

6

Exploring Conditional Program Flow

Not only do the values of variables change when a program runs, but the flow of execution can also change through a program. The order of statement execution can be altered depending upon the results of conditional expressions. In a conditional program flow, there is one mandatory branch and one optional branch. If the condition is met, the first branch, or path, will be taken; if not, the second path will be taken.

We can illustrate such a branching with the following simple example:

*Is today Saturday?
If so, do the laundry.
Else, go for a walk.*

In this conditional statement, we are instructed to do the laundry if today is Saturday. For the remaining days that are not Saturday, we are instructed to go for a walk.

This is a simple conditional statement; it has a single conditional expression, a single mandatory branch, and a single optional branch. We will explore conditional statements that evaluate multiple conditions. We will also explore conditional statements where multiple branches may be taken.

The following topics will be covered in this chapter:

- Understanding various conditional expressions
- Using `if() ... else...` to determine whether a value is even or odd
- Using a `switch() ...` statement to give a message based on a single letter
- Determining ranges of values using `if() ... else if() ... else...`

- Exploring nesting `if()... else...` statements
- Understanding the pitfalls of nesting `if()... else...` statements

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>.

Understanding conditional expressions

We have seen how execution progresses from one statement to the next with simple statements. We have also seen how program execution can be diverted or redirected via a function call, after which it returns to the same place. We are now going to see how the flow of execution can change and how statements can be executed or skipped with some of C's complex statements.

Execution flow will be determined by the result of evaluating conditional expressions, which we learned about in the previous chapter. Conditional expressions may be either simple or complex. Complex conditional statements should be clear and unambiguous. If they cannot be made clear and unambiguous, they should be reworked to be less complex. When reworking them results in more awkward code, the complex conditional expression should, however, be thoroughly commented. Careful consideration should be given to the valid inputs and expected results of the conditional expression.

Conditional expressions appear in very specific places within a complex statement. In every case, the conditional expression appears enclosed between `(` and `)` in the complex statement. The result will always be evaluated as `true` or `false`, regardless of the expression's complexity.

Some conditional expressions are given as follows:

```
( bResult == true )
( bResult )                               /* A compact alternative. */

( status != 0 )
( status )                               /* A compact alternative where
status is                                */
```

```
/* only ever false when it is 0
*/
( count < 3 )
( count > 0 && count <= maxCount ) /* Both must be true for overall
expression */
/* to be true.
*/
```

In each case, the result is one of two possible outcomes. In this manner, we can use the result to perform one pathway, or branch, or the other branch.

Introducing the `if()... else...` complex statement

`if()... else...` is a complex statement that can have two forms—a simple form where the `if()...` part is present, and a complete form where both the `if()...` and `else...` parts are present.

In the `if()... else...` complex statement, either the `true` path or the `false` path will be executed. The path that is not taken will be ignored. In the simplest `if()...` statement where there is no `false` path, the `true` path is executed only if the conditional expression is true.

The statement has two syntactical forms, as follows:

- The simplest form (no false path), as illustrated in the following code snippet:

```
if( expression )
    statement1

statement3          /* next statement to be executed */
```

- The complete form (both, the `true` path and `false` path), as illustrated in the following code snippet:

```
if( expression )
    statement1
else
    statement2

statement3          /* next statement to be executed */
```

In both the `if() ... (simple)` and `if()... else... (complete)` statements, `expression` is evaluated. If the result is `true`, `statement1` is executed. In the complete form, if the expression result is `false`, `statement2` is executed. In either case, the execution continues with the next statement, `statement3`.

Note that the statements do not indicate whether there is a semicolon or not. This is because `statement1` or `statement2` may either be simple statements (which are terminated with a `;`) or they may be compound statements, enclosed between `{` and `}`. In the latter case, each statement with the statement block will be terminated by `;`, but not the overall block itself.

A simple use of this statement would be in determining whether an integer value is even or odd. This is a good opportunity to put the `%` modulo operator into action. A function that does this using the simple form is illustrated in the following code block:

```
bool isEven( int num ) {  
    bool isEven = false; // Initialize with assumption that  
                          // it's not false.  
    if( (num % 2) == 0 )  
        isEven = true;  
    return isEven;  
}
```

In the preceding function, we first assume the value is not even, and then test it with a simple `if` statement and return the result. Notice that the `if() ...` branch is a single statement that is only executed if `num % 2` is 0. We had to use a relational operator in the condition because when `num` is even, `num % 2` evaluates to 0, which would then be converted to the Boolean `false`, which would not be the correct result. If the condition is false, the `if` branch is not executed, and we return the value of `isEven`.

This function could also be written in a slightly condensed form using multiple `return` statements instead of the `num` local variable, as follows:

```
bool isEven ( int num) {  
    if( num % 2 )  
        return false;  
    return true;  
}
```

In the preceding function, we use a very simple conditional that will be nonzero only if `num` is odd; therefore, we must return `false` for the function to be correct. This function has two exit points, one that will be executed if the test condition is `true`, and the second that will be executed when the `if` branch is not executed. When `num` is even, the second `return` statement will never be executed.

The most condensed version of this function would be as follows:

```
bool isEven ( int num)  {
    return !(num % 2 )
}
```

When `num` is even, `num % 2` gives 0 (false), and so we have to apply NOT to return the correct result.

A function that uses the complete form of this complex statement would be as follows:

```
bool isOdd ( int num)  {
    bool isOdd;
    if( num % 2 ) isOdd = true;
    else          isOdd = false;
    return isOdd;
}
```

In the preceding function, `isOdd` is not initialized. We must, therefore, ensure that whatever value is given as input to the `isOdd` function is assigned either `true` or `false`. As in the preceding examples, each branch of the complex statement is a single statement assigning a value to `isOdd`.

To explore various uses of this complex statement, let's begin exploring a function that calculates leap years. Understanding of leap years in Western civilization did not appear until the year 1752. In that year, it was then assumed the solar year was 365.25 days, or 356 + 1/4, hence the use of the modulo of 4. So, in our first approximation of calculating leap years, our function would look like this:

```
bool isLeapYear( int year )  {
    // Leap years not part of Gregorian calendar until after 1752.
    // Is year before 1751?
    // Yes: return false.
    // No: "fall through" to next condition.
    //
    if( year < 1751 ) return false;

    // Is year an multiple of 4? (remainder will be 0)
    // Yes: return true.
    // No: "fall through" and return false.
    //
    if( (year % 4) == 0 ) return true;

    return false;
}
```

In this function block, each time we know whether the given year is a leap year or not, we return from the function block with that result using return logic to stop the execution of statements within the function. Note that there are several ways in which we could have written the second conditional expression, some are which are shown in the following code block:

```
if( (year % 4) == 0 ) ...
if( (year % 4) < 1 ) ...
if( !(year % 4) ) ...
```

Of these, the first way is the most explicit, while the last is the most compact. All are correct.

The remainder of this program looks like this:

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

bool isLeapYear( int );

int main( void ) {
    int year;

    printf( "Determine if a year is a leap year or not.\n\n" );
    printf( "Enter year: " );
    scanf( "%d" , &year );

    // A simple version of printing the result.
    if( isLeapYear( year ) )
        printf( "%d year is a leap year\n" , year );
    else
        printf( "%d year is not a leap year\n" , year );

    // A more C-like version to print the result.
    printf( "%d year is%s a leap year\n" , year , isLeapYear( year ) ? " " : "
not " );

    return 0;
}
```

In the main() code block, we use both an if()... else... statement to print out the result and a more idiomatic C-like printf() statement, which uses the ternary (conditional) operator. Either method is fine. In the C-like version, %s represents a string (a sequence of characters enclosed in "..."), to be filled in the output string.

Create a new file named `cle` and add the code. Compile the program and run it. You should see the following output:

```
> cc leapYear1.c
> a.out
Determine if a year is a leap year or not.

Enter year: 2000
2000 year is a leap year
2000 year is a leap year
> a.out
Determine if a year is a leap year or not.

