THINGS TO LOOK FOR…

- C Arithmetic and logical operators.
- Flow of control constructs supported by the C language.
- An understanding of branching type constructs.
- An understanding of the structure and control of looping type constructs.
- Proper use of the break and continue statements.
- An understanding of the differences between entrance and exit condition loops.
- Good design and coding styles.

2.0 INTRODUCTION

In this tutorial we will open with a review the C language arithmetic and logical operators then follow with a brief study of the constructs by which we can alter and control the flow through a program. Such constructs include relational expressions as well as those that control branching, jumping, and looping within an application. As the different topics are presented, we will suggest techniques and methodologies that can help one to learn and practice good coding style and to design and develop more robust programs.

In an embedded application, the C operators permit us to perform various kinds of computations on data that we collect either directly or through measurements of outside world signals and events. Relational expressions support the ability to test or compare any returned values then to make decisions based upon the results of such comparisons. The branching, jumping, and looping constructs enable the designer to control how, when, and how often sets of instructions are executed based upon some specified condition(s). Branching constructs permit the selection of several different paths of execution. Jumps permit the program flow to change from executing one set of code to a completely different set; loops permit the application to perform the same set of instructions repeatedly.
2.1 C Operators

The variables in a programming language allow us to express different kinds of data and information about or used by an embedded application. It is the relationships supported by that language between and among that data and information that enable us to develop and build powerful applications using the language. The power of a language derives from the operators that it provides in support of those relationships. The C language supports a rich set of capabilities through the various kinds of operators listed in Figure 2.0.

| • Arithmetic Operators  
| • Logical Operators    
| • Bitwise Operators   |

Figure 2.0 General Categories of C Operators

2.1.1 Introducing the C Language Operators

Most of the C language operators perform the operations that we might expect. Our objective will be to examine how they apply in embedded applications, look at some techniques whereby one can improve their performance in such a context, and identify potential problem areas.

2.1.2 The C Arithmetic Operators

C uses the four familiar arithmetic operators to perform addition, subtraction, division, and multiplication operations on variables. The operators, which are the same for both integral and floating point types, are also those we expect to find on a typical calculator. These are given in Figure 2.1 and they work as we would expect.

2.1.2.1 Operator Precedence and Associativity

Let’s take a look at what appears to be a simple arithmetic problem. If we’re not careful, however, this problem can give us some rather interesting and probably undesired results.

Precedence

Let’s start by declaring some variables

```c
int a = 20;
int b = 25;
int c = 0;
```

We then write

```c
c = a * 2/b + 15;
```

The variable `c` is...

- `a` times 2 ... divided by `b` ... plus 15
…which is really 40 divided by 40.

One might expect the value of the variable \( c \) to be 1. However, if these four lines of code are included in a program which is then compiled and run, the result will print as the number 16. What happened? Why did that happen? The answer to these questions lies in a concept called operator precedence or simply precedence. The ANSI/ISO C language standard specifies both C Operator precedence and associativity. These are given in Table 2.0.

