4 Using Variables and Assignment

Programs manipulate data values. Whether a program performs a single calculation—such as, say, converting a temperature value from Fahrenheit to Celsius—reads data only to display it, or performs much more complex calculations and interactions, the values a program manipulates must be both accessible and assignable. Accessible means that a value must reside somewhere in computer memory and should be retrievable. Assignable means a value or the result of a calculation must be stored somewhere in computer memory to be later retrieved. Each value has a data type and a named location where it is stored. These can either be variables or constants.

Variables, or non-constant variables, hold values that are the result of calculations or data that will change as the program executes. **Constant variables** are variables that don't change their value once they are given a value. Variables, whether constant or variable, receive their values via **assignment**. Assignment is a simple expression. **Literal values**, or **literals**, are values encoded in the program and can never change.

The following topics will be covered in this chapter:

- Using types to determine the interpretation of values
- Learning some of the pitfalls of the `#define` directive and why constants are preferred
- Writing a program to set various constants in different ways
- Writing a program to use variables and constants (not just set them)
- Understanding four types of assignment

Technical requirements

Continue to use your computer with the following:

- A plaintext editor of your choice
- A console, terminal, or command-line window (depending on your OS)
- A compiler—either GCC or `clang`—for your particular OS

The source code for this chapter can be found at `https://github.com/PacktPublishing/Learn-C-Programming`.

Understanding types and values

Every value in a computer program has an associated type. The type of a value can be inferred by how it is expressed in the program code and how it is coded. Alternatively, the type of a value can be explicitly determined by you, the programmer. A value in C always has a type. So, a value can have either an inferred or *implicit* type or it can have an *explicit* type.

There are also inferred types from literal values. A literal value is a sequence of digits in the program code whose value is implicitly determined by the compiler at compile time, which is when the program is compiled. The value of a literal can never change; it is baked into the program code.

When a value is given an explicit type, the compiler assigns a type to that value. A value of one type can also be converted into another type, either implicitly by how it is used or explicitly with **typecasting**.

So, we should always think of the value/type pair. The type determines not only how the value is interpreted but also what possible valid ranges of values it can have.

If we have a value, then we should always ask *what is its type?* If we have a type, then we should always ask *what values can it have?* and *what value is it now?* This kind of thinking will be critical when we look at looping and arrays.

Introducing variables

A **variable** is a location in memory that holds a value of a specified type that can vary over the life of the variable, identified by its name. When the variable is defined with both a type and an identifier, its life begins. It can hold the same value throughout its life or it can be modified or overwritten with a new value of that type. The variable's life ends—that is, the memory it identifies is deallocated—when the block in which it was declared ends. We'll talk more about variable lifetimes in `Chapter 25`, *Understanding Scope*.

So, a variable is a memory location with an *identifier* (name) associated with a type that contains a value. The following three components are essential:

- A unique identifier or name
- A type
- A value

The variable should always have some known starting value, even if it is 0; this is called **initialization**. If we don't give the variable an initial value, we can never be sure what value it might have from one run to the next. When function blocks and programs are deallocated, the values occupied by their memory are left behind. It is, therefore, up to us to ensure that we initialize the memory we use for known good values.

A variable is initialized and overwritten by means of an assignment operation, where a value is assigned to the memory location identified by the variable. Once a constant variable is given a value, that value can never change.

Before we explore values and assignment, we need to understand explicit typing when we create and declare variables.

Naming variables

Every variable has an identifier or a name. A variable name is an identifier; function names are identifiers. We will encounter other kinds of identifiers that are used in many different contexts.

An **identifier**, or name, in C is a sequence of capital letters (`A..Z`) and small letters (`a..z`), digits (`0..9`), and the underscore (_) character. An identifier may not begin with a digit. Upper and lowercase letters are different from each other, so `achar`, `aChar`, `AChar`, and `ACHAR` would identify different variables. An identifier may not have the same spelling as a C keyword. A list of C keywords can be found in the *Appendix* section of this book.

As with function identifiers, relying on the casing of letters to differentiate variables is not good programming practice. The most essential guideline is that variable names should closely match the kinds of values they hold. Name variables so that their names clearly reflect their purpose—for example, `inch`, `foot`, `yard`, and `mile`.

There are many conventions for naming variables. Two common methods to make variable names descriptive yet easy to read are camel-case and underscore-separated, also known as **snake-case**. Camel-case names have the beginning characters of words within the name capitalized. In underscore-separated names, _ is used between words:

- **All-lowercase**: `inchesperminute`, `feetpersecond`, and `milesperhour`
- **Camel-case**: `inchesPerMinute`, `feetPerSecond`, and `milesPerHour`
- **Snake-case** (or **underscore-separated**): `inches_per_minute`, `feet_per_second`, and `miles_per_hour`

As you can see, all-lowercase names are somewhat difficult to read. However, these are not nearly as difficult to read as all-uppercase names. The other two ways are quite a bit easier to read. Therefore, we prefer to use either of the last two. We will use camel-case identifiers throughout this book.

If you choose one identifier naming convention, stick to it throughout your program. Do not mix different identifier naming schemes as this makes remembering the exact name of a thing, function identifiers, and other identifiers much more difficult and error-prone.

We can now explore explicit types of variables.

Introducing explicit types of variables

The format of a variable declaration is `type identifier;` or `type identifier1, identifiers, ... ;`.

Here, `type` is one of the data types that we encountered earlier and `identifier` is the name of the variable we are declaring. In the first example, a single variable is declared. In the second form, multiple variables are declared, each having the same type, separated by commas. Note that each one is a C statement because it concludes with `;`. Consider the following variable declarations:

```
#include <stdbool.h>   /* So we can use: bool, true, false */

int       aNumber;
long      aBigNumber;
long long aReallyBigNumber;
float     inches;
float     feed;
float     yards;
double    length, width, height;
bool      isItRaining;
```

In each of these declarations, we use spacing to make the type and name of each variable easier to read. Unfortunately, these declarations are not necessarily the best we could use. The values of the variables do not have *any* value yet. However, they are not yet initialized.

Using explicit typing with initialization

A better format for a variable declaration is one where the variable is *initialized* or given a starting value when it is declared, such as `type  identifier1 = value1;` ,`type identifier2 = value2;`, or

`type identifier1 = value1 , identifier2 = value2 , ... ;`.

Here, `value` is a literal constant or an already-declared variable. Note that in the second form of variable declaration, each variable must have its own initializing value. These are often accidentally omitted. Therefore, the first form is preferred. Consider the following declarations with initialization:

```
#include <stdbool.h>   /* So we can use: bool, true, false */

int       aNumber          = 10;
long      aBigNumber       = 3211145;
long long aReallyBigNumber = 425632238789;
```

```
float     inches          = 33.0;
float     feet            = 2.5;
float     yards           = 1780;
double    length = 1 , width = 2 , height = 10;
bool      isItRaining     = false;

int myCounter = 0;
int aDifferentCounter = myCounter;
```

As before, arbitrary spacing is used to vertically align variable names as well as to align their initial values. This is only done for readability. This sort of practice is not required but is generally considered good practice.

Initialization is a form of assignment; we are assigning a starting value to each variable. This is indicated by the = sign.

Note that because the type of the variable is explicit in its declaration, the assigned values—in this case, literal constants—are converted values of that type. So, while the following is correct, it may mislead an already-tired or overworked programmer reading it:

```
double length = 1 , width = 2 , height = 10;
```

Make note here of how a comma (`,`) is used to separate variable identifiers and the initialization. This declaration is still a single statement since it ends with a semi-colon (`;`). We will explore this further in `Chapter 5`, *Exploring Operators and Expressions*.

It is almost always better, although not required, to be explicit. A slightly better version of the preceding would be as follows:

```
double length = 1.0;
double width  = 2.0;
double height = 10.0;
```

Each type, identifier, and initial value is on a single line.

Having examined variables whose values can change, we can now turn our attention to constant variables (yes, that sounds odd to me, as well) and literal constants.

Exploring constants

We use variables to hold values that are computed or can change over their lifetime, such as counters. However, we often have values that we don't ever want to change during their lifetime. These are constants and can be defined in a number of ways for different uses.

Literal constants

Consider the following literal character sequences:

```
       65
      'A'
      8.0
 131072.0
```

Each of these has an internal byte stream of `0000 0000 0100 0001`. However, because of the punctuation surrounding these values, the compiler can infer what types they have from their context:

```
       65 --> int
      'A' --> unsigned char
      8.0 --> float
 131072.0 --> double
```

These values are literally typed into our source code and their types are determined by the way in which they are written, or precisely by how punctuation around them specifies the context for their data type.

The internal value for each is constant; it is that same bit pattern. The literal `65` value will always be interpreted as an integer with that value. The literal `'A'` value will always be interpreted as a single character. The literal `8.0` value *may* be interpreted as a float; if it is interpreted as a double, it will have a slightly different internal bit pattern. The literal `131072.0` value also may be interpreted as a float or a double. When it is interpreted as a double, it will also have the same bit pattern as the others.

