

The Public Domain licencing notice is removed on inspection copies.

ShockSoc C Programming Support Sessions

Oliver Dixon
ShockSoc Technical Officer
od641@york.ac.uk

P/T/401 2pm–5pm
Spring Term, 2021–2022

Lab Script 04: Conditionals and Iteration (Part 1)

INTERNAL SHOCKSOC INSPECTION COPY

Please do not disseminate. Direct any comments to Oliver Dixon.

Table of Contents

1	Conditionals and Iteration: A Synopsis	2
2	Conditionals	2
2.1	Forming Conditionals: The <code>if</code> Construct	2
2.1.1	Extended Footnote: Omission of Braces	3
2.2	Useful Operators	4
2.3	The In-Line Ternary & Comma Operators	5
2.4	The <code>switch-case</code> Construct	6
2.4.1	A Practical Example: A Four-Operation Digit Calculator	8
2.4.2	A Practical Example: Duff's Device (Fall-Through) . . .	9
2.5	Exercises: Forming Conditionals	9
3	Next Week . . .	11

Due to formatting and typesetting restrictions, some listings in this document have been pushed to the final pages, irrespective of their context. For readers on a computer, this is not an issue, as all intra-document references are rendered as PDF hyperlinks, and virtual indexes are compiled correctly, however I do offer my heartfelt apologies to all readers enjoying a physical medium.

1 Conditionals and Iteration: A Synopsis

One of many cornerstones of any modern language lies in its ability to describe decision and iterative constructs: that is, executing certain code providing an arbitrary condition holds, and repeatedly executing blocks of code while some similarly arbitrary condition holds. As many would expect, the free-form syntax of C lends itself extremely well to powerful and versatile conditional and iterative programming constructs, allowing remarkably concise (albeit readable) code to perform a network of otherwise-complex tasks. First, we shall examine basic conditional statements, how to construct them, and work through a few increasingly dense situations in which a working knowledge of C syntax will prove invaluable. In the second section of the script, we shall move onto the various flexible iterative structures, which use conditionals extensively to determine to point at which an iteration should halt.

2 Conditionals

2.1 Forming Conditionals: The `if` Construct

The `if` keyword denotes a sequence point. Its syntax is very predictable: the condition is placed in parentheses to the right-hand side of the `if`, followed by an

```
1 #include <stdio.h>
2
3 int main ( int argc, char ** argv )
4 {
5     if ( argc > 1 )
6         puts ( "Some arguments were passed." );
7     else
8         puts ( "No arguments were passed." );
9
10    return 0;
11 }
```

Listing 1: Sequence Point, Example 1. Notice the `argc` value stores the number of arguments passed to the binary. On virtually every modern shell, the first “argument” will be the name of the executable, in which we are uninterested; testing for `argc > 0` would always return true.

(optional) set of braces. If the condition evaluates to something other than zero, the code within the braces is executed; if the condition evaluates to zero, the code surrounded by the braces is skipped. C also supports the `else` and `else if` keywords, allowing a chain of sequence points to be defined. If used, the lone `else` keyword must always appear at the end of an `if-else` chain, as it will only be executed if no previous expression in the chain has evaluated as true. Before going any further, let’s examine some examples: Listing 1, Listing 2¹ (once we introduce the tenary operator and `switch-case` statement, we will be able to write vastly condensed versions of these “algorithms”).

2.1.1 Extended Footnote: Omission of Braces

In the common case of an `if` statement executing only one statement, then the braces surrounding the statement may be omitted, although correct indentation must still be used. Although this is a very common practice, and one preferred by the PSS author, some oft-embedded programmers will vehemently insist upon the use of braces in all cases, as careless omission can result in very obscure bugs. For example, consider the code listed in Listing 8. As the preprocessor will execute before the compiler proper, the `#define` will be expanded into multiple statements, and the call appearing as a normal function call may be explicitly expanded into multiple. Although this minimal example may seem obvious, many symbols appearing as functions in larger libraries tend to be implemented as preprocessor macros, and programmers are often unaware of their real nature. All modern compilers, with the correct arguments, will raise warnings or errors in such cases of unguarded constructs; see Listing 9 for an example of a GCC output script complaining about such a case. Although written in the context