**Table 2.0**

### C Operator Precedence and Associativity

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>()  []  .  -&gt;  ++  --</td>
<td>Parentheses (grouping) Subscript Member selection direct Member selection indirect Unary postfix increment / postfix decrement</td>
<td>left</td>
</tr>
<tr>
<td>2</td>
<td>++  --  +  -  !  ~  *  &amp;  sizeof</td>
<td>Unary prefix increment / prefix decrement Unary plus / minus Unary logical negation Unary bitwise complement Dereference Address of Return size in bytes</td>
<td>right</td>
</tr>
<tr>
<td>3</td>
<td>(type)</td>
<td>Unary cast (change to type)</td>
<td>left</td>
</tr>
<tr>
<td>4</td>
<td>*  /  %</td>
<td>Multiplication / division / mod</td>
<td>left</td>
</tr>
<tr>
<td>5</td>
<td>+  -</td>
<td>Addition / subtraction</td>
<td>left</td>
</tr>
<tr>
<td>6</td>
<td>&lt;&lt;  &gt;&gt;</td>
<td>Bitwise shift left / shift right</td>
<td>left</td>
</tr>
<tr>
<td>7</td>
<td>&lt;  &lt;=  &gt;  ==  &gt;=</td>
<td>Relational less than / less than or equal to Relational greater than / greater than or equal to</td>
<td>left</td>
</tr>
<tr>
<td>8</td>
<td>==  !=</td>
<td>Relational is equal to / not equal to</td>
<td>left</td>
</tr>
<tr>
<td>9</td>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>left</td>
</tr>
<tr>
<td>10</td>
<td>^</td>
<td>Bitwise Exclusive OR</td>
<td>left</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>Bitwise inclusive OR</td>
</tr>
<tr>
<td>12</td>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>left</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>?  :</td>
<td>Ternary conditional</td>
<td>right</td>
</tr>
<tr>
<td>15</td>
<td>=  +=  -=  *=  /=  &amp;=  &amp;=  ^=</td>
<td>=  &lt;&lt;=  &gt;&gt;=</td>
<td>Assignment Shortcut addition / subtraction Shortcut multiplication / division Shortcut mod / bitwise AND Shortcut bitwise exclusive / inclusive OR Shortcut bitwise shift left / right assignment</td>
</tr>
<tr>
<td>16</td>
<td>,</td>
<td>Comma</td>
<td>right</td>
</tr>
</tbody>
</table>
Precedence specifies the order in which operators are evaluated and associativity establishes whether we work with the operator to the left or to the right in a chained operation.

Let’s revisit the earlier arithmetic expression. From the table, we see that the division operator appears above (has higher precedence than) the addition operator. This means that the compiler will perform the division before the addition. However, multiplication is listed before division (associativity), therefore, it will perform multiplication before division. We will examine associativity in greater detail momentarily.

In the light of precedence,
1. \(a\) is multiplied by 2 first based upon associativity.

This yields 20 times 2 which is 40.

2. Next 40 is divided by \(b\) because division comes before addition.

This is 40 divided by 25 which is 1.6.

3. Finally, 15 is added to 1

To give 16

If it’s necessary for the evaluation to proceed in a different order, one can always override the precedence by using parentheses. To force \(b + 15\) to be done before the division, enclose those terms in parentheses. We now have,
\[
c = a \times \frac{2}{(b + 15)};
\]

Using parenthesis to enclose an operation, changes the evaluation order because the grouping or parentheses operator has higher precedence than any arithmetic operator which means that any expression inside of the parentheses must be evaluated first. Parentheses can be nested to any depth to achieve whatever evaluation order is needed or desired.

The expression is now evaluated as follows,
1. 15 is added to \(b\) because the parentheses must be evaluated first.

This is 25 plus 15 which is 40.

2. Next \(a\) is multiplied by 2 because multiplication precedes division.

This is 20 times 2 or 40.

3. Finally 40 is divided by 40 to give 1.

Associativity

A reasonable question to ask at this point is, if precedence determines which operator is applied first, what happens when all of the operators in a line of code have the same precedence?
Let’s look at the following equation
\[ d = a + b - c; \]

Does the compiler evaluate \( a + b \) or \( b - c \) first? The order is determined by operator associativity. If we look up addition and subtraction in the table above, we will see the associativity is left. This means the operator on the left is evaluated first. In the above line, \( a + b \) will be evaluated first then \( c \) will be subtracted. This is important to remember, because it can affect our mathematical calculations and can produce bugs that are very difficult to find.

**Coding Style**

It is good practice to always use parentheses to ensure a preferred evaluation order.

**Integer Division**

In the arithmetic example above, there was one step in which 40 is divided by 25. The expected answer is 1.6. Since integers do not have fractional parts, the 0.6 is simply truncated. In C, 40 divided by 25 returns 1 because there is only one 25 in 40 with a remainder of 15. Because the two operands are integers, the compiler expresses the result as an integer and 0.6 can’t be expressed as an integer.