Enter year: 1900
1900 year is a leap year
1900 year is a leap year
> █
```

But is this program correct? No, it is not, because of leap centuries. The year 2000 is a leap year but 1900 is not. It turns out that the solar year is slightly less than 365.25 days; it is actually approximately 365.2425 days, or $(365 + 1/4 - 1/100 + 1/400)$. Notice that we'll have to account for $1/100$ —every 100 years—and $1/400$ —every 400 years. Our function will have to get a bit more complicated when we revisit it after the next section.

Using a `switch()`... complex statement

With the `if()... else...` statement, the conditional expression evaluates to only one of two values—`true` or `false`. But what if we had a single result that could have multiple values, with each value requiring a different bit of code execution?

For this, we have the `switch()...` statement. This statement evaluates an expression to a single result and selects a pathway where the result matches the known values that we care about.

The syntax of the `switch()...` statement is as follows:

```
switch( expression ) {
    case constant-1 :    statement-1
    case constant-2 :    statement-2
    ...
    case constant-n :    statement-n
    default :           statement-default
}
```

Here, the expression evaluates to a single result. The next part of the statement is called the **case-statement block**, which contains one or more `case` clauses and an optional `default :` clause. In the case-statement block, the result is compared to each of the constant values in each `case` clause. When they are equal, the code pathway for that `case` clause is evaluated. These are indicated by the `case <value> :` clauses. When none of the specified constants matches the result, the default pathway is executed, as indicated by the `default :` clause.

Even though the `default :` clause is optional, it is always a good idea to have one, even if only to print an error message. By doing so, you can ensure that your `switch()` statement is being used as intended and that you haven't forgotten to account for a changed or a new value being considered in the `switch` statement.

As before, each statement could be a simple statement, or it could be a compound statement.

We could rewrite our leap year calculation to use the `switch` statement as follows:

```
switch( year % 4 ) {  
    case 0 :  
        return true;  
    case 1 :  
    case 2 :  
    case 3 :  
    default :  
        return false;  
}
```

Notice that for the `case 1`, `case 2`, and `case 3`, there is no statement at all. The evaluation of the result of the expression continues to the next case. It falls through each case evaluation into the default case and returns `false`.

We can simplify this to the only value we really care about, which is 0, as follows:

```
switch( year % 4 ) {  
    case 0 :  
        return true;  
    default :  
        return false;  
}
```

In the context of the `isLeapYear()` function, the `return` statements exit the `switch()` statement as well as exiting the function block.

However, there are many times where we need to perform some actions for a given pathway and then perform no further case comparisons. The match was found and we are done with the `switch` statement. In such an instance, we don't want to fall through. We need a way to exit the pathway as well as exit the case-statement block. For that, there is the `break` statement.

The `break` statement causes the execution to jump out, and to the end of the statement block, where it is encountered.

To see this in action, we'll write a `calc()` function that takes two values and a single character operator. It will return the result of the operation on the two values, as follows:

```
double calc( double operand1 , double operand2 , char operator ) {
    double result = 0.0;

    printf( "%g %c %g = " , operand1 , operator , operand2 );
    switch( operator ) {
        case '+':
            result = operand1 + operand2;          break;
        case '-':
            result = operand1 - operand2;          break;
        case '*':
            result = operand1 * operand2;          break;
        case '/':
            if( operand2 == 0.0 ) {
                printf( "*** ERROR *** division by %g is undefined.\n" ,
                    operand2 );
                return result;
            } else {
                result = operand1 / operand2;
            }
            break;
        case '%':
            // Remaindering: assume operations on integers (cast first).
            result = (int) operand1 % (int) operand2;
            break;
        default:
            printf( "*** ERROR *** unknown operator; operator must be + - * / or
%%\n" );
            return result;
            break;
    }
    /* break brings us to here */
    printf( "%g\n" , result );
    return result;
}
```

In this function, `double` data types are used for each operand so that we can calculate whatever is given to us. Through implicit type conversion, `int` and `float` will be converted to the wider `double` type. We can cast the result to the desired type after we make the function call because it is at the function call where we will know the data types being given to the function.

For each of the five characters-as-operators that we care about, there is a case-clause where we perform the desired operation assigning a result. In the case of `/`, we do a further check to verify that division by zero is not being done.

Experiment—comment out this `if` statement and do the division without the check to see what happens.

In the case of `%`, we cast each of the operands to `int` since `%` is an integer-only operator.

Experiment—remove the casting and add some calls to `calc()` with various real numbers to see what happens.

Also, notice the benefit of having a `default:` clause that handles the case when our `calc` program is called with an invalid operator. This kind of built-in error checking becomes invaluable over the life of a program that may change many times at the hands of many different programmers.

To complete our `calc.c` program, the rest of the program is comprised as follows:

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

double calc( double operand1, double operand2 , char operator );

int main( void ) {
    calc( 1.0 , 2.0 , '+' );
    calc( 10.0 , 7.0 , '-' );
    calc( 4.0 , 2.3 , '*' );
    calc( 5.0 , 0.0 , '/' );
    calc( 5.0 , 2.0 , '%' );
    calc( 1.0 , 2.0 , '?' );

    return 0;
}
```

In this program, we call `calc()` with all of the valid operators as well as an invalid operator. In the fourth call to `calc()`, we also attempt a division by zero. Save this program, and compile it. You should see the following output:

```
> cc calc.c
> a.out
1 + 2 = 3
10 - 7 = 3
4 * 2.3 = 9.2
5 / 0 = *** ERROR *** division by 0 is undefined.
5 % 2 = 1
1 ? 2 = *** ERROR *** unknown operator; operator must be + - * / or %
> █
```

In each case, the correct result is given for both valid operations and for invalid operations. Don't forget to try the experiments mentioned.

The `switch()... statement` is ideal when a single variable is being tested for multiple values. Next, we'll see a more flexible version of the `switch()... statement`.

Introducing multiple `if()... statements`

The `switch` statement tests a single value. Could we have done this another way? Yes—with the `if()... else if()... else if()... else... construct`. We could have written the `calc()` function using multiple `if()... else... statements`, in this way:

```
double calc( double operand1 , double operand2 , char operator ) {
    double result = 0.0;

    printf( "%g %c %g = " , operand1 , operator , operand2 );
    if( operator == '+' )
        result = operand1 + operand2;
    else if( operator == '-' )
        result = operand1 - operand2;
    else if( operator == '*' )
        result = operand1 * operand2;
    else if( operator == '/' )
        if( operand2 == 0.0 ) {
            printf( "*** ERROR *** division by %g is undefined.\n" ,
                    operand2 );
            return result;
        } else {
            result = operand1 / operand2;
        }
}
```

```
else if( operator == '%' ) {
    // Remaindering: assume operations on integers (cast first).
    result = (int) operand1 % (int) operand2;
} else {
    printf( "*** ERROR *** unknown operator; operator must be + - * / or
%%\n" );
    return result;
}
printf( "%g\n" , result );
return result;
}
```

This solution, while perfectly fine, introduces a number of concepts that need to be clarified, as follows:

- The first `if()`... and each `else if()`... statement correlates directly to each `case :` clause of the `switch()`... statement. The final `else...` clause corresponds to the `default:` clause of the `switch` statement.
- In the first three `if()`... statements, the `true` path is a single simple statement.
- In the fourth `if()`... statement, even though there are multiple lines, the `if()`... `else...` statement is itself a single complex statement. Therefore, it is also a single statement.
This `if()`... statement is slightly different than the other statements; it uses a conditional expression for a different variable, and it occurs in the `else if()`... branch as a complex statement. This is called a *nested if()*... statement.
- Again, in the last two pathways, each branch has multiple simple statements. To make them into a single statement, we must make them a part of a compound statement by enclosing them in `{` and `}`.
Experiment—remove the `{` and `}` from the last two pathways to see what happens.
- You might argue that for this case, the `switch()`... statement is clearer and thus preferred, or you may not think so.

In this code sample, we are comparing a single value to a set of constants. There are instances where a switch statement is not only unusable but cannot be used. One such instance is when we want to simplify ranges of values into one or more single values. Consider the `describeTemp()` function that, when given a temperature, provides a relevant description of that temperature. This function is illustrated in the following code block:

```
void describeTemp( double degreesF ) {
    char* message;
    if( degreesF > 100.0 ) message = "hot! Stay in the shade.";
    else if( degreesF >= 80.0 ) message = "perfect weather for swimming.";
    else if( degreesF >= 60.0 ) message = "very comfortable.";
    else if( degreesF >= 40.0 ) message = "chilly.";
    else if( degreesF >= 20.0 ) message = "freezing, but good skiing
weather.";
    else message= "way too cold to do much of anything!" ;
    printf( "%g°F is %s\n" , degreesF , message );
}
```

In this function, a temperature is given as a `double` type, representing °F. Based on this value, a series of `if()... else...` statements select the appropriate range, and then print a message describing that temperature. The `switch()...` statement could not be used for this.