Here are some examples to expand on each of these explanations:

- **Integer constants**: `65`, `1024`, `-17` , `163758` , `0` , and `-1`
- **Double constants**: `3.5`, `-0.7` , `1748.3753`, `0.0`, `-0.0000007`, `15e4`, and `-58.1e-4`
- **Character constants**: `'A'`, `'a'`, `'6'`, `'0'`, `'>'`, `'.'`, and `'\n'`

Notice that while the `0` value is present in each case, because they are all typed differently, they will have different internal bitstreams. Integer `0` and double `0.0` are patterns of all zeros. However, the `'0'` character will have a different value, which we will see in `Chapter 15`, *Working with Strings*.

You might also have noticed the strange notation for the `15e4` and `-58.1e-4` doubles. These values are expressed in scientific notation, where the first value evaluates to `15 x` 10^4, or `150,000`, and the second evaluates to `-58.1 x` 10^{-4}, or `-0.000581`. This is a shorthand notation for values that are either too large or too small to easily express with common decimal notation.

When a constant integer value is too large to fit into the valid ranges of the default type, the type is altered to hold the larger value. Therefore, an `int` value may become `long int` or `long long int`, depending on the value given and the machine architecture. This is implicit typecasting. The compiler does not want to lose any part of a value by stuffing it into a type with a smaller range, so it picks a type that will fit the value.

There is the same issue regarding implicit conversion of floats and doubles—when a literal value is interpreted, the compiler will pick the most appropriate data type depending on both the value and how that literal value is used.

The preceding literal constants are most common and are evaluated using the base-10 or decimal numbering system. In base-10, there are 10 symbols, `0` through `9`, to represent 10 values. Every value in a base-10 number system can be represented as a power of 10. The value `2,573` is really `2,000 + 500 + 70 + 3`, or, using exponents of base-10, it is $2*10^3 + 5*10^2 + 7*10^1 + 3*10^0$.

Sometimes, we may want to express a constant value in base-8, or *octal*, or base-16, or *hexadecimal*. In C, octal constants begin with the digit `0` and subsequent digits must be in the range of valid octal digits, `0` through `7`. Hexadecimal constants begin with `ox` or `0X` (`0` followed by the letter `x`) and the following characters may be the valid hexadecimal digits, `0` through `9` and `a` through `f` or `A` through `F`.

Without going into greater detail about octal and hexadecimal, here are some examples:

- **Octal integers**: `07`, `011`, and `036104`
- **Unsigned octal**: `07u`, `011u`, and `036104u`
- **Long octal**: `07L`, `011L`, and `036104L`
- **Hexadecimal integers**: `0x4`, `0Xaf`, `0x106a2`, and `ox7Ca6d`
- **Unsigned hexadecimal**: `0x4u`, `0Xafu`, `0x106a2u`, and `ox7Ca6du`
- **Long hexadecimal**: `0x4L`, `0XafL`, `0x106a2L`, and `ox7Ca6dL`

Just remember that base-10, base-8, and base-16 are just different ways to represent values.

Additionally, if you want to guarantee that a decimal number will be a smaller float type than a double, you can follow it with `f` or `F`. If you want it to be a much larger long-double, you can follow it with `l` or `L`. `L` is preferred over using `l` so as not to confuse the number `1` with the letter `l`. Here are some examples:

- **Float literals**: `0.5f`, `-12E5f`, and `3.45e-5F`
- **Double literals**: `0.5`, `-12E5`, and `3.45e-5`
- **Long-double literals**: `0.5L`, `-12E5fL`, and `3.45e-5FL`

Literal constants are interpreted by the compiler and embedded into our program.

We use these kinds of constants typically when we initialize our variables, as we saw in the previous section, or when we want to perform some known calculation on a variable, such as the following:

```
feet  = inches / 12.0;
yards = feet / 3.0;
```

In these calculations, both denominators are decimal numbers. Therefore, they determine the contexts of the resultant calculation. Regardless of what type `inches` is, the result will be a decimal number. Then, that result will be implicitly converted into whatever type `feet` is upon assignment. Likewise, the result of `feet / 3.0` will be a decimal number (float or double); upon assignment, that result will be converted into the type of `yards`.

Defined values

Another way to define constants is to use the `#define` preprocessor directive. This takes the form of `#define symbol text`, where `symbol` is an identifier and `text` is a literal constant or a previously defined symbol. Symbol names are typically all in uppercase and underscores are used to distinguish them from variable names.

An example would be to define the number of inches in feet or the number of feet in a yard:

```
#define INCHES_PER_FOOT 12
#define FEET_PER_YARD    3

feet  = inches / INCHES_PER_FOOT;
yards = feet / FEET_PER_YARD;
```

When the preprocessing phase of compilation encounters a definition such as this, it carries out a textural substitution. There is no type associated with the symbol and there is no way to verify that the actual use of a symbol matches its intended use. For this reason, the use of these kinds of constants is discouraged. We only included them here for completeness since many older C programs may make extensive use of this mechanism.

Because `#define` enables textural substitution, there are many other ways it can be and is used. This feature is so powerful that it must be used with extreme caution, if it is used at all. Many of the original reasons for relying on the preprocessor are no longer valid. The proper place for exploration of the preprocessor would be in a much more advanced programming course.

Explicitly typed constants

C provides a safer means of declaring named constants, other than by using the preprocessor. This is done by adding the `const` keyword to a variable declaration. This sort of declaration must be of the `const type identifier = value;` form, where `type`, `identifier`, and `value` are the same as in our preceding variable declaration form—except here, the initializing value is not optional. The constant variable loses its ability to change after the statement is evaluated. If we don't give it a value when we declare it, we cannot do so later. Such a declaration without an initializing value is, therefore, useless.

When we declare a constant in this manner, it is named; it has a type and it has a value that does not change. So, our previous example becomes as follows:

```
const float kInchesPerFoot = 12.0;
const float kFeetPerYard   =  3.0;

feet  = inches / kInchesPerFoot;
yards = feet / kFeetPerYard;
```

This is considered safer because the constant's type is known and any incorrect use of this type or invalid conversion from this type to some other type will be flagged by the compiler.

It is an arbitrary convention to begin constant names with the letter `k`; it is not mandatory to do so. We could also have named these constants `inchesPerFootConst` and `feetPerYardConst`, or, simply, `inchesPerFoot` and `feetPerYard`. Any attempt to change their values would result in a compiler error.

Naming constant variables

C makes no distinction between a variable identifier and a constant identifier. However, it is often useful to know whether the identifier you are using is a constant.

As with functions and variables, there are several conventions commonly used for naming constants. The following conventions are relatively common to arbitrarily differentiate constants from variables:

- Prefix a constant name with `k` or `k_`—for example, `kInchesPerFoot` or `k_inches_per_foot`.
- Suffix a name with `const` or `_const`—for example, `inchesPerFootConst` or `inches_per_foot_const`.
- Use snake-case with all the capitals—for example, `THIS_IS_A_CONSTANT`. All-uppercase is quite unreadable. This is typically used for the `#define` symbols to show that they are not just a constant—for example, `INCHES_PER_FOOT`.
- None. C does not distinguish between constants—for example, `int inchesPerFoot` versus `const int inchesPerFoot`. It should be obvious that the number of inches per foot does not ever change. Therefore, there is no real need to distinguish its name as a constant.

As with other naming conventions, any convention for constants, if one exists, should be clearly defined and consistently used throughout a program or set of program files. I tend to use either the first or the last conventions in my programs.

Using types and assignment

So, we have variables to hold values of a specified type that we can retrieve and manipulate by their identifiers. What can we do with them? Essentially, believe it or not, we can just copy them from one place to another. Values in variables or constants can only be changed through assignment. When we use them, their value is copied as a part of the evaluation but the value remains unchanged. A variable's value can be used in many ways over its lifetime, but that value will not change except when a new value is copied over it. We will now explore the various ways that variables are copied:

- Explicit assignment using the = operator
- Function parameter assignment

- Function return assignment
- Implicit assignment (this will be covered when we look at expressions in the next chapter)

Let's look at the first three ways of copying the variables in the subsequent sections.

Using explicit assignment, the simplest statement

We have already seen explicit assignment used when we initialized variables and constants. After we have declared a variable, we can change it by using the = assignment operator. An assignment statement is of the form `identifier = value;`, where `identifier` is our already-declared variable and the value can be a constant, another variable, the result of a calculation, or the returned value from a function. We will later see how all of these are expressions are evaluated and provide a result.

Here is an example of assignment statements:

```
feet = 24.75;
```

The `24.75` literal constant is evaluated as a value of float or double type and is assigned to the `feet` variable:

```
feet = yards/3.0 ;
```

The value of `yards` is obtained and then divided by the `3.0` literal constant. The result of the evaluation is assigned to the `feet` variable. The value of `yards` is unchanged:

```
feet = inchesToFeet( inches );
```

The `inchesToFeet()` function is called with the value obtained from the `inches` variable and is executed. Its result (its return value) is assigned to the `feet` variable. The value of `inches` is unchanged.