If the application needs non integer numbers, certainly one solution is to use floating-point variables. However, in embedded applications, one would rather not bring in the floating point math package unless it’s absolutely necessary. For simple calculations such as this one, it’s much more efficient to take a page from the early computer designer’s book and use scaling.

**The Modulus Operator %**

Recall from the earlier discussion of integer types that integer division gives a whole number part and a fractional part (which could be 0). Such an operation is important enough in mathematics that it’s been given a special name – modulus and operator symbol \(-\%\). The operator is also called the remainder operator, modulus operator, or mod operator. The next example illustrates how the mod operator works.

**Example 2.0**

Let’s apply the modulus operator to the integers in following code fragment,

```c
int aNum = 15;
printf("aNum % 3 is %d \n", aNum %3); // prints 0
// ...3 divides 15 five times with 0 remainder
printf("aNum % 5 is %d \n", aNum %5); // prints 0
//...5 divides 15 three times with 0 remainder
printf("aNum % 10 is %d \n", aNum %10); // prints 5
//...10 divides 15 one time with 5 remainder
printf(" aNum % 2 is %d \n", aNum%2); // prints 1
// ...2 divides 15 seven times with 1 remainder
```
The modulus operator can be used as part of an algorithm to extract the hundreds, tens, and units digits from an integer.

```c
int myInteger = 987; // an integer value
int hundreds = myInteger/100; // gives 9 hundreds
int tens = (myInteger – hundreds*100) / 10; // gives 8 tens
int units = myInteger%10; // gives 7 units
```

### 2.1.3 Logical Operators and Logical Expressions

Logical operators provide another means by which one can specify relationships between and among variables. Such operators permit one to express in software the same logical relationships that are implemented in hardware using combinational logic gates.

In C, there are three common *logical* operators: **AND**, **OR**, and **NOT**.

> **Caution**: One must take care to distinguish the *logical* operators from the *bitwise* operators of similar name.

The logical operators work on expressions while the bitwise operators (discussed in the text) apply the same kind of operations to individual bits.

Like the hardware analog, the AND relationship evaluates to *true* when *all conjuncts* evaluate to *not false*. This is a subtle, but important point here. The C language does not have the notion of true and false in a Boolean sense. In the language, such variables are integral values rather than *booleans* as in C++. To that end, in C, *false* is 0 and *true* is not *false* – be careful not to read true as 1.

The OR relationship evaluates to *true* when *any disjunct is not false* – be careful not to read this as 1. The NOT relationship evaluates to *true* when its operand is *false* and vice versa.

**The Logical AND Operator &&**

Let’s consider the code fragment in Figure 2.2,

```c
int a = 5;
int b = 5;
if ( 5 == a)
{
   if (5 == b)
   {
      printf( "a AND b are both 5\n\n");
   }
}
```

---

**Example 2.1**

The modulus operator can be used as part of an algorithm to extract the hundreds, tens, and units digits from an integer.

```c
int myInteger = 987; // an integer value
int hundreds = myInteger/100; // gives 9 hundreds
int tens = (myInteger – hundreds*100) / 10; // gives 8 tens
int units = myInteger%10; // gives 7 units
```
The two *if* statements can be rewritten using a logical *AND* expression as is illustrated *if* in Figure 2.3. If we do so, we have,

```
if (( 5 == a) && (5 == b))
{
    printf( "a AND b are both 5\n");
}
```

*Figure 2.3 Using the Logical AND Operator*

Expressing the relation as was done above has provided more readable code and more closely reflects the intent of the author.

**Caution:** It’s important to understand how such an expression is evaluated. The left hand operand is evaluated first. If it evaluates to false, that is ‘0’, the AND operation cannot be true; therefore, there is no reason to evaluate any of the remaining operands. This is different from the way a hardware AND gate works.

An unintended side effect is that if the value of any of the remaining operands is determined by evaluating a function or other code fragment and other program variables depend upon that evaluation occurring, there is a problem. None of the dependent variables will be updated.

