The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

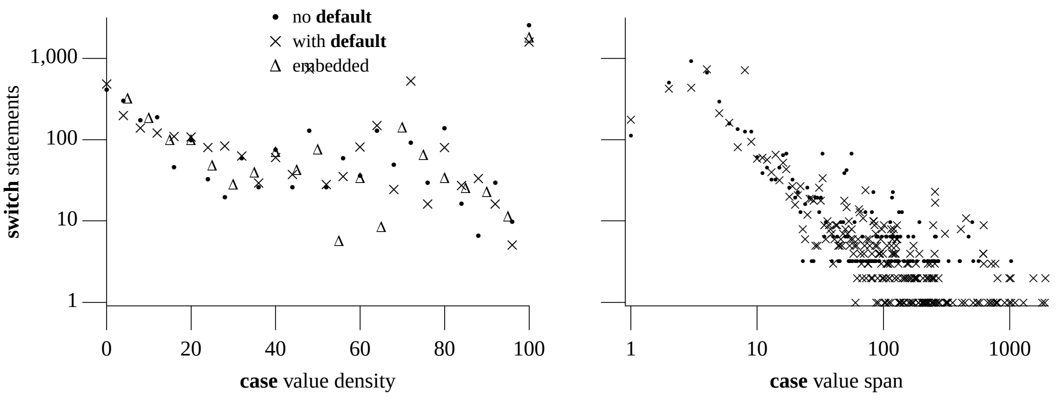

Figure 1748.1: *Density* of **case** label values (calculated as (maximum **case** label value minus minimum **case** label value minus one) divided by the number of **case** labels associated with a **switch** statement) and span of **case** label values (calculated as (maximum **case** label value minus minimum **case** label value minus one)). Based on the translated form of this book's benchmark programs and embedded results from Engblom[2] (which were scaled, i.e., multiplied by a constant, to allow comparison). The *no* ***default*** results were scaled so that the total count of **switch** statements matched those that included a **default** label.

6.8.4.2 The `switch` statement

Constraints

switch statement

The controlling expression of a **switch** statement shall have integer type. 1748

Commentary

A **switch** statement uses the exact value of its controlling expression and it is not possible to guarantee the exact value of an expression having a floating type (there is a degree of unpredictability in the value between different implementations). For this reason implementations are not required to support controlling expressions having a floating type.

C++

6.4.2p2 *The condition shall be of integral type, enumeration type, or of a class type for which a single conversion function to integral or enumeration type exists (12.3).*

If only constructs that are available in C are used the set of possible expressions is the same.

Common Implementations

The base document did not support the types **long** and **unsigned long**. Support for integer types with rank greater than **int** was added during the early evolution of C.[4]

Other Languages

There are some relatively modern languages (e.g., Perl) that do not support a **switch** statement. Java does not support controlling expressions having type **long**. Some languages (e.g., PHP) support controlling expressions having a string type.

Coding Guidelines

A controlling expression, in a **switch** statement, having a boolean role might be thought to be unusual, an **if** statement being considered more appropriate. However, the designer may be expecting the type of the controlling expression to evolve to a non-boolean role, or the **switch** statement may have once contained more **case** labels.

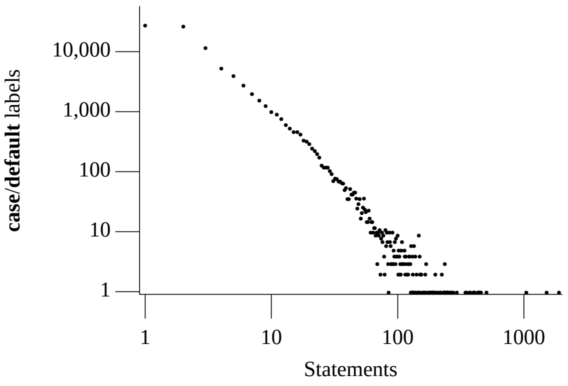

Figure 1748.2: Number of **case/default** labels having s given number of statements following them (statements from any nested **switch** statements did not contribute towards the count of a label). Based on the visible form of the `.c` files.

Table 1748.1: Occurrence of **switch** statements having a controlling expression of the given type (as a percentage of all **switch** statements). Based on the translated form of this book's benchmark programs.

Type	%	Type	%
int	29.5	bit-field	3.1
unsigned long	18.7	**unsigned short**	2.8
enum	14.6	**short**	2.5
unsigned char	12.4	**long**	0.9
unsigned int	10.0	other-types	0.2
char	5.1		

1749 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.133)

switch past variably modified type

Commentary

The declaration of an identifier having variable modified type can occur in one of the sequence of statements labeled by a **case** or **default**, provided it appears within a compound statement that does not contain any other **case** or **default** labels associated with that **switch** statement, or it appear after the last **case** or **default** label in the **switch** statement. In the compound statement case the variably modified type will not be within the scope of any **case** or **default** labels (its lifetime terminates at the end of the compound statement).