We could also have written `describeTemp()` using the `&&` logical operator, as follows:

```
void describeTemp( double degreesF ) {
    char* message;
    if( degreesF >= 100.0 )
        message = "hot! Stay in the shade.";
    if( degreesF < 100.0 && degreesF >= 80.0 )
        message = "perfect weather for swimming.";
    if( degreesF < 80.0 && degreesF >= 60.0 )
        message = "very comfortable.";
    if( degreesF < 60.0 && degreesF >= 40.0 )
        message = "chilly.";
    if( degreesF < 40.0 && degreesF >= 20.0 )
        message = "freezing, but good skiing weather.";
    if( degreesF < 20.0 )
        message= "way too cold to do much of anything!" ;

    printf( "%g°F is %s\n" , degreesF , message );
}
```

In this version of `describeTemp()`, each `if()`... statement checks for a range of values of `degreesF`. Most of the conditional expressions test an upper limit and a lower limit. Notice that there is no `else()`... clause for any of these. Also, notice that for any given value of `degreesF`, one—and only one—`if()`... statement will be satisfied. It is important that all ranges of possible values are covered by this kind of logic. This is called *fall-through* logic, where executions fall through each `if()`... statement to the very end. We will see further examples of fall-through logic in the next section.

The full program, which also exercises the various temperature ranges, is found in `temp.c`. When you compile and run this program, you see the following output:

```
[> cc temp.c
> a.out
100°F is hot! Stay in the shade.
85°F is perfect weather for swimming.
70°F is very comfortable.
55°F is chilly.
40°F is chilly.
25°F is freezing, but good skiing weather.
10°F is way too cold to do much of anything!
-5°F is way too cold to do much of anything!
> █
```

We can now return to our `leapYear.c` program and add the proper logic for leap centuries. Copy the `leapYear1.c` program to `leapYear2.c`, which we will modify. We keep some parts from the preceding function, but this time, our logic includes both leap centuries (every 400 years) and non-leap centuries (every 100 years), as follows:

```
// isLeapYear logic conforms to algorithm given in
// https://en.wikipedia.org/wiki/Leap_year.
//
bool isLeapYear( int year ) {
    bool isLeap = false;

    // Leap years not part of Gregorian calendar until after 1752.

    if( year < 1751 )                // Is is before leap years known.
        isLeap = false;
    else if( (year % 4) != 0 )        // Year is not a multiple of 4.
        isLeap = false;
    else if( (year % 400) == 0 )      // Year is a multiple of 400.
        isLeap = true;
    else if( (year % 100) == 0 )      // Year is multiple of 100.
        isLeap = false;
    else
        isLeap = true; // Year is a multiple of 4 (other conditions 400
```

```
        // years, 100 years) have already been considered.  
    return isLeap;  
}
```

This underlying logic we are trying to mimic lends itself naturally to a series of `if()...else...` statements. We now handle 100-year and 400-year leap years properly. Save `leapYear2.c`, compile it, and run it. You should see the following output:

```
> cc leapYear2.c  
> a.out  
Determine if a year is a leap year or not.  
  
Enter year: 2000  
2000 year is a leap year  
> a.out  
Determine if a year is a leap year or not.  
  
Enter year: 1900  
1900 year is not a leap year  
> █
```

We are again using a sequence of `if()... else if()... else...` statements to turn a year value into a simple Boolean result. However, in this function, instead of returning when the result is known, we assign that result to a local variable, `isLeap`, and only return it at the very end of the function block. This method is sometimes preferable to having multiple `return` statements in a function, especially when the function becomes long or contains particularly complicated logic.

Notice that it correctly determines that 2000 is a leap year and that 1900 is not.

Using nested `if()... else...` statements

Sometimes, we can make the logic of `if()... else...` statements clearer by nesting `if()... else...` statements within either one or both clauses of the `if()... else...` statements.

In our `isLeap()` example, someone new to the intricacies of Gregorian calendar development and the subtleties of century leap year calculation might have to pause and wonder a bit about our `if/else` fall-through logic. Actually, this case is pretty simple; much more tangled examples of such a logic can be found. Nonetheless, we can make our logic a bit clearer by nesting an `if ... else ...` statement within one of the `if()... else...` clauses.

Copy `leapYear2.c` to `leapYear3.c`, which we will now modify. Our `isLeap()` function now looks like this:

```
bool isLeapYear( int year ) {
    bool isLeap = false;

    // Leap years not part of Gregorian calendar until after 1752.
    //
    if( year < 1751 )                // Is is before leap years known?
        isLeap = false;
    else if( (year % 4 ) != 0 )      // Year is not a multiple of 4.
        isLeap = false;
    else {                          // Year is a multiple of 4.
        if( (year % 400 ) == 0 )
            isLeap = true;
        else if( (year % 100 ) == 0 )
            isLeap = false;
        else
            isLeap = true;
    }
    return isLeap;
}
```

Again, we use a local variable, `isLeap`. In the last `else` clause, we know at that point, that year is divisible by four. So, now, we nest another `if()... else...` series of statements to account for leap centuries. Again, you might argue that this is clearer than before, or you might not.

Notice, however, that we enclosed—*nested*—the last series of `if()... else...` statements in a statement block. This was done not only to make its purpose clearer to other programmers but also to avoid the *dangling else problem*.

The dangling else... problem

When multiple statements are written in `if()... else...` statements where a single statement is expected, surprising results may occur. This is sometimes called the **dangling else problem** and is illustrated in the following code snippet:

```
if( x == 0 ) if( y == 0 ) printf( "y equals 0\n" );
else printf( "what does not equal 0\n" );
```


To which `if()` ... does the `else...` belong—the first one or the second one? To correct this possible ambiguity, it is often best to always use compound statements in `if()` ... and `else...` clauses to make your intention unambiguous. Many compilers will give an error such as the following: `warning: add explicit braces to avoid dangling else [-Wdangling-else]`.

On the other hand, some do not. The best way to remove doubt is to use brackets to associate the `else...` clause with the proper `if()` ... clause. The following code snippet will remove both ambiguity and the compiler warning:

```
if( x == 0 ) {
    if( y == 0 ) printf( "y equals 0\n" );
}
else printf( "x does not equal 0\n" );
```

In the usage shown in the preceding code block, the `else...` clause is now clearly associated with the first `if()` ..., and we see `x does not equal 0` in our output.

In the following code block, the `else...` clause is now clearly associated with the second `if()` ... clause, and we'll see `y does not equal 0` in our output. The first `if()` ... statement has no `else...` clause, as can be seen here:

```
if( x == 0 ) {
    if( y == 0 ) printf( "y equals 0\n" );
    else printf( "y does not equal 0\n" );
}
```

Notice how the `if(y == 0)` statement is nested within the `if(x == 0)` statement.

In my own programming experience, whenever I have begun writing an `if()` ... `else...` statement, *rarely* has each clause been limited to a single statement. Most often, as I add more logic to the `if()` ... `else...` statement, I have to go back and turn them into compound statements with `{` and `}`. As a consequence, I now *always* begin writing each clause as compound statements to avoid having to go back and add them in. Before I begin adding the conditional expression and statements in either of the compound statements, my initial entry looks like this:

```
if( ) {
    // blah blah blahblah
} else {
    // gabba gabba gab gab
}
```

You may find that the `} else {` line is better when broken into three separate lines. The choice is a matter of personal style. This is illustrated in the following code snippet:

```
if( ) {  
    // blah blah blahblah  
}  
else {  
    // gabba gabba gab gab  
}
```

In the first code example, the `else...` closing block and the `if()` opening block are combined on one line. In the second code example, each is on a line by itself. The former is a bit more compact, while the latter is less so. Either form is acceptable, and their use will depend upon the length and complexity of the code in the enclosed blocks.

Stylistically, we could also begin each of the `if` blocks with an opening `{` on its own line, as follows:

```
if( )  
{  
    // blah blah blahblah  
} else {  
    // gabba gabba gab gab  
}
```

Alternatively, we could do this:

```
if( ) {  
    // blah, blah blah blah.  
}  
else  
{  
    // gabba gabba, gab gab.  
}
```

Some people prefer block *openings* on their own line, while others prefer block openings at the end of the conditional statement. Again, either way is correct. Which way is used depends on both personal preference and the necessity of following the conventions of the existing code base; consistency is paramount.