¹In the second example, the braces around the outer sequence point could technically be removed, since the inner chain is a single symbol block, however many compilers would warn that this may lead to dangerous levels of ambiguity. Compiler designers will often informally refer to this as a “dangling else”.

```
1 #include <stdio.h>
2
3 int main ( int argc, char ** argv )
4 {
5     if ( argc > 1 ) {
6         if ( * ( argv [ 1 ] ) == '+' )
7             puts ( "Adding..." );
8         else if ( * ( argv [ 1 ] ) == '-' )
9             puts ( "Subtracting..." );
10        else if ( * ( argv [ 1 ] ) == '*' )
11            puts ( "Multiplying..." );
12        else if ( * ( argv [ 1 ] ) == '/' )
13            puts ( "Dividing..." );
14        else
15            fputs ( "Unknown operation!\n", stderr );
16    } else
17        fputs ( "No argument passed.\n", stderr );
18
19    return 0;
20 }
```

Listing 2: Sequence Points, Example 2. As with many other scope types in C, `if` statements can be nested. With `else if/else` chains, once one condition evaluates to true, the corresponding code block is run and the execution jumps to the end of the *entire* chain.

of `if` statements, all of the above also applies for `else`, `else if`, `for`, `while`, and `do` arrangements. We have seen two of these arrangements so far; the latter three concern iteration, and will be explored shortly.

2.2 Useful Operators

In the previous examples, we have already encountered the `>` (greater-than) and `==` (equivalence test) operators. For the sake of completeness, see Table 1 for a complete list of similarly useful operators. In every case, employing one of these operators will reduce the expression to `0` or `1`. As mentioned, sequence point constructs will consider `0` to be false, and non-zero (`!0`) to be true. Table 2 enumerates the commutative and non-commutative operators, and demonstrates cases in which multiple compound operators may be simplified into a single expression. Note that these operators needn't be contained within a well-defined sequence point construct, such as an `if` statement. Demonstrating the power of C syntax, the use of these operators will reduce the entire statement to a true or false integer value. For example, the peculiar-looking initialisation statement `int a = (x > y);` would set `a` to zero if `x` is strictly greater than `y`, or one otherwise. Although the parentheses are not strictly necessary, it is wise to enclose any expression involving boolean or order relations within an explicit parenthetical structure, as that local structure will reduce to its aforementioned integer representation.

Operator	Symbol	Example	Class
Equivalence	<code>==</code>	<code>x == y</code>	Equality Test
Negated Equivalence	<code>!=</code>	<code>y != z</code>	Equality Test
Unary Negation	<code>!</code>	<code>!x</code>	Boolean Logic
Conjunction	<code>&&</code>	<code>y && z</code>	Boolean Logic
Disjunction	<code> </code>	<code>x z</code>	Boolean Logic
Greater-Than	<code>></code>	<code>a > b</code>	Order Relation
Less-Than	<code><</code>	<code>b < a</code>	Order Relation
<i>G.T.</i> -or-equal-to	<code>>=</code>	<code>b >= a</code>	Order Relation
<i>L.T.</i> -or-equal-to	<code><=</code>	<code>a <= b</code>	Order Relation

Table 1: Some operators capable of reducing an expression to ‘0’ or ‘1’.

St. Operator(s)	Statement	Implication	Im. Operator(s)
Equivalence	<code>x == y</code>	<code>y == x</code>	Equivalence
Negated Equivalence	<code>x != y</code>	<code>y != x</code>	Negated Equivalence
Unary Negation	<code>!x</code>	<code>!x</code>	Unary Negation
Conjunction	<code>x && y</code>	<code>y && x</code>	Conjunction
Disjunction	<code>x y</code>	<code>y x</code>	Disjunction
Greater-Than	<code>x > y</code>	<code>!(x <= y)</code>	Neg. and L.E.Q.
Less-Than	<code>x < y</code>	<code>!(x >= y)</code>	Neg. and G.E.Q.
G.E.Q.	<code>x >= y</code>	<code>!(x < y)</code>	Neg. and L.T.
L.E.Q.	<code>x <= y</code>	<code>!(x > y)</code>	Neg. and G.T.

Table 2: Many operators are commutative, but some may be simplified.