**Coding Style**

Notice that the comparison in the *if* construct above tests *(5==a)* rather than *(a==5)*. We make the comparison with the constant as the left hand operand so as not to inadvertently assign the number 5 to the variable a by unintentionally writing a = 5; As written, if we drop one of the ‘*==*’ symbols, the compiler will complain that 5 = a is an illegal assignment.

Always use your tools to help you.

Consider the code fragment in Figure 2.4.

```
int a = 5;
int b = 6;
int c = 7;
.....
if (( 5 == a) && (5 == b) && (5 == f(c)))
{
    printf("a AND b AND f(c) are 5 \n");
}
```

*Figure 2.4 Working with the Logical AND Operator*
Because \((5 == b)\) evaluates false, the expression \((5 == f(c))\) is not evaluated. Depending upon the value of ‘b’ at the time the AND is evaluated, processing may be different from that which is expected. In the above case, the function \(f(c)\) is never evaluated.

### Coding Style

Never depend upon side affects – the affecting code may never be executed.

---

The Logical OR Operator \(||\)

Now let’s look at the logical OR operator. We’ll begin with the code fragment in Figure 2.5.

```c
int a = 5;
int b = 5;
if (5 == a)
{
  printf("a OR b is 5\n");
}
else
{
  if (5 == b)
  {
    printf("a OR b is 5\n");
  }
}
```

Figure 2.5

Testing the Disjunction of Several Statements

As we did with the AND, the fragment can be rewritten using the logical OR operator as we see in Figure 2.6.

```c
if ((5 == a) || (5 == b))
{
  printf("a OR b is 5\n");
}
```

Figure 2.6

Working with the Logical OR Operator

As demonstrated earlier, rewriting gives more readable code. Again note that the order of evaluation of the expressions is left to right. Here if \((5 == a)\) is not false, evaluation will stop. There is no need to continue evaluating. As a result, the same cautions apply: don’t depend upon side affects.
To see this, let’s expand the OR expression in the code fragment in Figure 2.7.

```c
int a = 4;
int b = 5;
int c = 6;

if ((5 == a) || (5 == b) || (5 == f(c))
{
    printf("a OR b OR f(c) is 5\n");
}
```

Figure 2.7
Working with the Logical OR Operator

Here (5 == a) is false. Evaluation continues because only one expression must be true. Next (5 == b) evaluates true. At this point, further processing stopped. The function f(c) is never evaluated.

The Logical NOT Operator !

The character ! is the logical not or negation operator; the symbol is often read as ‘bang’. The code fragment in Figure 2.8 illustrates how the operator is used when testing if a condition is not satisfied.

```c
int a = 5;
int b = 5;
if (!(5 == a))
{
    printf("a OR b is NOT 5\n");
}
else
{
    if (!(5 == b))
    {
        printf("a OR b is NOT 5\n");
    }
}
```

Figure 2.8
Testing the Disjunction of Several Negated Statements

Using the same if statements from the earlier OR example, we change what is displayed when either expression is false rather than true. When an expression evaluates to true, the NOT operator inverts the sense to false. Common coding styles for the NOT are,

```c
if (0 != x) // reads if x not equal to zero
if (!(0 == x) // reads if (it is not true that x is equal to zero )
```
2.2 Getting Started

As a program is executing, the flow of control through the set of instructions can take a variety of paths, repeating some, skipping others, or just moving ahead. We now examine each of these to understand how they work and how they can be effectively used in embedded applications.

2.3 Sequential Flow of Control

When a set of firmware executes instructions sequentially, each instruction is selected, evaluated, and performed one following another, in sequence – as the name suggests. Each instruction is completed before the machine proceeds to next. Such a flow is illustrated graphically in Figure 2.9.