Assigning values by passing function parameters

When we declare a function prototype that takes parameters, we also declare those parameters as formal parameters. Formal parameters have no value, only a type and, optionally, a name. However, when we call the function, we supply the actual parameters, which are the values that are copied into those placeholder parameters.

Consider the `printDistance()` function declaration and definition in the following program:

```
#include <stdio.h>

void printDistance( double );

int main( void )
{
  double feet = 5280.0;
  printDistance( feet );
  printf( "feet = %12.3g\n" , feet );
  return 0;
}

  // Given feet, print the distance in feet and yards.
  //
void printDistance( double f )
{
  printf( "The distance in feet is %12.3g\n" , f );
  f = f / 3.0 ;
  printf( "The distance in yards is %12.3g\n" , f );
}
```

In this function, we focus on the assignment that occurs between the function call and the execution of the function (at its definition).

The function prototype says that the function takes a single parameter of the `double` type. The name is not given since it is optional. Even if we did give a name, it is not required to match the parameter name given in the function definition. In the function definition, the parameter is named `f`. For the function, the `f` variable is created as a double with the value given in the function call. It is as if we had assigned the value of the `feet` variable to the `f` function variable. In fact, this is exactly how we do that. We can manipulate `f` because it is assigned a copy of the value of `feet` at the time the function is called. What this example shows is that value of the `f` variable is divided by `3.0` and assigned back to the `f` variable, and then printed to the console. The `f` variable has a lifetime of the block of the function and the changed value in `f` does not change the value in the `feet` variable.

Because we want to use good coding practices that make our intentions clear, rather than obtuse, a better version of this function would be as follows:

```
  // Given feet, print the distance in feet and yards.
  //
void printDistance( double feet )
{
  double yards = feet / 3.0 ;
```

```
    printf( "The distance in feet is %12.3g\n" , feet );
    printf( "The distance in yards is %12.3g\n" , yards );
  }
```

This is clearer because of the following:

- We declare the actual parameter, `feet`, which tells the reader what exactly is being passed into the function.
- We are explicitly declaring the `yards` variable and assigning it a value that is the number of feet, or `feet`, divided by `3.0`.

The lifetime of the `feet` actual parameter and the `yards` declared variable begins with the start of the function block and ends with the end of the function block.

Assignment by the function return value

A **function** is a statement that can return the result of its execution to its caller. When the function has a data type that is not `void`, the function is replaced by its returned value, which can then be assigned to a variable of a compatible type.

The returned value may be explicitly assigned to a variable or it may be implicitly used within a statement and discarded at the completion of the statement.

Consider the `inchesToFeet()` function declaration and definition:

```
#include <stdio.h>

double inchesToFeet( double );

int main( void )
{
  double inches = 1024.0;
  double feet = 0.0;
  feet = inchesToFeet( inches );
  printf( "%12.3g inches is equal to %12.3g feet\n" , inches , feet );
  return 0;
}

  // Given inches, convert this to feet
  //
double inchesToFeet( double someInches)
{
  double someFeet = someInches / 12.0;
  return someFeet;
}
```

In this function, we focus on the assignment that occurs at the `return` statement in the function block and the caller.

In this function, `inches`—known to the function as the `i` variable—is converted via a simple calculation to feet and that value is assigned to the `f` variable in the function. The function returns the value of `f` to the caller. Essentially, the function call is replaced by the value that it returns. That value is then assigned (copied) to the `feet` variable in the `main()` function.

First, we have the following two lines:

```
double feet = 0.0;
feet = inchesToFeet( inches );
```

Those two lines could be replaced with the following single line:

```
double feet = inchesToFeet( inches );
```

This statement both declares `feet` as a double and initializes it via the return value from the function call.

Summary

Variables are the means by which we store values and their associated types. Variables are identified by a given name. Variable declarations allocate memory for the lifetime of the variable. This depends on where the variable is declared. Variables declared within a block, `{` and `}`, only exist while that block is executing. There are variables whose values can change while the program executes, constant variables whose values do not change once they are given a value, and literal values that never change.

Variables are declared with explicit types. However, the type of a value can be implicitly inferred by how it is used. Literal values are constants whose type is inferred both by how they appear and how they are used.

The only way to change the value of a variable is by assigning a value to it. Initialization is the means of giving a variable or constant a value when it is declared. Otherwise, the value of variables can only change through direct assignment, which is assignment as the result of a function return value. Functions receive values via their function parameters when the function is called; these values are copies of the values when called and are discarded when the function returns.

Now, you might be wondering *how do we manipulate variables, not just copy them?* Variables are changed through assigning the resulting value from an evaluation. C has a rich set of operators to perform evaluations; we will explore these in the next chapter.

5
Exploring Operators and Expressions

So far, we have seen how values are stored to and from variables. Simply storing/retrieving values to/from variables, while important, is only a small part of handling values. What is far more important is the ability to manipulate values in useful ways, which corresponds to the ways we manipulate real-world values, such as adding up our restaurant bill or calculating how much further we have to go to get to grandma's house and how much longer that might take.

The kinds of manipulations that are reasonable to perform on one or more values depend entirely on what kinds of values they are, that is, their data types. What makes sense for one data type may not make sense for another. In this chapter, we will explore the myriad ways that values can be manipulated.

The following topics will be covered in this chapter:

- Understanding expressions and operations
- Exploring operations on numbers and understanding the special considerations regarding numbers
- Understanding how to convert values from one type to another—type conversion and casting
- Exploring character operations
- Exploring various ways to compare values
- Writing a program to print out truth tables
- Examining a snippet that performs simple bitwise operations
- Exploring the conditional operator
- Examining the sequence operator
- Exploring compound assignment operators
- Examining multiple assignment statements

- Exploring increment and decrement operators
- Writing a program to illustrate grouping
- Understanding operator precedence and why to avoid relying upon it (there is an easier way)

Technical requirements

As detailed in the *Technical requirements* section of `Chapter 1`, *Running Hello, World!*, continue to use the tools you have chosen.

The source code for this chapter can be found at `https://github.com/PacktPublishing/Learn-C-Programming`.

Expressions and operations

What are expressions? Well, in simple terms, an expression is a way of computing a value. We've seen how to do this with functions and return values. We will now turn to C's basic set of arithmetic operators for addition, subtraction, multiplication, and division, which are common in most programming languages. C adds to this a large number of operations that includes incrementation/decrementation, relational operators, logical operators, and bitwise manipulation. C further extends assignment operations in useful ways. Finally, C includes some unusual operators that are not commonly found in other languages, such as conditional and sequence operators.

The expressions we will explore in this chapter consist of one or more values as variables or constants combined with the help of operators. Expressions can be complete statements; however, just as often, expressions are components of complex statements. Operators work on one or more expressions, where an expression can be simple or complex:

- 5 is a literal expression that evaluates to the value of 5.
- 5 + 8 is an arithmetic expression of two simple expressions (literal constants), which, with the addition operator, evaluates to 13.
- A more complex expression, 5 + 8 - 10, is really two binary arithmetical operations where 5 and 8 are first evaluated to produce an intermediate result, then 10 is subtracted from it.

- 5; is an expression statement that evaluates to 5 and then moves on to the next statement. A more useful version of this would be `aValue = 5;`, which is really two expressions: the evaluation of `5` and then the assignment of that value to the `aValue` variable.

Each value of an expression can be one of the following:

- A literal constant
- A variable or constant variable
- The returned value from a function call

An example expression using all of these would be as follows:

```
5 + aValue + feetToInches( 3.5 )
```

Consider the following statement:

```
aLength = 5 + aValue + feetToInches( 3.5 );
```

The preceding statement is, in reality, five distinct operations in one statement:

- The retrieval of the value from the `aValue` variable
- The function call to `feetToInches()`
- The addition of the literal value `5` with the value of `aValue` giving an intermediate result
- The addition of the function call result to the intermediate result
- The assignment of the intermediate result to the `aLength` variable

An alternative way in which to calculate the same result can involve three simple statements instead of one complex statement, as follows:

```
aLength = 5;
aLength = aLength + aValue;
aLength = aLength + feetToInches( 3.5 );
```

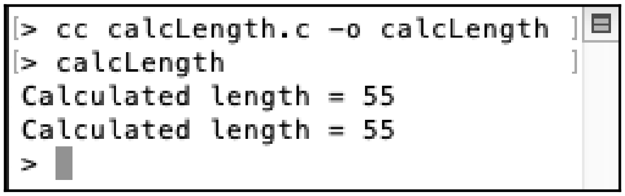
In this way, the different values are evaluated and added to the `aLength` variable. Instead of one assignment, there are three. Instead of a temporary intermediate result, the results of the additions are accumulated explicitly in the `aLength` variable as a result of each statement.

