Chapter 5
Conditionals, Boolean Variables

5.1 Chapter Overview

In structured programming, there are three control structures that regulate the flow of execution: sequence (one statement after another), repetition (looping, which was the previous chapter), and selection (making a decision, which is this chapter). Conditional statements are used when making a selection, or decision. In these notes, we will cover `if`, `if-else`, and `if-else-if` statements, as well as `switch` statements. Also, Boolean variables will be explained.

5.2 `if` Statement

The `if` statement is used when we want the computer to choose between two alternative courses of action. If the condition is true, the code in the body of the `if` statement will execute; if it is false, the body of the `if` statement will be skipped and execution will continue with the code immediately after the `if` statement.

```
#include <stdio.h>

int main(void) {
    int age;

    if ((age >= 13) && (age <= 19)) {
        printf("You are a teenager\n");
    }

    . . .

    return 0;
}
```

The listing above shows an example of using an `if` statement. When we encounter an `if` statement, we first evaluate the condition. In this example, our condition is colored red in the code above. The condition is normally a binary expression that evaluates to either true or false. If the condition evaluates to true, the body of the `if` statement (in this case, the print statement) executes. If the condition evaluates to false then the body does not execute and execution continues from the statement following the `if` statement (in this case, whatever code may be in the place noted by the ellipsis above).

Note the use of brackets around the code in the body of the `if` statement. If the code in the body consists of only one line, then the brackets are not necessary. But if the code in the body consists of more than one line of code, then the brackets ***are necessary***. It is good practice to use the brackets all the time, at least as a beginning programmer, even if there is only one line of code, so that if you go back later and add another line of code, you won't have to remember to also add the brackets.

5.3 `if-else` Statement

The previous `if` statement included code that executes when a condition is true. If we want certain code to execute when a condition is true and other code that executes when a condition is false we use `if-else` statements.

```
#include <stdio.h>

int main(void) {
    int age;

    if ((age >= 13) && (age <= 19)) {
        printf("You are a teenager\n");
    }
    else {
        printf("You are not a teenager\n");
    }

    . . .

    return 0;
}
```

In the listing above, we show an example of using `if-else` statement. We start by evaluating the Boolean condition in the `if` statement (in red). If the condition evaluates to true, the body of the `if` part is executed. When the body of the `if` part is executed, the program resumes execution at the point indicated by the ellipsis, skipping the body of the `else` part. However, if the condition evaluates to false, the body of the `else` part is executed skipping the body of the `if` part. After executing the body of the `else` part, execution resumes at the point indicated by the ellipsis.

5.4 `if-else-if` Statement

Instead of nesting `if-else` statements, C provides us with the `if-else-if` construct.

```
#include <stdio.h>

int main(void) {
    int day;

    if ((day > 2) && (day <= 6)) {
        printf("Weekday\n");
    }
    else if (day == 7){
        printf("Saturday\n");
    }
    else {
        printf("Sunday\n");
    }

    . . .

    return 0;
}
```

The listing above provides an example of using `if-else-if` statements. First, the condition for the `if` part (in red) is evaluated. If it is true, the body for that part is executed, the rest is skipped, and execution resumes at the point indicated by the ellipsis. If the condition is false, the next condition in the `else if` part is evaluated. If this condition is true, then the body for that part is executed, the rest is skipped, and execution resumes at the point indicated by the ellipsis. Note that if this condition is true, the code in the `if` part does not execute. However, if this condition is also false, then the body of the `else` part executes, followed by the statements indicated by the ellipsis. In general only one condition will be true and its corresponding code will execute.

5.5 Nested `if` Statements

There are times where you may want/need to have "nested" `if` statements. A "nested" `if` statement is one that has `if` statements inside of `if` statements. Remember that an `else` belongs to the `if` part that immediately precedes it unless brackets are used to explicitly direct which `if` part it belongs to. The following example demonstrates the importance of the use of brackets.

The code below is an example of the ***dangling*** else problem. It will be treated as it is written in the box to the right.

```
if (gameIsOver == 0)
    if (playerToMove == YOU)
        printf ("Your Move \n");
else
    printf ("The game is over \n");
```

```
if (gameIsOver == 0)
    if (playerToMove == YOU)
        printf ("Your Move \n");
    else
        printf ("The game is over \n");
```

If it is desirable for the code to execute the way it is arranged in the first box above, then curly braces are required, as in the box below:

```
if (gameIsOver == 0) {
    if (playerToMove == YOU)
        printf ("Your Move \n");
}
else
    printf ("The game is over \n");
```

5.6 switch Statement

C provides another construct to use that is more convenient than nesting `if-else` statements. This construct is the `switch` statement.

```
#include <stdio.h>

int main (void) {
    float value1, value2;
    char operator;

    printf (“Type in your expression. \n”);
    scanf (“%f %c %f”, &value1, &operator, &value2);

    switch (operator) {
        case ‘+’:
            printf (“%.2f\n”, value1 + value2);
            break;
        case ‘-‘:
            printf (“%.2f\n”, value1 - value2);
            break;
        case ‘*’:
            printf (“%.2f\n”, value1 * value2);
            break;
        case ‘/’:
            if (value2 == 0)
                printf (“Division by zero.\n”);
            else
                printf (“%.2f\n”, value1 / value2);
            break;
        default:
            printf (“Unknown operator.\n”);
            break;
    }

    return 0;
}
```

The listing above provides an example of using `switch` statements. We first need to determine the variable which we will consider for evaluation. In this case, the variable is `operator`. The user is asked to enter an expression. We then examine the different cases (values) of `operator`. The first case is we check if `operator` is `+`. If it is, then the body of that case executes (the print statement will show the result of adding the two numbers). Because the `break;` statement is at the end of the case after the print statement, all the rest of the switch statement will be skipped and program execution will continue with the first line of code after the switch statement, which in this example, is the return statement.