An instruction may be *simple* such as the following,

```
expression;
    a = b;
    a = sqrt (c);
```

or it may be *compound* meaning that it comprises a set (or block) of simple expressions, enclosed in `{}`. In contrast to most other statements in C, such a block of instructions is not terminated with a ‘;’. The curly brackets specify that the enclosed group of expressions is to be treated as a set or block. The language specifies that a compound expression can be used anywhere that a simple expression is used. *Flow of Control* proceeds in order, executing one statement after another, from top to bottom as we see in Figure 2.9.

2.4 The break and continue Statements

The *break* and *continue* statements are used to alter the flow of control inside of loops; the *break* performs the same function inside of switch statements.

**break**

1. Causes the execution of smallest enclosing *while*, *do*, *for*, or *switch* block to be terminated.

2. Execution resumes at the point immediately beyond terminated block - the block is exited.

**continue**

1. Causes execution of smallest enclosing *while*, *do*, *for*, or *switch* block to be terminated.

2. Execution resumes at the point immediately at the end of the terminated block - the block is not exited.
2.5 Conditional - Branch

With branching ability, the sequential flow construct is extended to permit the selection of alternate paths of execution (at some point in the flow) based upon the value of a specified variable or expression. Following the branch, flow resumes sequential execution on the new path as shown in Figure 2.10.

ANSI/ISO C supports two forms of branch construct: the familiar if-else and a multi-way switch. In either case, the flow of control can be represented schematically as in the graphic.

2.5.1 If-Else Construct

The if-else construct enables the selection from several alternative execution paths through the program based upon the value of a specified variable or upon a value returned from a function evaluation or I/O port access. The structure is expandable so as to permit a more complex structures to be built from simple components. The basic flow of control follows the template given in Figure 2.11. Starting with such a structure as a base, an embedded application can be designed to support a rich and sophisticated set of complex decision capability.

2.5.1.1 The Simple Branch

Conditioned on the results of the evaluation of the control expression, the basic form of the branch construct permits the selection of an alternate path of execution at the decision point. If the control expression evaluates to zero, the body of the if block is skipped; otherwise, the statements contained in the body are evaluated before proceeding on the original path.

Coding Style

If the control expression evaluates to non zero, only the first line of code following the if statement is evaluated. If it’s necessary that a block of statements be evaluated, they must be enclosed in curly brackets. So grouped, they are treated as a single statement.

It is good practice to always enclose the body of the if construct in a set of brackets even if it comprises only a single line. Such a practice will help to prevent difficult to find bugs.
2.5.1.2 The Two-Way Branch and More

With the simple branch shown in Figure 2.12a, we are given the choice of executing a designated block of code prior to continuing on the original path. We can extend that construct by more tightly controlling what happens should the if portion fail; that is, evaluate if to 0.

![Syntax](image)

At first such a construct doesn't seem to offer anything beyond the basic if construct, however, let's look at the code fragments in Figure 2.12b. Do they do the same thing?

```
case 1:
  if (x > 0)
    printf("Big\n");
  printf("Small\n");

case 2:
  if (x > 0)
    printf("Big\n");
  else
    printf("Small\n");
```

What if x evaluates to 3, what prints?

- In case 1, the printing of Small is independent of the value of x – both Big and Small print.
- In case 2, only Big or Small prints based upon the value of x. If x has the value 3, only Big will print.

One can extend the basic if-else construct as follows. First, write statement1 under the else portion as we see in the first code fragment in Figure 2.13 that is, express statement1 as an if-else construct itself. Following the substitution, the control flow appears as we see in the lower right hand code fragment. The construct can be similarly extended to imple-
ment an n-way branch. Care must be taken with such constructs as the next program demonstrates.

