n has type unsigned long auto p {std::numbers::pi}; // p has type double

The compiler will deduce the types for m, n, and p from the initial values you supply. You can use functional or assignment notation with auto for the initial value as well:

auto m = 10; // m has type int

auto n = 200UL; // n has type unsigned long auto p(std::numbers::pi); // p has type double

Having said that, this is not really how the auto keyword is intended to be used. When defining variables of fundamental types, you often might as well specify the type explicitly. We’ll meet the auto keyword again later in the book where it is more appropriately and much more usefully applied.

Caution We need to be careful when using braced initializers with the auto keyword. For example, suppose we write this (notice the equals sign!):

auto m = { 10 }; // m has type std::initializer\_list<int>

Then the type deduced for m will not be int, but instead will be std::initializer\_list<int>. To give some context, this is the same type we would get if we’d use a list of elements between the braces:

auto list = { 1, 2, 3 }; // list has type std::initializer\_list<int>

We will see later that such lists are typically used to specify the initial values of containers such as std::vector<>. As of C++17, this is the only minor quirk we need to be aware of. If we are using an older compiler, however, unexpected types may be deduced in place of auto in many more cases. Here’s an overview:

/\* C++11 and C++14 \*/

auto i {10}; // i has type std::initializer\_list<int> !!! auto pi = {3.14159}; // pi has type std::initializer\_list<double> auto list1{1, 2, 3}; // list1 has type std::initializer\_list<int> auto list2 = {4, 5, 6}; // list2 has type std::initializer\_list<int>

/\* C++17 and later \*/

auto i {10}; // i has type int

auto pi = {3.14159}; // pi has type std::initializer\_list<double> auto list1{1, 2, 3}; // error: does not compile!

auto list2 = {4, 5, 6}; // list2 has type std::initializer\_list<int>

To summarize, if our compiler properly supports C++17, you can use braced initialization to initialize any variable with a single value, provided you do not combine it with an assignment. If our compiler is not fully up- to-date yet, however, you should simply never use braced initializers with auto. Instead, either explicitly state the type or use assignment or functional notation.

## std::initializer\_list

An object of type std::initializer\_list<E> is constructed from an initializer list as if the implementation

generated and materialized (7.3.5) a prvalue of type “array of N const E”, where N is the number of elements

in the initializer list. Each element of that array is copy-initialized with the corresponding element of the

initializer list, and the std::initializer\_list<E> object is constructed to refer to that array.

TODO: GIVE some examples ..etc

## Summary

In this chapter, we covered the basics of computation in C++. You learned about most of the fundamental types of data that are provided for in the language. The essentials of what we’ve discussed up to now are as follows:

* Constants of any kind are called literals. All literals have a type.
* You can define integer literals as decimal, hexadecimal, octal, or binary values.
* A floating-point literal must contain a decimal point or an exponent or both. If there is neither, you have specified an integer.
* The fundamental types that store integers are short, int, long, and long long. These store signed integers, but you can also use the type modifier unsigned preceding any of these type names to produce a type that occupies the same number of bytes but stores unsigned integers.
  + The floating-point data types are float, double, and long double.
  + Uninitialized variables generally contain garbage values. Variables may be given initial values when they’re defined, and it’s good programming practice to do so. A braced initializer is the preferred way of specifying initial values.
  + A variable of type char can store a single character and occupies one byte. Type char may be signed or unsigned, depending on our compiler. You can also use variables of the types signed char and unsigned char to store integers. Types char, signed char, and unsigned char are different types.
  + Type wchar\_t stores a wide character and occupies either two or four bytes, depending on our compiler. Types char8\_t, char16\_t, and char32\_t may be better for handling Unicode characters in a cross-platform manner.
  + You can fix the value of a variable by using the const modifier. The compiler will check for any attempts within the program source file to modify a variable defined as const.
  + The four main mathematic operations correspond to the binary +, -, \*, and / operators. For integers, the modulus operator % gives you the remainder after integer division.
  + The ++ and -- operators are special shorthand for adding or subtracting one from a numeric variable. Both exist in postfix and prefix forms.
  + You can mix different types of variables and constants in an expression. The compiler will arrange for one operand in a binary operation to be automatically converted to the type of the other operand when they differ.
  + The compiler will automatically convert the type of the result of an expression on the right of an assignment to the type of the variable on the left when these are different. This can cause loss of information when the left-side type isn’t able to contain the same information as the right-side type—double converted to int, for example, or long converted to short.
  + You can explicitly convert a value of one type to another using the static\_cast<>()

operator.

* + The std::format() function of the <format> module offers a multitude of options to tune the formatting of textual output.

**EXERCISES**

The following exercises enable you to try what you’ve learned in this chapter. If you get stuck, look back over the chapter for help. If you’re still stuck after that, you can download the solutions from the Apress website ([www.apress.com/book/download.html](http://www.apress.com/book/download.html)), but that really should be a last resort.

Exercise 2-1. Create a program that converts inches to feet and inches. In case you’re unfamiliar with imperial units: 1 foot equals 12 inches. An input of 77 inches, for instance, should thus produce an output of 6 feet and 5 inches. Prompt the user to enter an integer value corresponding to the number of inches and then make the conversion and output the result.

Exercise 2-2. Write a program that will compute the area of a circle. The program should prompt for the radius of the circle to be entered from the keyboard. Calculate the area using the formula area = pi \* radius \* radius, and then display the result.

Exercise 2-3. For our birthday you’ve been given a long tape measure and an instrument that measures angles (the angle between the horizontal and a line to the top of a tree, for instance). If you know the distance, d, you are from a tree, and the height, h, of our eye when peering into our angle-measuring device, you can calculate the height of the tree with the formula h + d\*tan(angle). Create a program to read h in inches, d in feet and inches, and angle in degrees from the keyboard, and output the height of the tree in feet.

* **Note** There is no need to chop down any trees to verify the accuracy of our program. Just check the solutions on the Apress website!

Exercise 2-4. Our body mass index (BMI) is our weight, w, in kilograms divided by the square of our height, h, in meters (w/(h\*h)). Write a program to calculate the BMI from a weight entered in pounds and a height entered in feet and inches. A kilogram is 2.2 pounds, and a foot is 0.3048 meters.

Exercise 2-5. Knowing our BMI with a precision higher than one decimal digit after the decimal point is, well, pointless. Adjust the program of Exercise 2-4 accordingly.

Exercise 2-6. Reproduce Table [2-6](file:///D:\1zero\My_CPP_BOOK\Beginning%20C++20%20-%20From%20Novice%20to%20Professional\Beginning%20C++20%20-%20From%20Novice%20to%20Professional-53-96%20(1).docx#_bookmark7) with a program, without hard-coding the numeric values or filling spaces, of course. If our command-line interface does not support Unicode characters (perfectly possible), you can replace π with “pi” and omit φ (the Greek letter “phi,” in case you were wondering).

Exercise 2-7. Add a row to our table of Exercise 2-6 for sin(π/4), showing the result with exponent notation and five digits after the decimal point. Make sure the exponent component begins with a capital E, not a lowercase e.

Exercise 2-8. Here’s an extra exercise for puzzle fans. Write a program that will prompt the user to enter two different positive integers. Identify in the output the value of the larger integer and the value of the smaller integer. Using the decision-making facilities of Chapter 5, this would be like stealing a piece of cake from a baby while walking in the park. What makes this a tough brain teaser, though, is that this can be done solely with the operators you’ve learned about in this chapter!