A simple program, `calcLength.c`, that applies each method of using simple and complex expressions, is as follows:

```
#include <stdio.h>

int feetToInches( double feet )
{
  int inches = feet * 12;
  return inches;
}

int main( void )
{
  int aValue   = 8;
  int aLength  = 0;

  aLength = 5 + aValue + feetToInches( 3.5 );
  printf( "Calculated length = %d\n" , aLength );

  aLength = 5;
  aLength = aLength + aValue;
  aLength = aLength + feetToInches( 3.5 );
  printf( "Calculated length = %d\n" , aLength );
}
```

This program calculates `aLength` from the sum of a literal value, which, in this case, is `5`; a variable, `aValue`; and the result of the `feetToInches()` function. It then prints out the result to the Terminal. The program itself is not very useful—we have no idea what we are calculating nor do we know why the values that were chosen are significant. For now, however, let's just focus on the expression of `aLength`. `aLength` is a value calculated by adding three other values together in one complex statement and again with three simple statements.

Now, create the `calcLength.c` file, type in the program, and then save the file. Compile the program and run it. You should see the following output:



```
> cc calcLength.c -o calcLength
> calcLength
Calculated length = 55
Calculated length = 55
>
```

As you can see, the single statement for calculating `aLength` is far less verbose than using the three statements to do so. However, neither approach is incorrect nor is one method always preferred over the other. When calculations are relatively simple, the first method might be clearer and more appropriate. On the other hand, when calculations become much more complex, the second method might make each step of the computation clearer and more appropriate. Choosing which method to employ can be a challenge as you are trying to find a balance between brevity and clarity. Whenever you have to choose one over the other, always choose clarity.

Introducing operations on numbers

The basic arithmetic operators on numbers are addition (+), subtraction (-), multiplication (*), and division (/). They are binary operations as they work on one pair of expressions at a time. They work largely as you would expect them to for both integers (whole numbers) and real numbers. Division for two real numbers results in a real number. Division for two whole numbers also results in a whole number; any possible fraction part is discarded. There is also the modulo operator (%) that will provide the integer remainder of the division of two integers.

For example, *12.0 / 5.0* (two real numbers) evaluates to *2.5*, whereas *12 / 5* (two integers) evaluates to *2*. If we were working only with integers and we needed the remainder of *12 / 5*, we would use the remainder operator, %. Thus, *12 % 5* evaluates to another integer, *2*.

Many languages have an exponent operator. C does not. To raise an expression to a power, standard C provides the library function, `pow( x , y)`. The prototype for this function is `double pow( double x , double y );`, which raises the value of `x` to the power of value `y` and yields `double` as its result. To use this function in your program, include the `<math.h>` header file wherever the prototype is declared.

Let's create a new file, `convertTemperature.c`, where we will create two useful functions, `celsiusToFahrenheit()` and `fahrenheitToCelsius()`, as follows:

```
  // Given a Celsius temperature, convert it to Fahrenheit.
double celsiusToFahrenheit( double degreesC )
{
  double degreesF = (degreesC * 9.0 / 5.0 ) + 32.0;
  return degreesF;
}

  // Given a Fahrenheit temperature, convert it to Celsius.
double fahrenheitToCelsius( double degreesF )
{
  double degreesC = (degreesF - 32 ) * 5.0 / 9.0 ;
  return degreesC;
}
```

Each function takes a `double` value type as an input parameter and returns the converted value as a `double`.

There are a couple of things to take note of regarding these functions.

First, we could have made them single-line functions by combining the two statements in each function body into one, as follows:

```
    return (degreesC * 9.0  / 5.0 ) + 32;
```

Here is another example:

```
    return (degreesF - 32 ) * 5.0 / 9.0;
```

Many programmers would do this. However, as your programming skills advance, this actually becomes a needless practice—it doesn't really save much of anything, and it makes debugging with a debugger (an advanced topic) far more difficult and time-consuming.

Many programmers are further tempted to turn these functions into `#define` macro symbols (another advanced topic), as follows:

```
  #define celsiusToFahrenheit( x )  ((x * 9.0  / 5.0 ) + 32)
  #define fahrenheitToCelsius( x )  ((x - 32 ) * 5.0 / 9.0)
```

Using macros can be dangerous because we could lose type information or the operations in such a macro might not be appropriate for the type of value given. Furthermore, we would have to be extremely careful about how we craft such preprocessor symbols to avoid unexpected results. For the few characters of typing saved, neither the single-line complex return statement nor the macro definitions are worth the potential hassle.

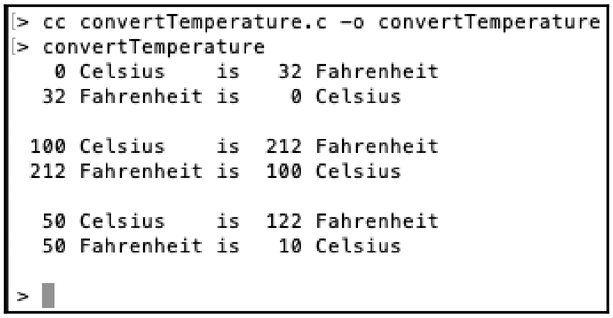
There are many temptations of using the preprocessor as much as possible—that is, to overuse the preprocessor. There lies the road to perdition! Instead, if you find yourself being pulled by such temptations for whatever reason, take a look at section *Using preprocessor effectively*, from `Chapter 25`, *Understanding Scope*.

Second, we use the grouping operator, `(` and `)`, to ensure our calculations are performed in the correct order. For now, just know that anything inside `(` and `)` is evaluated first. We will discuss this in more detail later on in this chapter.

We can now finish the program that uses the two functions we created. Add the following to `convertTemperature.c` before the two function definitions:

```
#include <stdio.h>

double celsiusToFahrenheit( double degreesC );
double fahrenheitToCelsius( double degreesF );

int main( void ) {
  int c = 0;
  int f = 32;
  printf( "%4d Celsius    is %4d Fahrenheit\n" ,
           c , (int)celsiusToFahrenheit( c ) );
  printf( "%4d Fahrenheit is %4d Celsius\n\n"  ,
           f , (int)fahrenheitToCelsius( f ) );
  c = 100;
  f = 212;
  printf( "%4d Celsius    is %4d Fahrenheit\n" ,
           c , (int)celsiusToFahrenheit( c ) );
  printf( "%4d Fahrenheit is %4d Celsius\n\n"  ,
           f , (int)fahrenheitToCelsius( f ) );
  c = f = 50;
  printf( "%4d Celsius    is %4d Fahrenheit\n" ,
          c , (int)celsiusToFahrenheit( c ) );
  printf( "%4d Fahrenheit is %4d Celsius\n\n"  ,
          f , (int)fahrenheitToCelsius( f ) );
  return 0
}

// function definitions here...
```

With all of the parts in place, save the file, compile it, and then run it. You should see the following output:



```
> cc convertTemperature.c -o convertTemperature
> convertTemperature
   0 Celsius    is   32 Fahrenheit
  32 Fahrenheit is    0 Celsius

 100 Celsius    is  212 Fahrenheit
 212 Fahrenheit is  100 Celsius

  50 Celsius    is  122 Fahrenheit
  50 Fahrenheit is   10 Celsius

> 
```

Notice how we exercised our functions with known values to verify that they are correct. First, freezing values for each scale were converted to the other scale. Then, boiling values for each scale were converted to the other scale. We then tried a simple middle value to see the results.

You may be wondering how to perform the conversions if we pass values other than doubles into the function. You might even be inclined to create several functions whose only difference is the type of variables. Take a look at the `convertTemperature_NoNo.c` program. Try to compile it for yourself and see what kind of errors you get. You will find that, in C, we cannot overload function names; that is, use the same function name but with different parameter and return types. This is possible with other languages, but not with C.

In C, each function is simply called by its name; nothing else is used to differentiate one function call from another. A function having one name with a given type and two parameters cannot be distinguished from another function of the same name with a different type and no parameter.

We could try to embed the type names into the function names, such as `fahrenheithDblToCelsiusInt()` and `celsiusIntToCelsiusDbl()`, but this would be extremely tedious to declare and define for all data types. Additionally, it would be extremely difficult to use in our programs. Compiler errors, due to mistyping the function names and even mistyping the calling parameters, would be highly likely and time-consuming to work through in a large or complicated program. So, how does C deal with this?

Don't fret! We will consider this very topic in the next section, along with a complete program on how to use these functions.

Considering the special issues resulting from operations on numbers

When performing calculations with any numbers, the possible ranges of both the inputs and outputs *must* be considered. For each type of number, there is a limit to both its maximum values and minimum values. These are defined on each system in the C standard library for that system in the header file, `limits.h`.

As the programmer, you must ensure that the results of any arithmetic operation are within the limits of the range for the data type specified, or your program must check for valid inputs thereby preventing invalid outputs. There are three types of invalid outputs that will cause the C runtime to abort— **Not a Number** (**NaN**), underflow, and overflow.

Understanding NaN

A NaN result occurs when the result of an operation is an undefined or an unrepresentable number.

Consider this equation: $y = 1 / x$. What is the value of y as x approaches zero from the positive side? It will become an infinitely large positive value. What then is the value of y as x approaches zero from the negative side? It will become an infinitely large negative value. Mathematically, this is called a **discontinuity**, which cannot be resolved. As we approach zero from either direction, the result is a value that will be infinitely different when we approach from one direction or the other (an infinitely large positive value or an infinitely small negative value). Therefore, division by zero is mathematically undefined. In the computer, the result is NaN.