Summary

From this chapter, we learned that we can not only alter program flow with function calls but also execute or omit program statements through the use of conditional statements. The `switch()` ... statement operates on a single value, comparing it to the desired set of possible constant values and executing the pathway that matches the constant value. The `if()` ... `else...` statement has a much wider variety of forms and uses. `if()` ... `else...` statements can be chained into longer sequences to mimic the `switch()` ... statement and to provide a richer set of conditions than possible with `switch()` ... `if()` ... `else...` statements can also be nested in one or both clauses to either make the purpose of that branch clear or the condition for each pathway less complex.

In these conditional statements, execution remains straightforward, from top to bottom, where only specific parts of the statement are executed. In the next chapter, we'll see how to perform multiple iterations of the same code path, with various forms of looping and the somewhat stigmatized `goto` statement.

7

Exploring Loops and Iteration

Some operations need to be repeated one or more times before their result is completely evaluated. The code to be repeated could be copied the required number of times, but this would be cumbersome. Instead, for this, there are loops—`for ...`, `while ...`, and `do ... while` loops. Loops are for statements that must be evaluated multiple times. We will explore C loops. After considering loops, the much-maligned `goto` statement will be considered.

The following topics will be covered in this chapter:

- Understanding brute-force repetition and why it might be bad
- Exploring looping with the `while()` ... statement
- Exploring looping with the `for()` ... statement
- Exploring looping with the `do ... while()` statement
- Understanding when you would use each of these looping statements
- Understanding how loops could be interchanged, if necessary
- Exploring the good, the bad, and the ugly of using `goto`
- Exploring safe alternatives to `goto`—`continue` and `break`
- Understanding the appropriate use of loops that never end

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>.

Understanding repetition

Very often, we need to perform a series of statements repeatedly. We might want to perform a calculation on each member of a set of values, or we might want to perform a calculation using all of the members in a set of values. Given a collection of values, we also might want to iterate over the whole collection to find the desired value, to count all the values, to perform some kind of calculation on them, or to manipulate the set in some way—say, to sort it.

There are a number of ways to do this. The simplest, yet most restrictive way is the **brute-force method**. This can be done regardless of the language being used. A more dynamic and flexible method is to iterate or repeatedly loop. C provides three interrelated looping statements—`while()`..., `for()`..., and `do...while()`. Each of them has a control or continuation expression, and a loop body. The most general form of these is the `while()`... loop. Lastly, there is the archaic `goto label` method of looping. Unlike other languages, there is no `repeat ... until()` statement; such a statement can easily be constructed from any of the others.

Each looping statement consists of the following two basic parts:

- The loop continuation expression
- The body of the loop

When the loop continuation expression evaluates to `true`, the body of the loop is executed. The execution then returns to the continuation expression, evaluates it, and, if `true`, the body of the loop is again executed. This cycle repeats until the continuation expression evaluates to `false`; the loop ends, and the execution commences after the end of the loop body.

There are two general types of continuation expressions used for looping statements, as follows:

- **Counter-controlled looping**, where the number of iterations is dependent upon a count of some kind. The desired number of iterations is known beforehand. The counter may be increasing or decreasing.
- **Condition- or sentinel-controlled looping**, where the number of iterations is dependent upon some condition to remain true for the loop to continue. The actual number of iterations is not known. A sentinel is a value that must attain a certain state before the loop completes.

We will explore counter-controlled looping in this chapter and return to sentinel-controlled looping in later chapters when we get input from the console and read input from files.

C also provides some additional looping control mechanisms such as `break`, which we saw in the `switch() ...` statement in Chapter 6, *Exploring Conditional Program Flow*, and `continue`. These provide even greater looping control when simple counters or sentinel values aren't enough to meet our needs in special circumstances.

To boost our motivation for iteration and repetition, let's visit a problem that was presented to the young, brilliant mathematician Gauss in the 17th century. When Gauss was in elementary school, to fill time before a recess, the teacher assigned the task of adding numbers 1 to 100 (or some such range). While all the other students were busily performing the tedious task of adding each number one by one (brute-force), young Gauss came up with a simple yet elegant equation to perform the task nearly instantly. His general solution was the following equation:

$$\text{sum}(n) = n * (n+1) / 2$$

Here, n is the highest number in a sequence of natural numbers (integers) beginning at 1.

So, throughout this chapter, we will explore the problem presented to the young Gauss—starting with his insightful equation and then moving on to various programmatic solutions—first, using brute-force addition, which Gauss cleverly avoided, then, performing the task using each of the looping constructs C provides, and, finally, looping with the dreaded `goto` statement.

The following code snippet shows Gauss's equation in a `sumNviaGauss()` C function:

```
int sumNviaGauss( int N ) {  
    int sum = 0;  
    sum = N * ( N+1 ) / 2;  
    return sum;  
}
```

The input parameter is N . The result is the sum of integer values 1 . . N . This function is a part of the `gauss_bruteforce.c` program and there are links in that program for delightful explanations of this equation, along with the variations of it, which we need not go into here. The curious reader can download `gauss_bruteforce.c` and explore the links given there.

Note that the $N * (N+1) / 2$ equation requires `()` because `*` and `/` have higher precedence than `+`. `()` has higher precedence than all the operators here, and thus gives us the desired result.

What is the point of providing this solution here? As C programmers, we have all of these wonderful C statements that we can use to construct complex calculations for solving a complex mathematical problem. However, we must remember that there may be an equation or *algorithm* that exists already that is much simpler and more generalized than anything that we may hope to concoct. For this reason, every programmer should be familiar with the *Numerical Recipes in X* books that provide complex mathematical solutions in the X language, where X is either C, Fortran, or C++, to some of the most demanding and challenging math problems that have vexed mathematicians, scientists, engineers, computer scientists, and operations researchers alike. Ignore such works at your peril!



As an aside, I should mention that some of the most interesting and useful algorithms I've ever encountered as a computer scientist have come from the operations research guys. They seem to be always attempting to solve some really difficult, yet important problems. But that is a topic out of scope for this book.

While young Gauss abhorred the use of brute force to solve the problem he was given, sometimes brute force may be the only way or even the best way, but not often. We examine that next.

Understanding brute-force repetition

In brute-force repetition, a statement or series of statements to be repeated is simply copied over and over the required number of times. This is the most restrictive form of repetition because the number of repeats is hardcoded in and can't be changed at runtime.

There are several other downsides to this type of repetition. First, what if you had to change one or more of the statements that have been copied over and over? Tedious would be the word to describe the work required to either change all of them (error-prone) or to delete, correct, and recopy the lines (also error-prone). Another downside is that it makes the code unnecessarily bulky. Copying 10 lines is one thing, but 100 or 1,000 is another thing altogether.

However, there are also times when copying a single statement multiple times is actually necessary. The situation where this occurs is in an advanced topic involving *loop unrolling*, which we will not cover in this book. If you are still interested after you have finished this chapter, you can perform your own internet search to find out more. It is, as a reminder, an advanced topic related to specialized high-performance situations. You will have many other *fish to fry* before—if ever—you will need to master that topic.

The `sum100viaBruteForce()` function is a brute-force function to perform our desired task, and is shown in the following code block:

```
int sum100bruteForce( void ) {
    int sum = 0;
    sum  = 1;
    sum += 2;
    sum += 3;
    ...
    ...
    sum += 99;
    sum += 100;
    return sum;
}
```

Notice that we do not include every single line of this over-100-line function. It is very tedious and dull. Yet, in fact, it correctly calculates the sum of 1 .. 100. This function only works for this sequence and no other. You'd need a different brute-force method to calculate 1 .. 10 or 1 .. 50. Yawn. Even more tedium.

Here is a second version, a bit more general, using some useful C operators—the `sum100viaBruteForce2()` function, illustrated in the following code block:

```
int sum100bruteForce2( void ) {
    int sum = 0;
    int num = 1;

    sum  =  num;
    sum += ++num;
    sum += ++num;
    sum += ++num;
    ...
    ...
    sum += ++num;
    sum += ++num; // 100

    return sum;
}
```

Notice, again, that we do not include every single tedious line of this over-100-line function. While this approach removes the need to actually type in each value from 1 to 100, it is equally tedious. I found that it was actually more difficult to create than the first version (that is, `sum100viaBruteForce()`) because it was hard to keep track of how many `sum += ++num;` lines I had copied. You'll see in the `gauss_bruteforce.c` file that I added comments to help keep things straightfoward and simple.

