MSC20-C. Do not use a switch statement to transfer control into a complex block

A `switch` statement can be mixed with a block of code by starting the block in one case label, then having another case label within the block. The block can be pictured as spanning more than one case statement.


> If a switch statement has an associated case or default label within the scope of an identifier with a variably modified type, the entire switch statement shall be within the scope of that identifier.\(^{154}\)

Footnote 154 says:

> That is, the declaration either precedes the switch statement, or it follows the last case or default label associated with the switch that is in the block containing the declaration.

Note that the standard does not disallow jumping via `goto` or `switch` into loops that do not involve variably modified type identifiers. Consequently, loops and other blocks can be freely intermixed with `switch` statements. Unfortunately, such intermixing creates code that is, at best, confusing and unclear in what it does, which can cause undesirable behavior.

The examples here fall under the exception MSC17-C-EX2 in MSC17-C. Finish every set of statements associated with a case label with a break statement.

Noncompliant Code Example

This example shows the use of the `switch` statement to jump into a `for` loop:

```c
int f(int i) {  
    int j=0;  
    switch (i) {  
        case 1:  
            for(j=0;j<10;j++) {  
                /* No break; process case 2 as well */  
                case 2: /* switch jumps inside the for block */  
                    j++;  
                    /* No break; process case 3 as well */  
                    case 3:  
                        j++;  
                        }  
                break;  
        default:  
            /* Default action */  
            break;  
        }  
    return j;  
}
```

Implementation Details

When \(i = 1\), the entire `for` loop is executed. When \(i = 2\), two increments to \(j\) are made before the loop starts. When \(i = 3\), one increment to \(j\) is made before the loop starts. The default case is no loop. Consequently, the function has the following behavior:

<table>
<thead>
<tr>
<th>(i)</th>
<th>(f(i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>Other values</td>
<td>0</td>
</tr>
</tbody>
</table>

Compliant Solution

The compliant solution separates the `switch` and `for` blocks:
int f(int i) {
    int j=0;
    switch (i) {
        case 1:
            /* No break; process case 2 as well */
            break;
        case 2:
            j++;
            /* No break; process case 3 as well */
            break;
        case 3:
            j++;
            break;
        default:
            /* Default action */
            return j;
    }
    for(j++;j<10;j++) {
        j+=2;
    }
    return j;
}

Noncompliant Code Example (Duff's Device)

Duff's device is a curious optimization applied to code intended to perform a serial copy. That is, it copies a series of bytes into one memory output in turn. A simple code to do this would be as follows:

    size_t count; /* Must be nonzero */
    char *to;     /* Output destination */
    char *from;   /* Points to count bytes to copy */
    do {
        *to = *from++;     /* Note that the "to" pointer */
        /* is NOT incremented. */
    } while (--count > 0);

However, this code might be unacceptably slow because the while condition is performed count times. The classic code for Duff's device unrolls this loop to minimize the number of comparisons performed:

    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++; }
        /* while (--count > 0); */
        case 7:  *to = *from++;
        case 6:  *to = *from++;
        case 5:  *to = *from++;
        case 4:  *to = *from++;
        case 3:  *to = *from++;
        case 2:  *to = *from++;
        case 1:  *to = *from++;
        } while (--n > 0);

In this code, the first iteration of the loop is subject to the switch statement, so it performs count % 8 assignments. Each subsequent iteration of the loop performs 8 assignments. (Being outside the loop, the switch statement is ignored.) Consequently, this code performs count assignments, but only n comparisons, so it is usually faster.

The code is widely considered to be valid C and C++ and is supported by all compliant compilers. When describing Duff's device, the creator [Duff 1988] noted,

    Many people . . . have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against.
Compliant Solution (Duff's Device)

This is an alternative implementation of Duff's device, which separates the switch statement and loop:

```c
int n = (count + 7) / 8;
switch (count % 8) {
  case 0: *to = *from++; /* Fall through */
  case 7: *to = *from++; /* Fall through */
  case 6: *to = *from++; /* Fall through */
  case 5: *to = *from++; /* Fall through */
  case 4: *to = *from++; /* Fall through */
  case 3: *to = *from++; /* Fall through */
  case 2: *to = *from++; /* Fall through */
  case 1: *to = *from++; /* Fall through */
}
while (--n > 0) {
  *to = *from++;
  *to = *from++;
  *to = *from++;
  *to = *from++;
  *to = *from++;
  *to = *from++;
  *to = *from++;
  *to = *from++;
}
```

Risk Assessment

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSC20-C</td>
<td>Medium</td>
<td>Probable</td>
<td>Medium</td>
<td>P8</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrée</td>
<td>19.04</td>
<td>switch-label</td>
<td>Fully checked</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>LANG.STRUCT.SW. MPC</td>
<td>Misplaced case</td>
</tr>
<tr>
<td>ECLAIR</td>
<td>1.2</td>
<td>CC2.MSC20</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>245 S</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rec. MSC20-C</td>
<td>Checks for situations where switch label is not at the outermost level of switch statement body (rec. fully covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>2019</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>19.04</td>
<td>switch-label</td>
<td>Fully checked</td>
</tr>
<tr>
<td>SonarQube C/C++ Plugin</td>
<td>3.11</td>
<td>S1036</td>
<td></td>
</tr>
</tbody>
</table>

Related Guidelines

<table>
<thead>
<tr>
<th>Standard</th>
<th>Guideline</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEI CERT C++ Coding Standard</td>
<td>VOID MSC20-CPP. Do not use a switch statement to transfer control into a complex block</td>
</tr>
<tr>
<td>ISO/IEC TR 24731-1:2007</td>
<td></td>
</tr>
<tr>
<td>MISRA C:2012</td>
<td>Rule 16.2 (required)</td>
</tr>
</tbody>
</table>

Bibliography

[Duff 1988]            Tom Duff on Duff's Device