NaNs also occur when the data types are real, but the result of the computation is a complex number, for example, the square root of a negative number or the logarithm of a negative number. NaNs can also occur where discontinuities appear in inverse trigonometric functions.

Understanding underflow NaN

Underflow occurs when the result of an arithmetic operation is smaller than the smallest value that can be represented by the type specified.

For integers, this would mean either a number less than 0 if the integer is `unsigned` or a very large negative number if the integer is `signed` (for instance, -2 is smaller than -1).

For real numbers, this would be a number very, very close to zero (that is, an extremely small fractional part), resulting from the division of an extremely small number by a very large number, or the multiplication of two extremely small numbers.

Understanding overflow NaN

Overflow occurs when the result of an arithmetic operation is greater than the greatest value that can be represented for the type specified.

This would occur with both the addition and multiplication of two extremely large numbers or the division of a very large number by an extremely small number.

Considering precision

When performing calculations with real numbers, we need to be concerned with the exponential difference between two of them. When one exponent is very large (positive) and the other very small (negative), we will likely produce either insignificant results or a NaN. This happens when the calculated result will either represent an insignificant change to the largest exponent value via addition and subtraction—therefore, precision will be lost—or be outside the possible range of values via multiplication and division—therefore, a NaN will result. Adding a very, very small value to a very, very large value may not give any significant change in the resulting value—again, precision in the result will be lost.

It is only when the exponents are relatively close, and the calculated result is within a reasonable range, that we can be sure of the accuracy of our result.

Granted that with 64-bit integer values and up to 128-bit real values, the ranges of values are vast, even beyond ordinary human conception. More often, however, our programs will use data types that do not provide the extreme limits of possible values. In those cases, the results of operations should always be given some consideration.

Exploring type conversion

C provides mechanisms that allow you to convert one type of value into another type of the same value. When there is no loss of precision, in other words, when the conversion of values results in the same value, C operates without complaining. However, when there is a possible loss of precision, or if the resulting value is not identical to the original value, then the C compiler does not provide any such warning.

Understanding implicit type conversion and values

So, what happens when expressions are performed with operands of different types, for example, the multiplication of an `int` with a `float`, or the subtraction of a `double` from a `short`?

To answer that, let's revisit our `sizes_ranges2.c` program from `Chapter 3`, *Working with Basic Data Types*. There, we saw how different data types took different numbers of bytes; some are 1 byte, some are 2 bytes, some are 4 bytes, and most values are 8 bytes.

When C encounters an expression of mixed types, it first performs an implicit conversion of the smallest data type (in bytes) to match the number of bytes in the largest data type size (in bytes). The conversion occurs such that the value with the narrow range would be converted into the other with a wider range of values.

Consider the following calculation:

```
int    feet  = 11;
double yards = 0.0;
yards = feet / 3;
```

In this calculation, both the `feet` variable and the `3` literal are integer values. The resulting value of the expression is an integer. However, the integer result is implicitly converted into a double upon assignment. The value of `yards` is `3.0`, which is clearly incorrect. This error can be corrected by either type casting with `feet` or by using a decimal literal, as follows:

```
yards = (double)feet / 3;

yards = feet / 3.0;
```

The first statement casts `feet` to a `double` and then performs the division; the result is a `double`. The second statement specifies a decimal literal, which is interpreted as a `double` and performs the division; the result is a `double`. In both statements, because the result is a `double`, there is no conversion needed upon assignment to `yards`; `yards` now has the correct value of `3.66667`.

Implicit conversion also occurs when the type of the actual parameter value is different from the defined parameter type.

A simple conversion is when a smaller type is converted into a larger type. This would include short integers being converted into long integers or floats being converted into doubles.

Consider the following function declaration and the statement that calls it:

```
long int add( long int i1 , long int i2 )  {
  return i1 + i2;
}

int main( void )  {
  signed char b1 = 254;
  signed char b2 = 253;
  long int r1;
  r1 = add( b1 , b2 );
  printf( "%d + %d = %ld\n" , b1 , b2 , r1 );
}
```

The `add()` function has two parameters, which are both long integers of 8 bytes each. Later, `add()` is called with two variables that are 1 byte each. The single-byte values of `254` and `253` are implicitly converted into the wider `long int` when they are copied into the function parameters. The result of the addition is `507`, which is correct.

Most integers can easily be converted into floats or doubles. In the multiplication of an `int` (4 bytes) with a `float` (4 bytes), an implicit conversion will happen—`int` will be converted into a float. The implicit result of the expression would be a `float`.

In the subtraction of a double (8 bytes) from a short (2 bytes), two conversions happen on the short— first, it is converted into a long (8 bytes), and then it is converted into a double (8 bytes). The implicit result of the expression would be a double. Depending on what happens next in the compound expression, the implicit result may be further converted. If the next operation involves an explicit type, then the implicit result will be converted into that type, if necessary. Otherwise, it may again be converted into the widest possible implicit type for the next operation.

However, when we assign an implicit or explicit result type to a narrower type in the assignment, a loss of precision is likely. For integers, loss involves the high-order bits (or the bits with the largest binary values). A value of `32,000,000` assigned to a char will always be `255`. For real numbers, truncation and rounding occur. Conversion from a double to a float will cause rounding or truncation, depending upon the compiler implementation. Conversion from a `float` to an `int` will cause the fractional part of the value to be lost.

Consider the following statements:

```
long int add( long int i1 , long int i2 )  {
  return i1 + i2;
}
```

```
int main( void )  {
  signed char b1 = 254;
  signed char b2 = 253;
  signed char r1;
  r1 = add( b1 , b2 );
  printf( "%d + %d = %ld\n" , b1 , b2 , r1 );
}
```

The only change in these statements is the type of the `r1` variable; it is now a single byte. So, while `b1` and `b2` are widened to `long int`, `add()` returns a `long int`, but this 8-byte return value must be truncated into a single byte. The value assigned to `r1` is incorrect; it becomes `252`.

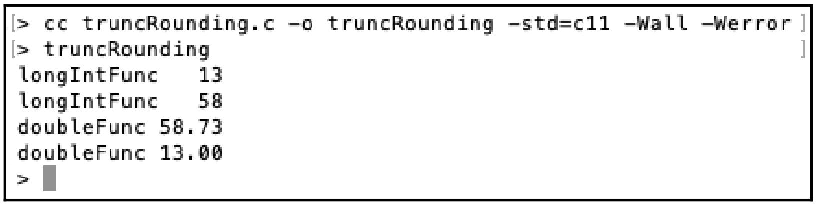
When performing complicated expressions that require a high degree of precision in the result, it is always best to perform the calculations in the widest possible data type, and only at the very end convert the result into a narrower data type.

Let's test this with a simple program. In `truncRounding.c`, we have two functions: one that takes a `double` as a parameter and prints it, and the other that takes a `long int` as a parameter and prints it. The following program illustrates implicit type conversion in which the parameter values are assigned to actual values:

```
#include <stdio.h>

void doubleFunc(  double   dbl );
void longintFunc( long int li );

int main( void ) {
  float floatValue   = 58.73;
  short int intValue = 13;
  longintFunc( intValue );
  longintFunc( floatValue ); // possible truncation

  doubleFunc( floatValue );
  doubleFunc( intValue );

  return 0;
}

void doublFunc( double dbl ) {
  printf( "doubleFunc %.2f\n" , dbl );
}

void longintFunc( long int li ) {
  printf( "longintFunc %ld\n" , li );
}
```

We have not yet explored the ways in which `printf()` can format values. For now, simply take for granted that `%.2f` will print a `double` value with 2 decimal places, and that `%ld` will print out a `long int`. This will be explained fully in `Chapter 19`, *Exploring Formatted Output*.

Enter, compile, and run `truncRounding.c`. You should see the following output:



```
> cc truncRounding.c -o truncRounding -std=c11 -Wall -Werror
> truncRounding
longIntFunc   13
longIntFunc   58
doubleFunc 58.73
doubleFunc 13.00
> 
```

Notice that no rounding occurs when `58.73` is converted into a `long int`. However, we do lose the fractional part; this is called **truncation**, where the fractional part of the value is cut off. A `short int` is properly converted into a `double` just as a `float` is properly converted into a `double`.

Also, notice that when you compiled and ran `truncRounding.c`, no compiler error nor runtime warning was given when the `float` was converted into a `long int` resulting in the loss of precision.

Using explicit type conversion – casting

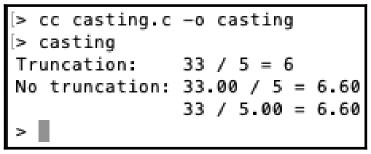
If we rely on implicit casting, our results may go awry or we may get unexpected results. To avoid this, we can cause an explicit, yet temporary, type change. We do this by casting. When we explicitly cast a variable to another type, its value is temporarily converted into the desired type and then used. The type of the variable and its value does not change.