2.3 The In-Line Ternary & Comma Operators

It is occasionally useful to perform the equivalent of an `if-else` statement when doing something in-line, such as for a variable assignment or within a function call. Unlike many languages (such as Python), C provides a convenient syntax for such cases, in the form of the *ternary* operator. The general syntax of the ternary operator is “(condition) ? (true-case) : (false-case)”, where **condition** is some expression to be tested for non-zero, and the **true-case** and **false-case** blocks contain statements to be executed accordingly for either case. This helpful operator can often be used to drastically reduce the size of various comparisons, whilst being functionally equivalent and equally readable.

For a quick example, consider Listing 1. Since we are calling the same function in each case, with the only differing element being the string literal, this is a perfect candidate for the ternary operator—see Listing 3. Although very useful in some cases, they should be used with care, as their nature as “syntactical sugar” can occasionally create difficult-to-read statements, especially when the condition is complex. As shown in the example, breaking lines at sensible points and adding padding requires little effort and creates readable expressions.

```
1 #include <stdio.h>
2
3 int main ( int argc, char ** argv )
4 {
5     puts ( ( argc > 1 ) ? "Some arguments were passed." :
6             "No arguments were passed." );
7
8     return 0;
9 }
```

Listing 3: The ternary operator can be used to vastly simplify typical single-statement `if-else` blocks.

```
1 /* ... */
2
3 if ( !success )
4     return ( errno = EINVAL, -1 );
5
6 /* ... */
7
8 if ( !success ) {
9     errno = EINVAL;
10    return -1;
11 }
12
13 /* ... */
```

Listing 4: Using the comma operator, these blocks are practically identical, where `stmt-1` is `'errno = EINVAL'` and `stmt-2` is `'-1'`.

There is also the seldom-used (yet superbly named) comma operator, denoted by the `','` token, and can be used to similarly reduce expression sets of specific types. The C syntax standard defines the comma syntax as such: `"(stmt-1), (stmt-2)"`, where `stmt-1` and `stmt-2` are evaluated, and the result of `stmt-2` is returned. This has many uses, one of which is shown in Listing 4.

2.4 The switch-case Construct

Finally, we have the `switch-case` construct; both of these words are C keywords, although a `case` must appear within a parent `switch`. This construct is particularly useful for cases in which an integer-like scalar variable consists of many potential values, all of which should invoke subtly different behaviour. A perfect candidate for a `switch-case` statement was (intentionally) shown with an equivalent `if-else` chain in Listing 2. By reviewing Listing 5, try and infer the general syntax of the `switch-case` construct.

The syntax itself proves to be fairly simple and intuitive, with the parentheses following the `switch` containing the variable under inspection; this variable

```

1 #include <stdio.h>
2
3 int main ( int argc, char ** argv )
4 {
5     if ( argc < 2 ) {
6         fputs ( "No argument passed.\n", stderr );
7         return 1;
8     }
9
10    switch ( * ( argv [ 1 ] ) ) {
11        case '+': puts ( "Adding..." ); break;
12        case '-': puts ( "Subtracting..." ); break;
13        case '*': puts ( "Multiplying..." ); break;
14        case '/': puts ( "Dividing..." ); break;
15
16        default: fputs ( "Unknown operation!\n", stderr );
17    }
18
19    return 0;
20 }

```

Listing 5: An algorithmically identical implementation of Listing 2, using a **switch-case** block as opposed to an obvious **if-else** chain. This particular example alters the “**argc** > 1” check to its logical inverse (which is valid for all integers), jumping out early to avoid nesting the entire **switch** block.

will be implicitly cast to an integer. For each value to handle, a **case** statement is inserted, followed by the value for which any code, following the colon, but before the **break**, should be executed². Generally, **case** statements should be indented one more than their parent **switch**, but some coding guidelines, such as the Linux Kernel standard, advise against this practice³. Finally, much like the terminating **else** keyword in an **if-else** chain, the **default** keyword can be used as a pseudo-label to denote code that should execute if none of the well-defined **case** conditions were matched. The **default** case *must* appear as the final case, and thus, does not require a **break** statement.