<table>
<thead>
<tr>
<th>if (expression_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement_0;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>statement_1;</td>
</tr>
</tbody>
</table>

Let statement_1 = if (expression_a)
| statement_a                      |
| else                             |
|     statement_b                  |

Figure 2.13
The Basic and Extended If-Else Constructs

Example 2.0

Let’s analyze the following bit of code in Figure 2.14.

```
/*
 * Conditional Statements - A mismatched else statement.
 */
#include <stdio.h>
void main (void)
{
    // declare some working variables
    unsigned int aValue = 0;
    unsigned int min = 3;
    unsigned int max = 10;

    // get some data
    printf( "Enter number: ");
    aValue = getchar();

    // convert from ASCII to decimal
    aValue -= '0';

    // Test for in range
    if ((aValue >= min) && (aValue <= max))
        // Test for even number
        if ((aValue%2) == 0)
            printf("aValue is even\n");

    // Out of range error
    else
        printf( "Number out of range\n" );
}
```

Figure 2.14
Working with the Basic If-Else Construct
Does this code fragment in Figure 2.15 print what is expected if the number 5 is entered for the variable aValue?

![Code fragment]

The indentation suggests that the else goes with the outer if. However, the compiler associates the else with the inner if.

If the designer does not specify otherwise using curly brackets, the compiler always associates an else with the most recent if. Such a mismatch can be very difficult to find in a complex program.

Re-stressing, it’s always best to use {} to specify the actual intention of the design.

Observe that when one implements a multiway decision block using a cascade of if-else statements, it’s not necessary that the control expression be the same for each decision. Each if and each else could be evaluating a different expression.

Although syntactically correct, the compound if-else construct can require a more complex set of instructions at the assembly language level which can then lead to larger, more complicated, and slower machine code. The switch construct, that we will study next, presents a more efficient alternative. For an embedded application, this can be important when we are trying to optimize memory size or execution speed.

### 2.6 The Switch Statement

When a multiway branch is based upon different values for the same control expression, the switch statement is a better choice than the if-else. The switch statement is similar to the case statement in Pascal or ADA.

The syntax for the switch is given as shown in Figure 2.16. The control expression must be an integral type. Within the body of the switch, the case label must also be an integral type; it must be a constant expression such as 1, 2, 3 or a string that evaluates to a constant. The default label is the keyword default. Within each case, statement can be either simple or compound.
The execution of the switch proceeds as follows,

1. The control expression evaluated
2. If the value of the control expression is equal to one of the case labels, control is transferred to the point indicated by the label and execution continues from that point.
3. If the value is not equal to any label and the default label is present, control is transferred to the default.
4. If the value is not equal to any label and the default label is not present, control is transferred to the statement after the switch block.

Caution: If the value of the control expression is equal to one of the case labels, control transferred to point indicated by the label and execution continues from that point and will continue sequentially through the remainder of the switch body.

If only the code associated with a single case is intended to be executed, a break statement must be inserted at the point where sequential execution is to be terminated.

The code fragment in Figure 2.17 illustrates the affect of the *break* statement. It is also important to remember that the break only redirects the flow of control to the right hand side of the closing curly bracket of the switch body. It does not alter the flow out of a context that may be enclosing the switch such as a loop or a function.
Let’s now look at an example of using the switch statement. In the program in Figure 2.18, the user is prompted to enter a specific value. Flow of control branches to a different execution path based upon the value that was entered.

If the value 3 is entered, control is transferred to the case 3 label and execution proceeds from that point. First, 3 stars will be printed followed by a newline, then the `break` causes the execution of the switch statement to be terminated. Control is transferred to the point just beyond terminated statement; in this case, the closing curly bracket. If the value 5 is entered, no stars are printed and the default case will be executed and the word `done` will be printed.
2.7 Loops and Iterations

The graphic in Figure 2.19 illustrates the flow of control in the loop construct. When we design a loop, the intention is that one statement or block of statements is executed repeatedly based upon some specified criteria. As is seen, one can make the decision to terminate the loop either before or after executing the block of statements. Thus, based upon the design of the loop, the number of iterations may be none, one, many, or unlimited.

The C language defines two forms of the loop construct. The first type is called an entry-condition loop and the second is called an exit-condition loop. When either type exe-
cutes, the *control expression* is tested on every *iteration* (or *cycle*) of the loop. In support of these constructs, the C language includes 3 types of loop: *do*, *while*, and *for*.