Each of these functions is over 100 lines long. That's like over 100 miles of dry, dusty, dull road to drive across on a hot, arid, boring day with no rest stops and no ice water. The compiler might not care but those who will later have to read/update your code will.

When you enter these functions yourself, it is acceptable in this case to use copy and paste to help overcome the tedium.

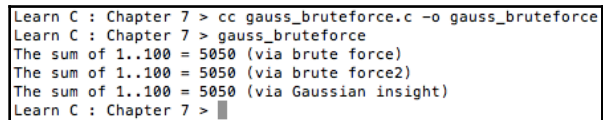
The `main()` function for `gauss_bruteforce.c` is as follows:

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

int sum100bruteForce( void );
int sum100bruteForce2( void );
int sumNviaGauss( int N );

int main( void ) {
    int n = 100;
    printf( "The sum of 1..100 = %d (via brute force)\n" ,
            sum100bruteForce() );
    printf( "The sum of 1..100 = %d (via brute force2)\n" ,
            sum100bruteForce2() );
    printf( "The sum of 1..%d = %d (via Gaussian insight)\n" ,
            n , sumNviaGauss( n ) );
    return 0;
}
```

Create the `gauss_bruteforce.c` file, then enter the `main()` function and the three `sum` functions. Compile the program with the `cc gauss_bruteforce.c -o gauss_bruteforce` command. The `-o` command option followed by a name generates an executable file with that name instead of `a.out` (the default executable name that we have been using before now). Run the program. You should see the following output from `gauss_bruteforce`:



```
Learn C : Chapter 7 > cc gauss_bruteforce.c -o gauss_bruteforce
Learn C : Chapter 7 > gauss_bruteforce
The sum of 1..100 = 5050 (via brute force)
The sum of 1..100 = 5050 (via brute force2)
The sum of 1..100 = 5050 (via Gaussian insight)
Learn C : Chapter 7 > █
```

In the preceding screenshot, you can see that each of the three methods to calculate the sum of 1..100 gives us the same result.

Thankfully, there are better ways (well, still not as good as Gauss's original solution, but better from a C programming perspective) to solve this problem, and—happily for us—to illustrate looping.

As we examine the various forms of repetitive methods, we will solve Gauss's problem each time. This approach affords a couple of advantages. As you work with loop iteration counters, any starting or stopping errors made in the loop counter condition will result in a different sum; so, by using the same problem that we have already solved, we can verify our code. Also, since we've already solved the problem in two ways, it will not be new, and will even become familiar. Therefore, we can focus more on the variations in the syntax of each looping method.

Introducing the `while()`... statement

`while () ... statement` has the following syntax:

```
while( continuation_expression ) statement_body
```

`continuation_expression` is evaluated. If its result is `true`, `statement_body` is executed and the process repeats. When `continuation_expression` evaluates to `false`, the loop ends; the execution resumes after the `statement_body`. If the `continuation_expression` initially evaluates to `false`, the `statement_body` loop is never executed.

The `statement_body` is—or may be—a single statement, or even the `null` statement (a single `;` without an expression), but most often, it is a compound statement. Note that there is no semicolon specified as a part of the `while () ... statement`. A semicolon would appear as a part of a single statement in a `statement_body`, or would be absent in the case of the `statement_body` consisting of a `{ ... }` compound statement.

Also note that, within the `statement_body`, there must be some means to change the value(s) used in the `continuation_expression`. If not, the loop will either never execute or it will never terminate once begun. The latter condition is also known as an **infinite loop**. Therefore, in counter-controlled looping, the counter must be changed somewhere in the body of the loop.

Returning to Gauss's problem, we'll use a `while () ... loop` in a function that takes `N` as a parameter, which is the highest value of the sequence to sum, and returns that sum of 1 to `N`. We need a variable to store the sum that is initialized to 0. We'll also need a counter to keep track of our iterations, also initialized to 0. The counter will have a range of 0 to (`N-1`). Our loop condition is: is the counter less than `N`? When the counter reaches the value of `N`, our loop condition will be `false` (`N` is not less than `N`) and our loop will stop. So, in the body of our loop, we accumulate the sum and we increment our counter. When looping has completed, the sum is returned.

The `sumNviaWhile()` function is shown in the `gauss_loops.c` program, as follows:

```
int sumNviaWhile( int N ) {
    int sum = 0;
    int num = 0;
    while( num < N ) // num: 0..99 (100 is not less than 100) {
        sum += (num+1); // Off-by-one: shift 0..99 to 1..100.
        num++;
    }
    return sum;
}
```

There is a bit of a wrinkle; this wrinkle is known as the *off-by-one problem*. This problem has many forms in other programming languages and not just in C. Notice that our counter starts at 0 and goes to N-1, to give us N iterations. We could start at 1 and check if the counter is less than N+1. Or, we could also start at 0 and test if the counter is less than or equal to N. This second approach would give a correct answer for this problem but would give us N+1 iterations instead of just N iterations. Starting at 0 and going to, say 10, would altogether be 11 iterations.

There is a valid reason we have chosen to start our counter at zero. It has more to do with C array indexes, which we will encounter in Chapter 11, *Working with Arrays*. This may seem confusing now, yet zero-based counting/indexing is a very consistent principle in C. Getting accustomed to it now will save many more headaches when working with array indexing and pointer addition later.

To help mitigate any possible confusion with counter ranges, I have found it is always helpful to indicate the expected range of values (that the counter or index will take) in comments. In that way, necessary adjustments, as done previously, can be made to any other calculations that use that counter.

On the other hand, there is another way. (There is always another way in C, as in most programming languages!) In this second way of implementing the `while()` ... loop, instead of counting up, we'll count down. Furthermore, we'll use the N input parameter value as the counter so that in this way, we don't need a separate counting variable. Remember that the function parameters are copied from the caller and also that they become local variables within the function then. We'll use N as a local variable to be our counter. Instead of incrementing our counter, we'll decrement it. The valid range will thus be from N down to 1. In this way, we'll let 0 be the stopping condition because it also evaluates to `false`.

Our `continuation_expression` is simply evaluating whether `N` is nonzero to continue. We could have also used `while(N > 0)`, which would be only slightly more explicit, even if redundant. In addition, we get some minor benefits of not having to deal with the off-by-one problem. Then, our counter is also an accurate representation of the value we want to add.

The revised `sumNviaWhile2()` function in the `gauss_loops2.c` program is shown in the following code block:

```
int sumNviaWhile2( int N ) {
    int sum = 0;
    while( N ) {          // N: N down to 1 (stops at 0).
        sum += N;
        N--;
    }
    return sum;
}
```

Is one approach better than the other? Not really. And certainly not in these examples, because our problem is rather simple. When the `statement_body` becomes more complex, one approach may be better in terms of clarity and readability than the other. The point here is to show how thinking about the problem in a slightly different way can make the code clearer sometimes. In this instance, the difference is in how the *count* is performed.

Introducing the `for()`... statement

The `for()`... statement has the following syntax:

```
for( counter_initialization ; continuation_expression ; counter_increment )
    statement_body
```

The `for()`... statement consists of a three-part control expression and a statement body. The control expression is made up of a `counter_initialization` expression, a `continuation_expression`, and a `counter_increment` expression, where a semicolon separates each part of an expression. Each one has a well-defined purpose. Their positions cannot be interchanged.

Upon executing the `for()`... statement, the `counter_initialization` expression is evaluated. This is performed only once. Then, `continuation_expression` is evaluated. If its result is `true`, the `statement_body` is executed. At the end of `statement_body`, the `counter_increment` expression is evaluated. Then, the process repeats, with the evaluation of `continuation_expression`. When `continuation_expression` evaluates to `false`, the loop ends; the execution resumes after the `statement_body`. If the `continuation_expression` initially evaluates to `false`, the `statement_body` loop is never executed.

The `statement_body` may be a single statement or even a null statement (a single `;` without an expression) but, most often, it is a compound statement. Note that there is no semicolon specified as a part of the `for()`... statement. A semicolon would appear as part of a single statement in the `statement_body` or would be absent in the case of the `statement_body` consisting of the `{ ... }` compound statement.

In the `for()`... statement, all of the control elements are present at the beginning of the loop. This design was intentional so as to keep all of the control elements together. This construct is particularly useful when the `statement_body` is either complex or overly long. There is no possibility of losing track of the control elements since they are all together at the beginning.

The `counter_increment` expression may be any expression that increments, decrements, or otherwise alters the counter. Also, when the counter is both declared and initialized within a `for` loop, it may not be used outside of that `for` loop's `statement_body`, much like the function parameters that are local to the function body. We will explore this concept in greater detail in Chapter 25, *Understanding Scope*.

Returning to Gauss's problem, we'll use a `for()`... loop in a function that takes as a parameter *N*, the highest value of the sequence to sum, and returns that sum of 1 to *N*. We need a variable to store the sum, initialized to 0. The counter we'll need will be both declared and initialized to 0 in the first part of the `for()`... statement. The counter will have the range of 0 to (*N*-1); when it reaches *N*, our loop condition *is the counter less than N?* will be `false` (*N* is not less than *N*) and our loop will stop. So, in the body of our loop, we need to only accumulate the sum. When looping has completed, the sum is returned.

The `sumNviaFor()` function in the `gauss_loop.c` program is shown in the following code block:

```
int sumNviaFor( int N ) {
    int sum = 0;
    for( int num = 0 ; num < N ; num++ ) { // num: 0..99 (it's a C thing)
        sum += (num+1); // Off-by-one: shift 0..99 to 1..100.
    }
```

```
    }  
    return sum;  
}
```

As we saw with the `while()` ... loop, we have encountered and had to deal with the off-by-one problem. But also, as before, there is a second way to perform this loop. In this second way of implementing the `for()` ... loop, instead of counting up, we'll count down. Again, we'll use the input parameter value (N) as the counter, so we don't need a separate counting variable. Remember that function parameters are copied from the caller, and also that they then become local variables within the function. We'll use N as a local variable to be our counter. Instead of incrementing our counter, we'll decrement it and let 0 be the stopping condition (as well as evaluate it to `false`).

As before, we get the somewhat minor benefit of not having to deal with the off-by-one problem. Then, our counter is also an accurate representation of the value we want to add.

The revised `sumNviaFor2()` function in the `gauss_loop2.c` program is shown as follows:

```
int sumNviaFor2( int N ) {  
    int sum = 0;  
    for( int i = N ; // range: 100..1  
        i > 0 ;      // stops at 1.  
        i-- ) {  
        sum += i;      // No off-by-one.  
    }  
    return sum;  
}
```

One final thing to notice in `sumNviaFor2()` is that the parts of the control expression are formatted such that each part is now on its own line. Doing this allows for more complex expressions and comments for each part.

For example, let's assume we want to simultaneously count up and down, using two counters. We can initialize more than one counter in the `counter_initialization` expression by using the `,` sequence operator. We can also increment more than one counter in the `counter_increment` expression, again by using the `,` operator. Our `for()` ... condition might look like this:

```
for( int i = 0 , int j = maxLen ;  
    (i < maxLen) && (j > 0) ;  
    i++ , j-- ) {  
    ...  
}
```

However, the indentation should be used to keep the control expression clearly identifiable from the loop body. In this simple example, such code formatting is unnecessary and should only be used where the control expression becomes more complex.

Introducing the `do ... while()` statement

The `do...while()` statement has the following syntax:

```
do statement_body while( continuation_expression );
```

The only difference between this statement and the `while()` statement is that, in the `do...while()` statement, `statement_body` is executed before `continuation_expression` is evaluated. If the `continuation_expression` result is true, the loop repeats. When `continuation_expression` evaluates to false, the loop ends. Note also the terminating semicolon. If the `continuation_expression` initially evaluates to false, the `statement_body` loop is executed once and only once.

Returning again to Gauss's problem, the similarities to the `while()` statement are clear. In fact, for this problem, there is a very little difference between the `while()` and `do...while()` statements.

The `sumNviaDoWhile()` function in the `gauss_loop.c` program can be seen in the following code block:

```
int sumNviaDoWhile( int N ) {
    int sum = 0;
    int num = 0;
    do {
        sum += (num+1);      // Off-by-one: shift 0..99 to 1..100.
        num++;
    } while ( num < N );    // num: 0..99 (100 is not less than 100).
    return sum;
}
```

Notice that, because the `statement_body` consists of more than one statement, a statement block is required; otherwise, a compiler error would result.

And, as we have already seen before, we can rework this function to use `N` as our counter and decrement it.

The `sumNviaDoWhile2()` function in the `gauss_loop2.c` program can be seen in the following code block:

```
int sumNviaDoWhile2( int N ) {
    int sum = 0;
    do {
        sum += N;
        N--;
    } while ( N );    // range: N down to 1 (stops at 0).
    return sum;
}
```

Before going any further, it's time to create not one but two programs, `gauss_loop.c`, and `gauss_loop2.c`.

The `main()` function of the `gauss_loop.c` program can be seen in the following code block:

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

int sumNviaFor(    int n );
int sumNviaWhile( int n );
int sumNviaDoWhile( int n );

int main( void ) {
    int n = 100;
    printf( "The sum of 1..%d = %d (via while() ... loop)\n" ,
           n , sumNviaWhile( n ) );
    printf( "The sum of 1..%d = %d (via for() ... loop)\n" ,
           n , sumNviaFor( n ) );
    printf( "The sum of 1..%d = %d (via do...while() loop)\n" ,
           n , sumNviaDoWhile( n ) );
    return 0;
}
```

Create the `gauss_loops.c` file, and enter the `main()` function, and the three `sum` function. Compile the program with the `cc gauss_loops.c -o gauss_loops` command. Run the program. You should see the following output from `gauss_loops`:

```
Learn C : Chapter 7 > cc gauss_loops.c -o gauss_loops
Learn C : Chapter 7 > gauss_loops
The sum of 1..100 = 5050 (via while() ... loop)
The sum of 1..100 = 5050 (via for() ... loop)
The sum of 1..100 = 5050 (via do...while() loop)
Learn C : Chapter 7 > □
```


In the preceding screenshot, you can see that each of the three looping methods to calculate the sum of 1..100 gives us the same result as well as the identical result from `gauss_bruteforce`.

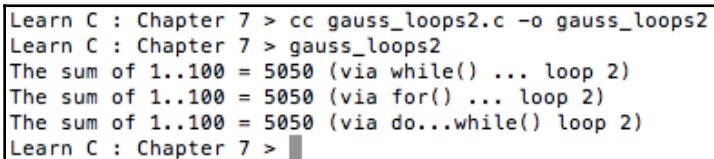
The main body of `gauss_loop2.c` is as follows:

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

int sumNviaFor2(    int N );
int sumNviaWhile2(  int N );
int sumNviaDoWhile2( int N );

int main( void ) {
    int n = 100;
    printf("The sum of 1..%d = %d (via while() ... loop 2)\n" ,
           n , sumNviaWhile2(n) );
    printf("The sum of 1..%d = %d (via for() ... loop 2)\n" ,
           n , sumNviaFor2(n) );
    printf("The sum of 1..%d = %d (via do...while() loop 2)\n",
           n , sumNviaDoWhile2(n) );
    return 0;
}
```

Create the `gauss_loops2.c` file, enter the `main()` function, and the three sum functions. Compile the program with the `cc gauss_loops2.c -o gauss_loops2` command. Run the program. You should see the following output:



```
Learn C : Chapter 7 > cc gauss_loops2.c -o gauss_loops2
Learn C : Chapter 7 > gauss_loops2
The sum of 1..100 = 5050 (via while() ... loop 2)
The sum of 1..100 = 5050 (via for() ... loop 2)
The sum of 1..100 = 5050 (via do...while() loop 2)
Learn C : Chapter 7 > █
```

In the preceding screenshot, you can see that each of the alternate three looping methods to calculate the sum of 1..100 gives us the same result as we have seen before.

Understanding loop equivalency

After having typed in both versions of each loop and run them, you may have begun to see some similarities between each of the looping statements. In fact, for counter-controlled looping, each of them is readily interchangeable.

To illustrate, let's examine each counter-controlled loop by comparing each of their essential parts.

The counter-controlled `while()` ... loop has the following syntax:

```
counter_initialization;
while( continuation_expression ) {
    statement_body
    counter_increment;
}
```

Notice that both counter initialization and counter increments have been added to the basic syntax of the `while()` ... loop and that they are somewhat scattered about.

The counter-controlled `for()` ... loop has the following syntax:

```
for( counter_initialization ; continuation_expression ; counter_increment )
    statement_body
```

It would be perfectly logical to assume that the `for()` ... loop is really just a special case of the `while()` ... loop.

The counter-controlled `do...while()` loop has the following syntax:

```
counter_initialization;
do {
    statement_body
    counter_increment;
} while( continuation_expression );
```

Notice that, as with the `while()` ... loop, the counter-control expressions have been added to the basic syntax of the `do...while()` loop and are also somewhat scattered about the loop.

For counter-controlled looping, the `for()` ... loop is a natural first choice over the other two. Nonetheless, any of these may be used. However, when we look at sentinel-controlled loops, these equivalencies begin to break down. We will see that the `while()` ... loop provides far more general use in many cases, especially when looking for a sentinel value to end continuation.

Understanding unconditional branching – the dos and (mostly) don'ts of goto

The `goto` statement is an immediate and unconditional transfer of program execution to the specified label within a function block. `goto` causes execution to *jump* to the label. In current C, unlike the bad old days, `goto` may not jump out of a function block, and so it may neither jump out of one function into the middle of another nor out of one program into another program (neither were uncommon in those days).

The `goto` statement consists of two parts. First, there must be a label declared either as a standalone statement, as follows—`label_identifier :—`or as a prefix to any other statement, like so: `label_identifier : statement.`

And secondly, there must be the `goto` statement to that `label_identifier`. The syntax for the `goto` statement is as follows:

```
goto label_identifier;
```

The reason for the `goto` statement being shunned comes from the *bad old days* before *structured programming*. The main tenet of structured programming was one entry point, one exit point. We don't hear much about this anymore because, well, programmers have been trained better and languages—starting with C—have become more disciplined, so that `goto` is not really needed. The main aim of the structured programming movement was to counter the spaghetti code of earlier programs, where `goto` ruled because a more disciplined mechanism did not exist, and the use of `goto` in some cases had got completely out of control. Programs jumped from here to there, and to anywhere, and code became extremely difficult to understand or modify (because the `goto` statement made it impossible to know all the possible flows of control). Often, the answer to the question, *How did we end up here?* or *What was the code path that got us to this point?* was not easily discernible, if it was discernible at all. Thanks to C, and subsequent derivative languages of C, the undisciplined use of `goto` was reined in.

The creators of C felt there was occasionally, albeit rarely, a need for `goto`, and so they left it in the language. In C, `goto` is highly constrained in terms of where it can—ahem—go to. Unlike the bad old days, you cannot use `goto` in a label inside another function. You cannot use `goto` out of the current function. You cannot use `goto` in another program, nor can you use `goto` somewhere in the runtime library or into system code. All these things were done and were often done for expediency, only without regard to the long-term maintainability of the code. Mayhem ruled. But no longer, at least with respect to `goto`.

Today, in C, the `goto` statement can only jump to a label within the same function. `goto` is extremely handy in the case of deeply nested `if... else...` statements or deeply nested looping statements when you just need to get out and move on. While this is sometimes necessary, it should be considered rarely necessary. It is also handy at times in high-performance computing. So, for those reasons alone, we consider it here.

Besides, C provides two other extremely useful and disciplined statements that rein in the undisciplined and chaotic use of `goto`, as we will see in the next section.

For the remainder of this section, we'll look at structured uses of `goto` and how to implement the looping statements we've already seen using `goto`. In each case, there is a pair of labels identifying the beginning and end of what in other statements would be the loop block.

In our first example, the end-of-loop label is not needed; it is there for clarity, as shown in the following example of the `sumNviaGoto_Do()` function of the `gauss_goto.c` program:

```
int sumNviaGoto_Do( int N )
{
    int sum = 0;
    int num = 0;
begin_loop:
    sum += (num+1);
    num++;
    if( num < N ) goto begin_loop;    // Go up and repeat: loop!
    // Else fall-through, out of loop.
end_loop:
    return sum;
}
```

In the `sumNviaGoto_Do()` function, we find all the elements of the preceding looping statements. There is the loop block beginning at the `begin_loop:` label. There is the loop block ending at the `end_loop:` label, and, in this example, the body of the loop block is executed exactly once before the loop condition is evaluated. This, then, is the `goto` equivalent of a `do ... while()` loop.

So, you might now be wondering what a `goto`-equivalent `while()` ... loop might look like. Here it is, in the `sumNviaGoto_While()` function of the `gauss_goto.c` program:

```
int sumNviaGoto_While( int N )
{
    int sum = 0;
    int num = 0;
begin_loop:
    if( !(num < N) ) goto end_loop;
```

```
    sum += (num+1);
    num++;
    goto begin_loop;
end_loop:
    return sum;
}
```

Notice how the loop condition had to be slightly modified. Also, notice when that loop condition is `true`, we go to the label that is after the `goto begin_loop` statement. This is the only way we get out of the loop, just as in the `while() ...` statement.

Finally, we can implement a `for() ...` loop with `goto`, as shown here in the `sumNviaGoto_For()` function of the `gauss_goto.c` program:

```
int sumNviaGoto_For( int N )
{
    int sum = 0;
    int num = 0;

    int i = 0;                      // Initialize counter.
begin_loop:
    if( !(i < N) ) goto end_loop;  // Loop continuation condition.
    sum += (num+1);
    num++;
    i++;                          // Counter increment.
    goto begin_loop;
end_loop:
    return sum;
}
```

To do this, we had to add a local counter variable, `i`, initialize it to 0, test that its value was not less than `N`, and, finally, increment it before we unconditionally branch to the top of our loop. You should be able to see how each of these statements corresponds to those in the `for() ...` statement.

In assembler language—a nearly direct translation to machine language—there is no `for()` ... , `while() ...`, or `do ... while()` loops. There is only `goto`. These `goto` looping constructs could very well be translated directly into either assembler language or directly to machine language. But we are programming C, so the point of demonstrating these constructs is to show the equivalence between the various looping mechanisms.

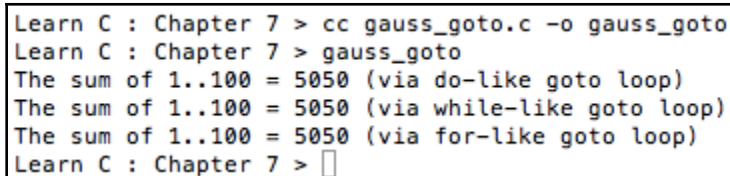
The `main()` function of the `gauss_goto.c` program is given as follows :

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

```
int sumNviaGoto_While( int N );
int sumNviaGoto_Do( int N );
int sumNviaGoto_For( int N );

int main( void )
{
    int n = 100;
    printf( "The sum of 1..%d = %d (via do-like goto loop)\n" ,
           n , sumNviaGoto_Do(n) );
    printf( "The sum of 1..%d = %d (via while-like goto loop)\n" ,
           n , sumNviaGoto_While(n) );
    printf( "The sum of 1..%d = %d (via for-like goto loop)\n" ,
           n , sumNviaGoto_For(n) );
    return 0;
}
```

Create the `gauss_goto.c` file, enter the `main()` function, and the three `sum` functions. Compile the program with the `cc gauss_goto.c -o gauss_goto` command. Run the program. You should see the following output:



```
Learn C : Chapter 7 > cc gauss_goto.c -o gauss_goto
Learn C : Chapter 7 > gauss_goto
The sum of 1..100 = 5050 (via do-like goto loop)
The sum of 1..100 = 5050 (via while-like goto loop)
The sum of 1..100 = 5050 (via for-like goto loop)
Learn C : Chapter 7 > █
```

In the preceding screenshot, you can see that each of the alternate three looping methods to calculate the sum of `1...100` gives us the same result as we have seen before.

So, now, the question before us is: *We can loop with `goto`, but should we?*

The answer is quite resoundingly—No! We don't need to, at all. We use `for()`... , `while()`..., or `do ... while()` instead!

These complex looping statements exist to make our code clearer as well as to obviate the need for `goto`. Let the compiler generate the `goto` statement for us. So, for general-purpose computing, `goto` should rarely be used, if ever. However, in certain high-performance computing situations, `goto` may be necessary.

Remember, the overuse and/or improper use of `goto` is a way to perdition! Use `goto` wisely.

Further controlling loops with break and continue

Rather than relying on `goto` to get out of sticky situations inside of deeply nested statements, the creators of C provided two very controlled goto-like mechanisms. These are `break` and `continue`.

`break` jumps out of and to the end of the enclosing statement block, whereas `continue` is used for looping, which goes immediately to the next iteration of the looping statement, skipping any statements that would otherwise be executed in the loop after the `continue` mechanism.

We have previously encountered the use of `break` in the `switch` statement in the preceding chapter, where `break` caused the execution to resume immediately after the `switch` statement block. `break` can also be used to jump to the end of the enclosing `statement_body` loop.

In the following `isPrime()` function, `break` is used to get out of a loop that determines if the given number is divisible by the counter value; if so, the number is not prime.

The `isPrime()` function of the `primes.c` program can be seen in the following code block:

```
bool isPrime( int num ) {
    if( num < 2 )    return false;
    if( num == 2 )  return true;

    bool isPrime = true;    // Make initial assumption that num is prime.
    for( int i = 2 ; i < num ; i++ ) {
        if( (num % i) == 0 ) { // We found a divisor of num;
                               // num is not prime.
            isPrime = false;
            break;           // No need to keep checking; leave the loop.
        }
    }
    return isPrime;
}
```

Here, we are demonstrating `break` in a rather simple example. In this case, you may recall, we could also have simply used `return false` instead, but where's the fun in that? Because `break` is not in a `switch` but is in a loop, `break` takes the execution out of the loop to the very next statement after the closing loop `}` bracket.

The `continue` statement only works within an enclosing `statement_body`. When encountered, the execution jumps to immediately before the closing loop `}` bracket, thereby commencing on the next `continuation_expression` loop and possible loop iteration (only if the continuation evaluates to `true`).

Let's say we want to calculate a sum for all prime numbers between 1 and N as well as all non-prime numbers in the same range. We can use the `isPrime()` function within a loop. If the candidate number is not prime, do no more processing of this iteration and begin the next one. Our function that adds only prime numbers would look like this `sumPrimes()` function of the `primes.c` program:

```
int sumPrimes( int num ) {
    int sum = 0;
    for( int i = 1 ; i < (num+1) ; i++ ) {
        if( !isPrime( i ) ) continue;

        printf( "%d " , i);
        sum += i;
    }
    printf("\n");
    return sum;
}
```

Similarly, a function that adds only non-prime numbers would look like this `sumNonPrimes()` function of the `primes.c` program:

```
int sumNonPrimes( int num ) {
    int sum = 0;
    for( int i = 1 ; i < (num+1) ; i++ ) {
        if( isPrime( i ) ) continue;

        printf( "%d " , i);
        sum += i;
    }
    printf("\n");
    return sum;
}
```

Care must be exercised when using the `continue` statement to ensure that the loop counter update is performed and not bypassed with the `continue` statement. Such oversight would result in an infinite loop.

The `main()` function of `primes.c`, which illustrates `break` and `continue`, does three things. First, it does a simple validation of our `isPrime()` function using a `for()`... loop. Then, it calls `sumPrimes()` via a `printf()` function, and, finally, it calls `sumNonPrimes()` again via a `printf()` function. If the program logic is correct, the sum of both prime and non-prime numbers should be the same as our preceding summing functions; that is how we will verify the correctness of the program. The `main()` function of the `primes.c` program is given as follows:

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

bool isPrime( int num );

int sumPrimes( int num );
int sumNonPrimes( int num );

int main( void ) {
    for( int i = 1 ; i < 8 ; i++ )
        printf( "%d => %sprime\n", i , isPrime( i ) ? " " : "not " );
    printf("\n");
    printf( "Sum of prime numbers 1..100      = %d\n" ,
            sumPrimes( 100 ) );
    printf( "Sum of non-prime numbers 1..100 = %d\n" ,
            sumNonPrimes( 100 ) );
    return 0;
}
```

Create and type in the `primes.c` program. Compile, run, and verify its results.

Create the `primes.c` file, enter the `main()` function, the `isprime()` function, and the two `sumPrime` functions. Compile the program with the `cc primes.c -o primes` command. Run the program. You should see the following output:

```
Learn C : Chapter 7 > cc primes.c -o primes
Learn C : Chapter 7 > primes
1 => not prime
2 => prime
3 => prime
4 => not prime
5 => prime
6 => not prime
7 => prime

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Sum of prime numbers 1..100      = 1060
1 4 6 8 9 10 12 14 15 16 18 20 21 22 24 25 26 27 28 30 32 33 34 35 36 38 39 40 4
2 44 45 46 48 49 50 51 52 54 55 56 57 58 60 62 63 64 65 66 68 69 70 72 74 75 76
77 78 80 81 82 84 85 86 87 88 90 91 92 93 94 95 96 98 99 100
Sum of non-prime numbers 1..100 = 3990
Learn C : Chapter 7 > █
```

In the preceding screenshot, you can see that for each function, we first validate that the `isPrime()` function properly determines the primeness of one through seven. Not only can you see the sum of prime numbers and non-prime numbers, but you can also see the numbers in each set, for further verification. Note that $1060 + 3990 = 5050$ is the expected correct result.

For a complete comparison of `break`, `continue`, `return`, and `goto`, consider the following code outline:

```
int aFunction( ... ) {
    ...
    for( ... ) { /* outer loop */
        for( ... ) { /* inner loop */
            ...
            if( ... ) break;          /* Get out of inner loop. */
            ...
            if( ... ) continue;      /* Next iteration of inner loop. */
            ...
            if( ... ) goto ERROR;    /* Get out of ALL loops. */
            ...
            /* Next statement after continue; */
            /* Also next iteration of inner-loop. */
        }
        /* Next statement after break; still in outer-loop. */
        ...
    }
    return 0; /* normal function exit */

ERROR:      /* Error recovery */
    ...
    return -1; /* abnormal function exit */
}
```

In this outline, there is an inner `for()` ... loop nested within an outer `for()` ... loop. `break` will only go to the end of its immediate enclosing `statement_body`. In this case, it goes to the end of the inner loop block and executes statements in the outer loop block. `continue` goes to the point immediately before the end of the enclosing `statement_body` to repeat the loop iteration. `goto ERROR` immediately jumps to the `ERROR:` label at the end of the function body and handles the error condition before returning from the function. In the statement before the `ERROR:` label, there is a `return 0` that returns from the function, thus preventing the execution of the error recovery statements.

Understanding infinite loops

So far, we have considered loops that have an actual end. In most cases, this is both intended and desirable. When loops never end, either unintentionally because we goofed up somewhere or intentionally, they are called an **infinite loop**. There are a few special cases where an infinite loop is actually intentional. The cases are as follows:

- When the user interacts with the program until the user chooses to quit the program
- When there is input with no known end, as in networking where data can come at any time
- Operating system event loop processing. This begins upon boot-up and waits (loops) for events to happen until the system is shut down.

When you start a program that accepts user input—keyboard strokes, mouse movements, and so on, it goes into an infinite loop to process each input. We would then need to use a `break`, `goto`, or `return` statement in the statement-body of our infinite loop to end it.

A simplified version using `for()` ... might look something like the following:

```
void get_user_input( void )
{
    ...
    for( ; ; )
    {
        ...
        if( ... ) goto exit;
        ...
        if( cmd == 'q' || cmd == 'Q' ) break;
        ...
    }

    exit:
    ... // Do exit stuff, like clean up, then end.
}
```

When `continuation_expression` is null, it is evaluated to `true`. The other parts of the control expression are optional.

The computer's main routine, after it loads all of its program parts, might look something like this:

```
void system_loop( void )
{
    ...
}
```

```
while( 1 )
{
    ...
    getNextEvent();
    handleEvent();
    ...
    if( system_shutdown_event ) goto shutdown;
    ...
}
shutdown:
... // Perform orderly shut-down activities, then power off.
}
```

This is an extremely oversimplified version of what actually goes on. However, it shows that somewhere in your computer, an infinite loop is running and processing events.

Summary

We have encountered various repetitious looping techniques, from the ridiculous (brute-force iteration) to the sublime (various loop statements with `break`, `continue`, and `goto`). With functions, conditional expressions, and—now—looping statements, we conclude our journey through C flow-of-control statements. Nearly all of these concepts can be easily translated into other programming languages.

In the bulk of the remainder of the book, we'll broaden our understanding and ability to manipulate data well beyond the simple forms we have so far encountered. In the next chapter, we will explore custom named values called **enumerations**.