A further interesting point regarding the **switch** structure concerns the native code generation. Whilst many modern compilers will be wildly unpre-

²Although rather common, the **break** statement is not strictly necessary, and can be used to create some very interesting algorithms, two of which will be inspected shortly. Unfortunately, many modern languages such as C#, have slightly less freely formed syntax, and do not support fall-through cases; a .NET compiler will refuse to compile cases without a corresponding **break** statement. Whilst enforcing against fall-through cases renders JIT compilation of **switch** statements significantly easier, an entire class of clever algorithms is excluded from implementation.

³As lines of code should be limited to seventy-two (ANSI) or eighty characters, the Linux Kernel standard claims that indenting **case** labels with another eight spaces is wasteful and makes code harder to format. Conversely, if you are more than three levels of indentation deep anyway, your code is probably too complex and should be split into modular functions. This is a great point of contention, and one which should be decided by the individual author or team. All examples in PSS scripts will indent such labels.

```

1 #include <errno.h>
2
3 static int op_add ( int a, int b ) { return a + b; }
4 static int op_sub ( int a, int b ) { return a - b; }
5 static int op_mul ( int a, int b ) { return a * b; }
6 static int op_div ( int a, int b ) { return a / b; }
7
8 int calc ( char op, int a, int b )
9 {
10     int ( * op_func ) ( int, int );
11
12     switch ( op ) {
13         case '+': op_func = op_add; break;
14         case '-': op_func = op_sub; break;
15         case '*': op_func = op_mul; break;
16         case '/': op_func = op_div; break;
17
18         default : return ( errno = EOPNOTSUPP, 0 );
19     }
20
21     return op_func ( a, b );
22 }

```

Listing 6: A function, `calc`, to take two integer operands and a binary arithmetic operation, and return the result of the operation applied to the operands.

dictable in this regard, it is useful to know the basic concept. Long `if-else` chains are problematic due to their reliance on branching: for a typical `if` statement, a traditional processor must perform the condition check, storing the result in the accumulator or status register, and then jump to a certain instruction address depending on the value in the accumulator. In contrast, with a `switch` statement, a compiler can generate a *jump table* and obviate the initial condition check entirely. On older CPU architectures that tend to branch relatively slowly, the generation of an immediate jump table will often prove an invaluable optimisation for iteratively invoked subroutines; for an arbitrary number of cases, this method of selection runs in $\mathcal{O}(1)$ constant time. For more complex situations, optimisers may generate a binary search for case-selection, in which case the time complexity becomes logarithmic⁴: $\mathcal{O}(\log_2(n))$.

2.4.1 A Practical Example: A Four-Operation Digit Calculator

Listing 6 demonstrates many of the concepts introduced today: the `switch` statement, the concept of a `default` case, and the comma operator.

⁴Do not worry if you are as-yet unfamiliar with the Big- \mathcal{O} notation. It is simply a method of measuring an execution parameter, often time, of an algorithm as a function of its input size n . In this instance, n denotes the number of defined cases, and \mathcal{O} denotes a measure of runtime selection performance. Constant time is conferred by $\mathcal{O}(1)$, that is to say execution time is independent of input; this is obviously the desirable case, but is seldom possible for most algorithms. For the trivial `if-else` chain, the time complexity of selection will likely be linear $\mathcal{O}(n)$.

```
1 void duff_send ( short * to, short * from, int count )
2 {
3     int n = ( count + 7 ) / 8;
4
5     switch ( count % 8 ) {
6         case 0: do { *to = *from++;
7                     case 7:      *to = *from++;
8                     case 6:      *to = *from++;
9                     case 5:      *to = *from++;
10                    case 4:      *to = *from++;
11                    case 3:      *to = *from++;
12                    case 2:      *to = *from++;
13                    case 1:      *to = *from++;
14                } while ( --n > 0 );
15    }
16
17    /* WHAT'S GOING ON?! */
18 }
```

Listing 7: The classical Duff's device implemented in C99. A modern compiler might complain about the lack of a `default` case, but in this instance of applying the modulo-eight operator, we can be fairly confident that we've enumerated all possibilities!