### 2.7.1 The Entry-Condition Loop

An entry-condition loop tests the condition upon entering the loop. To execute the code inside the loop, the condition must be *true*. Recall that in C *true is not false* and *false is zero*. Thus, to execute the code inside the loop, the condition must not be zero. If the condition is not true, the loop is skipped. With such loops, there is the possibility that the code inside the loop will never be executed. Entry-condition loops use the C keywords *while* and *for*. Such loops are usually referred to as *while loops* or *for loops*.

To set up an entry-condition loop one must,
- Set a control expression
- Test the expression to see if the loop should execute
- Re-test the expression to see if the loop should execute again

#### 2.7.1.1 The while Loop

The simplest example of an entry-condition loop construct is the *while* loop shown in Figure 2.20. The *loop body* is delimited by the pair of curly brackets following the *while* keyword. Such a construct means that a loop can contain multiple statements thereby permitting one to build loops of arbitrary complexity. Observe that if *control expression* is initially false, then *statement* is never executed.

**Syntax**

```
while (control expression)
{
    statements
}
```

*Figure 2.20 The while Loop*
Execution proceeds as follows.

1. The control expression evaluated
2. If exp != zero
   Evaluate the statement(s)
3. Repeat the process
4. Execution is complete when exp evaluates to zero

One can affect an extraordinary exit from the while loop by executing a *break* statement or bypass portions of the body of the construct by executing a *continue* statement as shown in Figure 2.21.

```
while (control expression)
{
    statement
    continue or break
    ::
    ::
    statement
}
```

Figure 2.21
Managing Control Flow in the while Construct

**Coding Style**

Control can be transferred out of a while loop using a *break* or *return* statement. Portions of the while body can be bypassed using a *continue* statement. This practice is strongly discouraged. We never want an asynchronous entry or exit from a block of code. We also want to ensure that we have only one entry and exit point to a block of code.

### 2.7.1.2 The for Loop

The for loop is also an example of an entry-condition loop and is considerably more general than the while loop. Its syntax is given in Figure 2.22.

```
Syntax
for (exp0; exp1; exp2)
{
    statements
}
```

Figure 2.22
The for Loop
The control expressions $exp_i$ perform the following.

- $exp_0$ is a comma separated list of initialization expressions.
  
The list of expressions is part of the for loop.
  
The list ends with a semi-colon.
  
  Initialization occurs once when the loop is entered. It is never repeated.

- $exp_1$ is the loop control expression.
  
  It is followed by a semi-colon.
  
  If the control expression does not evaluate to zero, execution of the loop continues.

  If it evaluates to zero, execution terminates.

  The control expression is tested on every iteration through the loop.

- $exp_2$ is used to update the loop control expression
  
  It will execute on each iteration of the loop.

The control expressions $exp_i$ are all optional. However, the semicolon separated places must still be present in the opening statement.

We can write

```
for (; ; )
```

but not

```
for ( )
```

We can use such a construct to create infinite loops which we’ll study shortly. These are essential in the design embedded applications. Unlike desktop applications, an embedded application is intended to run until stopped rather than after a single execution.

The execution of the for loop proceeds as follows.

1. Evaluate $exp_0$ if present
2. Evaluate $exp_1$ if present
   
   if zero
   
   terminate
   
   if not zero or not present
   
   continue execution
3. Execute the body of the for statement
4. Evaluate $exp_2$ if present
As we noted for the while loop, control can be transferred out using a `break` or `return` statement as illustrated in Figure 2.23. Once again, such a practice is strongly discouraged.

Let’s look at an example of a simple for loop.

**Example 2.2**

```c
/*
 * A simple for loop to print some numbers
 */
#include <stdio.h>
void main(void)
{
    int i = 0; // declare and initialize a loop variable
    for (i=0; i < 5; i++)
    {
        printf("The loop index is: %i\n",i);
    }
    return;
}
```

**Figure 2.24**

Working with the for loop Construct

### 2.7.3 Infinite Loops

In an embedded application, a program will often be structured as a sequence of initializing statements followed by an infinite loop running the intended tasks of the system. Such a loop can be constructed using either a `for` loop with no control expressions or a `for`
while loop with a control expression that always evaluates to true. These are written as is shown in Figure 2.25.

```
while (1)
{
    statement
}
```

Figure 2.25
Creating an Infinite Loop

So constructed, a loop can be used to control the execution of a series of tasks as the following code module illustrates.