Any expression can be prefixed by `(type)` to change its explicit type to the indicated type for the lifetime of the expression. This lifetime is typically a single statement. The explicit type is never changed, nor is the value stored in that explicitly typed variable. An example of this is given in the following program, `casting.c`:

```
#include <stdio.h>

int main( void ) {
  int numerator   =  33;
  int denominator =   5;
  double result   = 0.0;
  result = numerator / denominator;
```

```
    printf( "Truncation: %d / %d = %.2g\n" ,
             numerator , denominator , result );
    result = (double) numerator / denominator;
    printf( "No truncation: %.2f / %d = %.2f\n" ,
            (double)numerator , denominator , result );

    result = numerator / (double)denominator;
    printf( "               %d / %.2f = %.2f\n" ,
             numerator , (double)denominator , result );
    return 0;
  }
```

Enter, compile, and run `casting .c`. You should see the following output:



```
> cc casting.c -o casting
> casting
Truncation:    33 / 5 = 6
No truncation: 33.00 / 5 = 6.60
               33 / 5.00 = 6.60
>
```

In `casting.c`, we can see, in the first division expression, that there is no casting and no implicit conversion. Therefore, the result is an `int` and the fractional part is truncated. When the `int` result is assigned to a double, the fractional part has already been lost. In the second and third division statements, we guarantee that the operation is done on double values by casting either one of them to `double`. The other value is then implicitly converted to `double`. The result is a `double`, so when it is assigned to a `double`, there is no truncation.

The types of *numerators* and *denominators* are not changed permanently but only within the context of the expression where casting occurs.

Introducing operations on characters

Since characters are internally represented as integers, any of the integer operations can be applied to them too. However, only a couple of operations make sense to apply to characters—the additive operators (that is, addition and subtraction). While multiplying and dividing characters are legal, those operations never produce any practical results:

- `char - char` yields `int`.
- `char + int` yields `char`.
- `char - int` yields `char`.

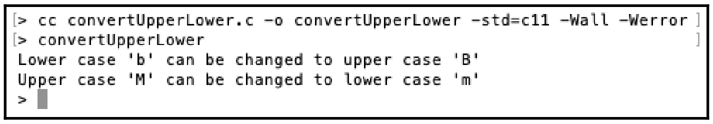
Remember that a `char` is only one unsigned byte, so any addition or subtraction outside of the range of `0..255` will yield unexpected results due to the truncation of high-order bits.

A common use of the addition and subtraction of characters is the conversion of a given ASCII character to uppercase or lowercase. If the character is uppercase, then simply adding 32 to it will give you its lowercase version. If the character is lowercase, then simply subtracting 32 from it will give you its uppercase version. An example of this is given in the following program, `convertUpperLower.c`:

```
#include <stdio.h>

int main( void ) {
  char lowerChar = 'b';
  char upperChar = 'M';

  char anUpper = lowerChar - 32;
  char aLower  = upperChar + 32;

  printf( "Lower case '%c' can be changed to upper case '%c'\n" ,
          lowerChar , anUpper );
  printf( "Upper case '%c' can be changed to lower case '%c'\n" ,
          upperChar , aLower );
}
```

Given a lowercase `'b'`, we convert it into uppercase by subtracting `32` from it. Given an uppercase `'M'`, we convert it into lowercase by adding `32` to it. We will explore characters much more thoroughly in `Chapter 15`, *Working with Strings*.

In your editor, create a new file and enter the `convertUpperLower.c` program. Compile and run it in a Terminal window. You should see the following output:



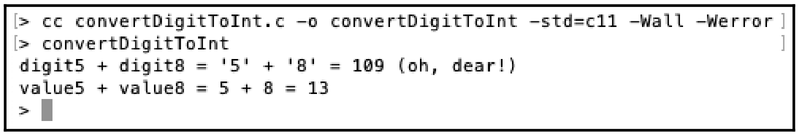
```
> cc convertUpperLower.c -o convertUpperLower -std=c11 -Wall -Werror
> convertUpperLower
Lower case 'b' can be changed to upper case 'B'
Upper case 'M' can be changed to lower case 'm'
>
```

Another common use of operations on characters is to convert the character of a digit (`'0'` to `'9'`) into its actual numerical value. The value of `'0'` is not `0` but some other value that represents that character. To convert a character digit into its numerical value, we simply subtract the character `'0'` from it. An example of this is given in the following program, `convertDigitToInt.c`:

```
#include <stdio.h>

int main( void ) {
 char digit5 = '5';
 char digit8 = '8';

 int sumDigits = digit5 + digit8;
 printf( "digit5 + digit8 = '5' + '8' = %d (oh, dear!)\n" ,
 sumDigits );

 char value5 = digit5 - '0'; // get the numerical value of '5'
 char value8 = digit8 - '0'; // get the numerical value of '8'
 sumDigits = value5 + value8;
 printf( "value5 + value8 = 5 + 8 = %d\n" ,
           sumDigits );
}
```

When we simply add characters together, unexpected results are likely to occur. What we really need to do is to convert each digit character into its corresponding numerical value and then add those values. The results of that addition are what we want.

In your editor, create a new file and enter the `convertDigitToInt.c` program. Compile and run it in a Terminal window. You should see the following output:



```
> cc convertDigitToInt.c -o convertDigitToInt -std=c11 -Wall -Werror
> convertDigitToInt
digit5 + digit8 = '5' + '8' = 109 (oh, dear!)
value5 + value8 = 5 + 8 = 13
>
```

In order to understand the difference between a character and its value, we will explore characters in much greater depth in `Chapter 15`, *Working with Strings*.

Exploring logical and relational operators

Early versions of C did not have explicit boolean (`true`, `false`) data types. To handle boolean values, C implicitly converts any zero value into the boolean `false` value and implicitly converts any nonzero value into the boolean `true` value. This implicit conversion comes in handy very often but must be used with care.

However, when we use `#include <stdbool.h>`, the official `bool` types and `true` and `false` values are available to us. We will explore later how we might choose to define our own boolean values with enumerations (`Chapters 9`, *Creating and Using Structures*) or with custom types (`Chapter 11`, *Working with Arrays*).

There are three boolean operators:

- `||`: The binary logical OR operator
- `&&`: The binary logical AND operator
- `!`: The unary logical NOT operator

These are logical operators whose results are always boolean `true` (nonzero) or `false` (exactly zero). They are so named in order to differentiate them from bitwise operators whose results involve a different bit pattern, which we shall learn about shortly.

The first two logical operators evaluate the results of two expressions:

```
expressionA operator expressionB
```

When the operator is logical AND `&&`, if `expressionA` evaluates to `false`, then `expressionB` is not evaluated (it does not need to be). However, when `expressionA` is true, `expressionB` must be evaluated.

When the operator is logical OR `||`, if `expressionA` evaluates to `true`, then `expressionB` is not evaluated (it does not need to be). However, when `expressionA` is false, `expressionB` must be evaluated.

The unary logical NOT `!` operator is employed, therefore, as `!expressionC`.

It takes the result of `expressionC`, implicitly converting it into a boolean result and evaluating it with its opposite boolean result. Therefore, `!true` becomes `false`, and `!false` becomes `true`.

In the `logical.c` program, three tables are printed to show how the logical operators work. They are known as **truth tables**. The values are printed as either decimal `1` or `0`, but they are really boolean values. The first truth table is produced with the `printLogicalAND()` function, as follows:

```
void printLogicalAND( bool z, bool o )
{
  bool zero_zero = z && z ;
  bool zero_one  = z && o ;
  bool one_zero  = o && z ;
  bool one_one   = o && o ;

  printf( "AND | %1d | %1d\n"     , z , o );
  printf( " %1d | %1d | %1d \n"   , z , zero_zero , zero_one );
  printf( " %1d | %1d | %1d \n\n" , o , zero_one  , one_one );
}
```

The next truth table is produced with the `printLogicalOr()` function, as follows:

```
void printLogicalOR( bool z, bool o )
{
  bool zero_zero = z || z ;
  bool zero_one  = z || o ;
  bool one_zero  = o || z ;
  bool one_one   = o || o ;

  printf( "OR | %1d | %1d\n"      , z , o );
  printf( " %1d | %1d | %1d \n"   , z , zero_zero , zero_one );
  printf( " %1d | %1d | %1d \n\n" , o , zero_one  , one_one );
}
```

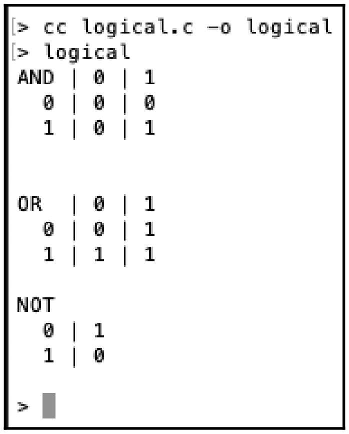
Finally, the `printLogicalNOT()` function prints the NOT truth table, as follows:

```
void printLogicalNOT( bool z, bool o )
{
  bool not_zero = !z ;
  bool not_one = !o ;
  printf( "NOT \n" );
  printf( " %1d | %1d \n"   , z , not_zero );
  printf( " %1d | %1d \n\n" , o , not_one );
}
```

Create the `logicals.c` file and enter the three truth table functions. Then, add the following program code to complete `logicals.c`:

```
#include <stdio.h>
#include <stdbool.h>

void printLogicalAND( bool z, bool o );
void printLogicalOR( bool z, bool o );
void printLogicalNOT( bool z, bool o );

int main( void )
{
  bool one = 1;
  bool zero = 0;

  printLogicalAND( zero , one );
  printLogicalOR( zero , one );
  printLogicalNOT( zero , one );
  return 0;
}
```

Save, compile, and run `logicals.c`. You should see the following output:



```
> cc logical.c -o logical
> logical
AND | 0 | 1
  0 | 0 | 0
  1 | 0 | 1


OR  | 0 | 1
  0 | 0 | 1
  1 | 1 | 1

NOT
  0 | 1
  1 | 0

> 
```

These are known as truth tables. When you perform the `AND`, `OR`, or `NOT` operations on a value in the top row and a value in the left column, the intersecting cell is the result. So, `1 AND 1` yields `1`, `1 OR 0` yields `1`, and `NOT 0` yields `1`.