2.4.2 A Practical Example: Duff's Device (Fall-Through)

Listing 7 shows a C99 implementation of a very classical and famous construct in C programming, designed to copy bytes from a source to a sink very quickly; it first appeared under Tom Duff while working at Lucasfilm. Advanced readers may wish to examine this example, and try and derive how the free-form syntax of C allows such an odd-looking arrangement of fall-through case statements interleaved with a `do...while` loop. (The various loop constructs will be introduced next week in the second half of this script.)

2.5 Exercises: Forming Conditionals

If you have managed to get to this stage, you should reward yourself with some exercises. Complete the following tasks, in order, and present your solutions to the PSS Author for scrutiny. The Author will be pleased to provide more exercises if you finish all of these.

1. Without using a nested arrangement, devise an `if` statement that evaluates to true if, and only if, `a` and `b` are non-zero, or `c` is greater than the sum of `a` and `b`.
2. Investigate assertions in C. The `assert.h` header file, and the man pages of the function signatures contained within, may be helpful. In what situations may an `assert(3)` call be useful? Why do you think it is disabled in many build circumstances?

```

1 #include <stdio.h>
2
3 #define PRINT_ERROR(msg) \
4     puts ( "Logger: system error. See stderr." ); \
5     fputs ( msg, stderr ); \
6     fputc ( '\n', stderr )
7
8 void report_status ( const char * msg, int iserror )
9 {
10     /* ... */
11
12     if ( iserror )
13         PRINT_ERROR ( msg );
14
15     /* ... */
16 }

```

Listing 8: A common mistake involving an unguarded if statement.

```

1 brace-omission-bug.c: In function 'report_status':
2 brace-omission-bug.c:4:9: warning: macro expands to multiple
3     statements [-Wmultistatement-macros]
4     4 |         puts ( "Logger: system error. See stderr." ); \
5       |         ~~~~
6 brace-omission-bug.c:13:17: note: in expansion of macro
7     'PRINT_ERROR'
8     13 |         PRINT_ERROR ( msg );
9        |         ~~~~~~
10 brace-omission-bug.c:12:9: note: some parts of macro expansion are
11     not guarded by this 'if' clause
12     12 |         if ( iserror )
13        |         ~

```

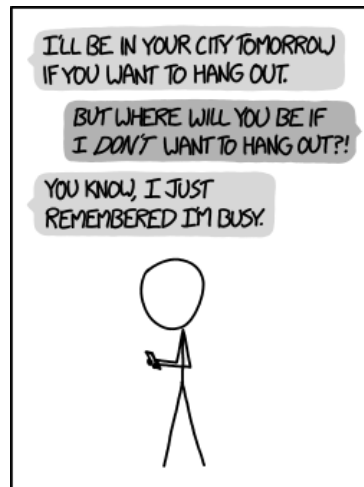
Listing 9: The output of `gcc -Wall` when trying to compile Listing 8. In the GCC and Clang/LLVM compilers, such errors are categorised under the `-Wmultistatement-macros` warning class.

3. What is wrong with the following conditional? “`a < b || c && d`”. How might it be fixed? You may want to investigate the precedence of binary operators in C, and the manners through which such ambiguous arrangements can be converted to unequivocal expressions.
4. Investigate the Duff’s device closer. Why does this work, and what assumptions does it make? What is loop-unrolling, and how does this common optimisation technique link to the previous discussion regarding slow branching and jump tables?
5. To force a familiarity with ternary operators, try constructing some nested ternary expressions. These operators are particularly useful in `#define` preprocessor macros, as they lend nicely to each other’s succinct nature.

3 Next Week...

Next week, we will continue with the theme of “Conditionals and Iteration”, shifting our focus to the various iterative structures of the C programming language. Once iteration has been covered, there is little left to cover: most of the basic concepts have been introduced, and only a few C-specific constructs such as `structs`, `unions`, and `enums` remain pertinent. As with any concept, practice is integral, and “good” programmers will have invested tens of thousands of hours to reach their level. **Please do not become discouraged if a certain concept does not immediately seem familiar or within easy reach.**

See you then, I hope you enjoyed the third PSS script.
PSS Author, Oliver Dixon



WHY I TRY NOT TO BE
PEDANTIC ABOUT CONDITIONALS.

XKCD #1652: “Conditionals”.
<https://xkcd.com/1652/>

PSS script dedicated to *M. Q.*