If the value of `operator` was not `+` then it will check to see if the value is `–`. If it is, then the body of that case executes, which is a print statement that shows the difference of the two numbers. When the `break;` statement is reached, the rest of the switch statement is skipped and the program continues execution at the first line of code after the switch statement, which is the return statement.

If the value of `operator` was neither `+` nor `-` then it will check to see if the value is `*`. If it is, the body of that case executes and upon reaching the `break;` statement, execution will skip down to the return statement.

If the value of `operator` was none of `+` `-` or `*` then it will check to see if the value is `/`. If it is division, the case statement for that operator contains an `if` statement to check to see if the second value (which would be the denominator) is `0` before doing any dividing, because you cannot divide by zero.

After that case, notice the `default` case. A `default` case is not required, but many times it provides a nice fall-through exception like in the above program. If none of the cases above the `default` are evaluated to "true", then the `default` case will definitely be the case that executes.

Another thing to note about the `break` statement. If we forget to add the `break` statement, execution will fall through executing all the statements until the first encountered `break`. Therefore, make sure you add a `break` statement for each case you handle. You can even add it with the `default` case just for consistency, though it's not required because that's the end of the switch statement anyway.

The following example evaluates a character entered by the user to determine if it is a vowel or not. Notice how each of the cases corresponding to a lower case vowel has no body. The meaning of that is as if there is an "or" in between the two cases. In other words, it is saying "if the input is a lower case a **or** upper case A, then print the following statement". It is the same with each vowel. Since the print statement is the same for all 5 pairs of vowels, how could this program be modified?

```
#include <stdio.h>

int main (void) {

   char input;

   printf("\nEnter a character: ");
   scanf("%c", &input);

   switch (input) {
      case 'a':
      case 'A':
         printf("\n%c is a vowel\n\n", input);
         break;
      case 'e':
      case 'E':
         printf("\n%c is a vowel\n\n", input);
         break;
      case 'i':
      case 'I':
         printf("\n%c is a vowel\n\n", input);
         break;
      case 'o':
      case 'O':
         printf("\n%c is a vowel\n\n", input);
         break;
      case 'u':
      case 'U':
         printf("\n%c is a vowel\n\n", input);
         break;
      default:
         printf("\n%c is not a vowel\n\n", input);

   }
}
```

The following program also combines cases. If the month is any of those listed first (1, 3, 5, 7, 8, 10 **or** 12), then it has 31 days. If the month is not any of those, but is 4, 6, 9 **or** 11, then it has 30 days. If it is none of any of those numbers, then the default picks up the case that it must be February and prints an appropriate message.

```
#include <stdio.h>

int main (void) {

    int month;

    . . .

    switch (month) {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12: printf("The number of days is 31\n");
            break;
        case 4:
        case 6:
        case 9:
        case 11: printf("The number of days is 30\n");
            break;
        default: printf("The number of days is either 28 or 29\n");
            break;
    }
}
```

5.7 Boolean Variables

A variable that assumes only one of two different values, such as a 1 (indicating TRUE) or a 0 (indicating FALSE), is called a ***flag***, and is probably where you would want to use a Boolean type of variable.

As we discussed in chapter 3, you can declare a Boolean variable using the basic C data type `_Bool`. The value of that variable would be either 1 or 0. Or you can include the `<stdbool.h>` at the top of your program and then you can declare variables to be of type `bool`. If you declare a variable to be of type `bool`, then the value of that variable can be either `true` or `false`.

These side by side programs is Program 5.10 from the book, pages 86-88. The one on the left shows an example of using the basic C date type `_Bool`. The one on the right shows the same program using the type `bool` with the inclusion of the `<stdbool.h>` library file at the top.

```
#include <stdio.h>


int main (void) {
    int p, d;
    _Bool isPrime;

    for (p = 2; p <= 50: ++p) {
        isPrime = 1;

        for (d = 2; d < p; ++d)
            if (p % d == 0)
                isPrime = 0;

        if (isPrime != 0)
            printf ("%i  ", p);
    }

    printf ("\n");

    return 0;
}
```

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

int main (void) {
    int p, d;
    bool isPrime;

    for (p = 2; p <= 50: ++p) {
        isPrime = true;

        for (d = 2; d < p; ++d)
            if (p % d == 0)
                isPrime = false;

        if (isPrime != false)
            printf ("%i  ", p);
    }

    printf ("\n");

    return 0;
}
```

5.8 Conditional Operators

Unlike other operators in C, which are either unary or binary operators, conditional operators are ***ternary*** – takes three operands.

```
condition  ?  expression1  :  expression2
```

The `condition` is an expression, usually a relational expression, and is evaluated first. If it evaluates to TRUE, then `expression1` takes place. If the `condition` evaluates to FALSE, then `expression2` takes place.

```
s = (x  <  0)  ?  -1  :  x * x;
```

In the above example, if x is < 0, then -1 would be assigned to s. If x is not < 0, then the value of x^2 would be assigned to s.