Not all operations can be simply expressed as strictly boolean values. In these cases, there are the relational operators that produce results that are convenient to regard as `true` and `false`. Statements such as `if` and `while`, as we shall learn, test those results.

Relational operators involve the comparison of the result of one expression with the result of a second expression. They have the same form as the binary logical operators shown previously. Each of them gives a boolean result. They are as follows:

- > (**greater than operator**): `true` if `expressionA` is greater than `expressionB`.
- >= (**greater than or equal operator**): `true` if `expressionA` is greater than or equal to `expressionB`.
- < (**less than operator**): `true` if `expressionA` is less than `expressionB`.
- <= (**less than or equal operator**): `true` if `expressionA` is less than or equal to `expressionB`.
- == (**equal operator** (note that this is different from the = assignment operator)): `true` if `expressionA` is equal to `expressionB`.
- `!=` (**not equal operator**): `true` if `expressionA` is not equal to `expressionB`.

We will defer exploring these operators in depth until we get to the `if`, `for`, and `while` statements in upcoming chapters.

Bitwise operators

Bitwise operators manipulate bit patterns in useful ways. The bitwise AND `&`, OR `|`, and XOR `^` operators compare the two operands bit by bit. The bitwise shifting operators shift all of the bits of the first operand left or right. The bitwise complement changes each bit in a bit pattern to its opposite bit value.

Each bit in a bit field (8, 16, or 32 bits) could be used as if it was a switch, or flag, determining whether some feature or characteristic of the program was *off* (`0`) or *on* (`1`). The main drawback of using bit fields in this manner is that the meaning of the bit positions can only be known by reading the source code, assuming that the proper source code is both available and well commented!

Bitwise operations are less valuable today, not only because memory and CPU registers are cheap and plentiful, but because they are now expensive operations computationally. They do, occasionally, find a useful place in some programs, but not often.

The bitwise operators are as follows:

- `&`: Bitwise AND; for example, 1 if both bits are 1.
- `|`: Bitwise OR; for example, 1 if either bit is 1.
- `^`: Bitwise XOR; for example, 1 if either but not both are 1.
- <<: Bitwise shift left. Each bit is moved over to the left (the larger bit position). It is equivalent to `value * 2`. `0010` becomes `0100`.
- >>: Bitwise shift right. Each bit is moved over to the right (the smaller bit position). It is equivalent to `value / 2`. `0010` becomes `0001`.
- `~`: Bitwise complement. Change each bit to its other; for example, `1` to `0`, and `0` to `1`.

The following is an example of bitwise operators and flags:

```
 /* flag name */  /* bit pattern */
const unsigned char lowercase 1; /* 0000 0001 */
const unsigned char bold 2; /* 0000 0010 */
const unsigned char italic 4; /* 0000 0100 */
const unsigned char underline 8; /* 0000 1000 */

unsigned char flags = 0;

flags = flags | bold; /* switch on bold */
flags = flags & ~italic; /* switch off italic; */
if((flags & underline) == underline) ... /* test for underline bit 1/on? */
if( flags & underline ) ... /* test for underline */
```

Instead of using bitwise fields, custom data types called **enumerations** and more explicit data structures, such as hash tables, are often preferred.

The conditional operator

This is also known as the ternary operator. This operator has three expressions—`testExpression`, `ifTrueExpression`, and `ifFalseExpression`. It looks like this:

```
testExpression ? ifTrueExpression : ifFalseExpression
```

In this expression, `testExpression` is evaluated. If the `testExpression` result is `true`, or nonzero, then `ifTrueExpression` is evaluated and its result becomes the expression result. If the `testExpression` result is `false`, or exactly zero, then `ifFalseExpression` is evaluated and its result becomes the expression result. Either `ifTrueExpression` is evaluated or `ifFalseExpression` is evaluated—never both.

This operator is useful in odd places, such as setting switches, building string values, and printing out various messages. In the following example, we'll use it to add pluralization to a word if it makes sense in the text string:

```
          printf( "Length = %d meter%c\n" , len, len == 1 ? '' : 's' );
```

Or, we can use it to print out whole words:

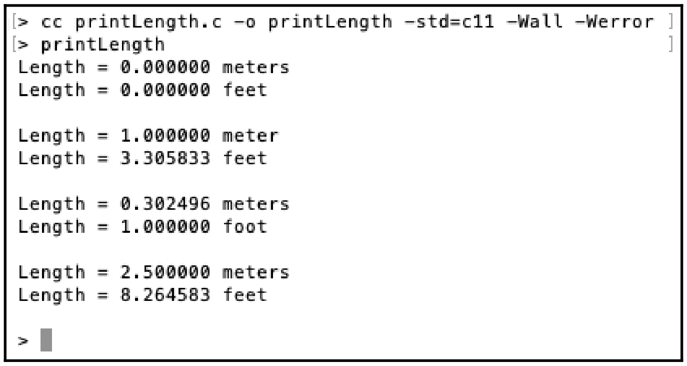
```
          printf( "Length = %d %s\n" , len, len == 1 ? "foot" : "feet" );
```

The following program uses these statements:

```
#include <stdio.h>

void printLength( double meters );

int main( void ) {
 printLength( 0.0 );
 printLength( 1.0 );
 printLength( 12.0 / 39.67 ); // very nearly 1 foot
 printLength( 2.5 );
}

void printLength( double meters ) {
 double feet = meters * 39.67 / 12.0;
 printf( "Length = %f meter%c\n" ,
         meters,
         meters == 1.0 ? ' ' : 's' );
 printf( "Length = %f %s\n" ,
         feet,
         0.99995 < feet && feet < 1.00005 ? "foot" : "feet" );
}
```

In the preceding program, you might be wondering why the statement for determining `"foot"` or `"feet"` has become so much more complex. The reason is that the `feet` variable is a computed value. Furthermore, because of the precision of the `double` data type, it is extremely unlikely than any computation will be *exactly* `1.0000`..., especially when division is involved. Therefore, we need to consider values of `feet` that are reasonably close to zero but might never be *exactly* zero. For our simple example, four significant digits will suffice.

When you type in the `printLength.c` program, save it, compile it, and run it, you should see the following output:



```
> cc printLength.c -o printLength -std=c11 -Wall -Werror
> printLength
Length = 0.000000 meters
Length = 0.000000 feet

Length = 1.000000 meter
Length = 3.305833 feet

Length = 0.302496 meters
Length = 1.000000 foot

Length = 2.500000 meters
Length = 8.264583 feet

>
```

Be careful, however, not to overuse the ternary operator for anything other than simple value replacements. In the next chapter, we'll explore how more explicit solutions are commonly used for general conditional executions.

The sequence operator

Sometimes, it makes sense to perform a sequence of expressions as though they were a single statement. This would rarely be used or make sense in a normal statement.

We can string multiple expressions together in sequence using the `,` operator. Each expression is evaluated from left to right in the order they appear. The value of the entire expression is the resultant value of the rightmost expression.

For instance, consider the following:

```
int x = 0, y = 0, z = 0;  // declare and initialize.
...
...
x = 3 , y = 4 , z = 5;
...
...
x = 4; y = 3; z = 5;
...
...
x = 5;
y = 12;
z = 13;
```

The single line assigning all three variables is perfectly valid. However, in this case, there is little value in doing it. The three variables are either loosely related or not related at all from what we can tell in this snippet.

The next line makes each assignment its own expression and condenses the code from three lines to one. While it is also valid, there is seldom a need for this.

This operator, however, does make sense in the context of iterative statements, such as `while() …`, `for() …`, and `do … while()`. They will be explored in `Chapter 7`, *Exploring Loops and Iteration*.