**Example 2.3**

In Figure 2.26, we have a very simple task loop. The loop comprises four tasks,

- Prompt for input
- An input operation
- A computation operation
- An output operation

Each task gets as much time as it needs and always completes execution before relinquishing control. The system runs forever. When analyzing the code, several questions should come to mind.

- Why is the second getchar() operation performed?
- What is the following line doing?
  ```
  result += (value - '0');
  ```

When one enters data to a C program from the standard input or `standard in` (generally the keyboard) each character is stored in a temporary buffer until the user enters a new line by pressing the `Enter key` on the keyboard. The `getchar()` function only reads one character at a time, thus, the newline character is left in the input buffer. That is what is being read and discarded during the second input operation.

When data is entered into a program from the keyboard, each character is encoded in ASCII. If it is necessary to perform traditional arithmetic operations on the data, each character must first be converted to a decimal integer.

If one were to look at the ASCII table, one would find that the characters 0..9 are represented by the hex numbers 30..39 (or decimal numbers 48..57). Specifically, the character ‘0’ has the hex value 30 (or decimal 48). Thus, if the character ‘7’ (hex 37) is entered, subtracting ‘0’ (30) from ‘7’ (37) gives us the integer value 7. This technique only works on the ASCII characters 0..9.
2.7.2 The Exit-Condition Loops

An exit-condition loop tests the condition at the end of the loop, after the code inside the loop has already executed. Such loops guarantee that the code inside the loop will execute at least once whether or not the control expression is true. Exit-condition loops are delimited by the C keyword pair `do` and `while`. Such loops are commonly referred to as `do-while` loops or simply `do` loops.

2.7.2.1 The `do` - `while` Loop

The `do-while` loops, illustrated in Figure 2.27, are similar to basic while loops. Execution proceeds analogously.

```
/*
 * Flow of Control - A simple task loop.
 */
#include <stdio.h>
void main(void)
{
    // declare some working variables
    char value;
    int result = 2;

    // build an infinite loop
    while (1)
    {
        // Prompt for input
        printf("Enter a character: ");

        // Read the keyboard – an input operation
        value = getchar();

        // Do some calculations and display result
        getchar();
        result += (value - '0');
        printf("The calculations show: %i\n", result);
    }
}
```
The execution of the do-while loop proceeds as follows.

1. Evaluate each statement in the loop body.
2. Evaluate the control expression
3. If the control expression \( \neq \) zero
4. Repeat the process
5. Execution complete
   When the control expression evaluates to zero.

Once again, as seen in Figure 2.28, control can be transferred out of the loop using a break or return statement...and once again, such a practice is strongly discouraged. Observe that with the do – while construct, \textit{statement} is always executed at least once.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{do-while.png}
\caption{Managing Control Flow in the do-while loop Construct}
\end{figure}

\textbf{Example 2.4}

The code module in Figure 2.29 will execute five iterations of the loop and will print the message and the loop index on each iteration.

\begin{verbatim}
/*
 * A simple do-while loop to print some numbers
 */
#include <stdio.h>
void main(void)
{
    int i = 0;
    do
    {
        printf("The loop index is:%d\n",i);
        i++;
    } while (i < 5);
    return;
}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{do-while.png}
\caption{Managing Control Flow in the do-while loop Construct}
\end{figure}
2.8 Summary

In this tutorial we opened with a review the C language arithmetic and logical operators then followed with a brief study of the constructs by which we can alter and control the flow through a program. We learned that such constructs include relational expressions as well as those that control branching, jumping, and looping within an application. We also covered a variety of techniques and methodologies that can help one to learn and practice good coding style and to design and develop more robust programs.
References