Compound assignment operators

As we have already seen, expressions can be combined in many ways to form compound expressions. There are some compound expressions that recur so often that C has a set of operators that make them shorter. In each case, the result is formed by taking the variable on the left of the operator, performing the operation on it with the value of the expression on the right, and assigning it back to the variable on the left.

Compound operations are of the form `variable operator= expression`.

The most common of these is incrementation with an assignment:

```
counter = counter + 1;
```

With the += compound operator, this just becomes the following:

```
counter += 1 ;
```

The full set of compound operators is as follows:

- `+=` assignment with addition to a variable
- `-=` assignment with subtraction to a variable
- `*=` assignment with multiplication to a variable
- `/=` assignment with division (integer or real) to a variable
- `%=` assignment with an integer remaindering to a variable
- `<<=` assignment with bitwise shift left
- `>>=` assignment with bitwise shift right
- `&=` assignment with bitwise AND
- `^=` assignment with bitwise XOR (exclusive OR)
- `|=` assignment with bitwise OR

These operators help to make your computations a bit more condensed and somewhat clearer.

Multiple assignments in a single expression

We have learned how to combine expressions to make compound expressions. We can also do this with assignments. For example, we could initialize variables as follows:

```
int height, width, length;
height = width = length = 0;
```

The expressions are evaluated from right to left, and the final result of the expression is the value of the last assignment. Each variable is assigned a value of `0`.

Another way to put multiple assignments in a single statement is to use the `,` sequence operator. We could write a simple swap statement in one line with three variables, as follows:

```
int first, second, temp;

 // Swap first & second variables.
temp = first, first = second, second = temp;
```

The sequence of three assignments is performed from left to right. This would be equivalent to the following three statements:

```
temp = first;
first = second;
second = temp;
```

Either way is correct. Some might argue that the three assignments are logically associated because of their commonality to being swapped, so the first way is preferred. Ultimately, which method you choose is a matter of taste and appropriateness in the given context. Always choose clarity over obfuscation.

Incremental operators

C provides even shorter shorthand (shortest shorthand?) operators that make the code even smaller and clearer. These are the autoincrement and autodecrement operators.

Writing the `counter = counter + 1;` statement is equivalent to a shorter version, `counter += 1;`, as we have already learned. However, this simple expression happens so often, especially when incrementing or decrementing a counter or index, that there is an even shorter shorthand way to do it. For this, there is the unary increment operator of `counter++;` or `++counter;`.

In each case, the result of the statement is that the value of the counter has been incremented by one.

Here are the unary operators:

- ++ autoincrement by 1, prefix or postfix
- -- autodecrement by 1, prefix or postfix

Postfix versus prefix incrementation

There are subtle differences between *how* that value of the counter is incremented when it is prefixed (++ comes before the expression is evaluated) or postfixed (++ comes after the expression).

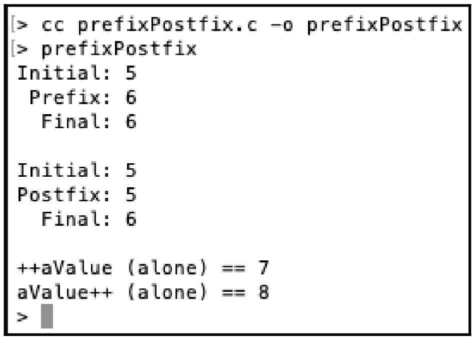
In prefix notation, ++ is applied to the result of the expression *before* its value is considered. In postfix notations, the result of the expression is applied to any other evaluation and then the ++ operation is performed.

Here, an example will be useful.

In this example, we set a value and then print that value using both the prefix and postfix notations. Finally, the program shows a more predictable method. That is, perform either method of incrementation as a single statement. The result will always be what we expect:

```
int main( void )
{
  int aValue = 5;

    // Demonstrate prefix incrementation.
  printf( "Initial: %d\n" , aValue );
  printf( " Prefix: %d\n" , ++aValue );  // Prefix incrementation.
  printf( "  Final: %\n"   , aValue );

  aValue = 5;   // Reset aValue.

    // Demonstrate postfix incrementation.
  printf( "Initial: %d\n" , aValue );
  printf( " Prefix: %d\n" , aValue++ );  // Postfix incrementation.
  printf( "  Final: %\n"  , aValue );

    // A more predictable result: increment in isolation.
  aValue = 5;
  ++aValue;
  printf( "++aValue (alone) == %d\n" , aValue );
  aValue = 5;
  aValue++;
  printf( "aValue++ (alone) == %d\n" , aValue );

  return 0;
}
```

Enter, compile, and run `prefixpostfix.c`. You should see the following output:



```
> cc prefixPostfix.c -o prefixPostfix
> prefixPostfix
Initial: 5
 Prefix: 6
  Final: 6

Initial: 5
Postfix: 5
  Final: 6

++aValue (alone) == 7
aValue++ (alone) == 8
>
```

In the output, you can see how the prefix and postfix notations affect (or not) the value passed to `printf()`. In prefix autoincrement, the value is first incremented and then passed to `printf()`. In postfix autoincrement, the value is passed to `printf()` as is, and after `printf()` is evaluated, the value is then incremented. Additionally, notice that when these are single, simple statements, both results are identical.

Some C programmers relish jamming together as many expressions and operators as possible. There is really no good reason to do this. I often go cross-eyed when looking at such code. In fact, because of the subtle differences in compilers and the possible confusion about if and when such expressions need to be modified, this practice is discouraged. Therefore, to avoid the possible side effects of the prefix or postfix incrementations, a better practice is to put the incrementation on a line by itself, when possible, and use grouping (we will discuss this in the next section).

Order of operations and grouping

When an expression contains two or more operators, it is essential to know which operation will be performed first, next, and so on. This is known as the **order of evaluation**. Not all operations are evaluated from left to right.

Consider *3 + 4 * 5*. Does this evaluate to *35*; *3 + 4 = 7 * 5 = 35*? Or does this evaluate to *23*; *4 * 5 = 20 + 3 = 23*?

If, on the other hand, we explicitly group these operations in the manner desired, we remove all doubt. Either *3 + **(**4 * 5**)*** or *(3 + 4) * 5* is what we actually intend.

C has built-in precedence and associativity of operations that determine how and in what order operations are performed. Precedence determines which operations have a higher priority and are, therefore, performed before those with a lower priority. Associativity refers to how operators of the same precedence are evaluated—from left to right or from right to left.

The following table shows all the operators we have already encountered along with some that we will encounter in later chapters (such as postfix `[]  .  ->` and unary `*  &`). The highest precedence is the postfix group at the top and the lowest precedence is the sequence operator at the bottom:

Group	**Operators**	**Associativity**
Postfix	`() [] . ->`	Left to right
Unary	`! ~ ++ -- + - * & (type) sizeof`	Right to left
Multiplicative	`* / %`	Left to right
Additive	`+ -`	Left to right
Shifting	`>> <<`	Left to right
Relational	`< <= > >=`	Left to right
Equality	`== !=`	Left to right
Bitwise AND	`&`	Left to right
Bitwise complement	`^`	Left to right
Bitwise OR	`|`	Left to right
Logical AND	`&&`	Left to right
Logical OR	`||`	Left to right
Conditional	`:?`	Right to left
Assignment	`= += -+ *= /= %= &= != <<= >>=`	Right to left
Sequence	`,`	Left to right

What is most interesting here is that (1) grouping happens first, and (2) assignment typically happens last in a statement. Well, this is not quite true. Sequencing happens after everything else. Typically, though, sequencing is not often used in a single statement. It can, however, be quite useful as a part of the `for` complex statement, as we shall learn in `Chapter 7`, *Exploring Loops and Iterations*.

While it is important to know precedence and associativity, I would encourage you to be very explicit in your expressions and use grouping to make your intentions clear and unambiguous. As we encounter additional operators in subsequent chapters, we will revisit this operator precedence table.

Summary

Expressions provide a way of computing a value. Expressions are often constructed from constants, variables, or function results combined together by operators.

We have explored C's rich set of operators. We have seen how arithmetic operators (such as addition, subtraction, multiplication, division, and remainder) can apply to different data types—integers, real numbers, and characters. We touched on character operations; we will learn much more about these in `Chapter 15`, *Working with Strings*. We have learned about implicit and explicit type conversions. We learned about C boolean values, created truth tables for logical operators, and learned how relational operations evaluate to boolean values. We have explored C's shorthand operators when used with assignments and explored C's shortest shorthand operators for autoincrement and autodecrement. Finally, we learned about C's operator precedence and how to avoid reliance on it with the grouping operator. Throughout the chapter, we have written programs to demonstrate how these operators work. Expressions and operators are the core building blocks of computational statements. Everything we have learned in this chapter will be an essential part of any programs we create going forward.

In the next two chapters, `Chapter 6`, *Exploring Conditional Program Flow*, and `Chapter 7`, *Exploring Loops and Iterations*, not only will we use these expressions for computations, but we will also learn how expressions are essential parts of other complex C statements (`if()… else…`, `for()…`, `while()…`, and `do…while()`). Our programs can then become more interesting and more useful.