The wording of the requirement is overly strict in that it prohibits uses that might be considered well behaved. For instance:

```
switch (i)
   {
   case 1:
         int x[n];
         /* ... */
         break;

   case 2:
         /* Statements that don't access x. */
   }
```

Attempting to create wording to support such edge cases was considered to be a risk (various ambiguities may later be found in it) that was not worth the benefit. Additional rationale for this requirement is discussed elsewhere.

goto past variably modified type

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

The C++ Standard contains the additional requirement that (the wording in a subsequent example suggests that being visible rather than *in scope* is more accurate terminology):

6.7p3 *It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps[77)] from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has POD type (3.9) and is declared without an* `initializer` *(8.5).*

```
void f(void)
{
switch (2)
   {
   int loc = 99; /* strictly conforming */
                 // ill-formed

   case 2: return;
   }
}
```

Example

```
extern int glob;

void f(int p_loc)
{
switch (p_loc) /* This part of the switch statement is not within the scope of a_1. */
   {
   case 1: ;
           int a_1[glob]; /* This declaration causes a constraint violation. */

   case 2: a_1[2] = 4;
           break;

   case 3: {
           long a_2[glob]; /* Conforming: no case label within the scope of a_2. */
           /* ... */
           }
           break;
   }
}
```

case label unique in same switch

The expression of each **case** label shall be an integer constant expression and no two of the **case** constant expressions in the same **switch** statement shall have the same value after conversion. 1750

Commentary

Two case labels having the same value is effectively equivalent to declaring two labels, within the same function, having the same name.

Coding Guidelines

Some sequences of case label values might be considered to contain suspicious entries or omissions. For instance, a single value that is significantly larger or smaller than the other values (an island), or a value missing from the middle of a contiguous sequence of values (a hole). While some static analysis tools check for such *suspicious values*, it is not clear to your author what, if any, guideline recommendation would be worthwhile.

1751 There may be at most one **default** label in a **switch** statement.

default label at most one

Commentary

A **default** label is the destination of a jump for some, possible empty, set of values of the controlling expression. As such it is required to be unique (if it occurs) within a **switch** statement.

A bug in the terminology being used in the standard "may" ⇒ "shall".

Coding Guidelines

Some coding guideline documents recommend that all **switch** statements contain a **default** label. There does not appear to be an obvious benefit (as defined by these coding guideline subsections, although there may be benefits for other reasons) for such a guideline recommendation. To adhere to the guideline developers simply need to supply a **default** label and an associated null statement. There are a number of situations where adhering to such a guideline recommendation leads to the creation of redundant code (e.g., if all possible values are covered by the **case** labels, either because they handle all values that the controlling expression can take or because execution of the **switch** statement is conditional on an **if** statement that guarantees the controlling expression is within a known range).

redundant code

Usage

In the visible form of the `.c` files, 72.8% of **switch** statements contain a **default** label.

1752 (Any enclosed **switch** statement may have a **default** label or **case** constant expressions with values that duplicate **case** constant expressions in the enclosing **switch** statement.)

Commentary

This specification (semantics in a Constraints clause) clarifies the interpretation to be given to the phrase "in the same **switch** statement" appearing earlier in this Constraints clause.

1750 case label unique in same switch

Semantics

1753 A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** label and the values of any **case** labels on or in the switch body.

switch statement causes jump

Commentary

This defines the term *switch body*. Developers also use the terminology *body of the switch*.

It is possible to write a **switch** statement as an equivalent sequence of **if** statements. However, experience shows that in some cases the **switch** statement appears to require less significantly less (cognitive) effort to comprehend than a sequence of **if** statements.

Common Implementations

Many processors include some form of instruction (often called an *indirect jump*) that indexes into a table (commonly known as a *jump table*) to obtain a location to jump to. The extent to which it is considered to be more efficient to use such an instruction, rather than a series of **if** statements varies between processors (whose behavior varies for the case where the index is out of range of the jump table) and implementations (the sophistication of the available optimizer). The presence of a **default** label creates additional complications in that all values of the controlling expression, not covered by a **case** label, need to be explicitly handled. Spuler[5] discusses the general issues.

Some translators implement **switch** statements as a series of **if** statements. Knowledgeable developers know that, in such implementations, placing the most frequently executed **case** labels before the less frequently executed ones can provide a worthwhile performance improvement. Some translators[1, 3] provide an option that allows the developer to specify whether a jump table or sequence of **if** statements should to be used.

Optimal execution time performance is not the only factor that implementations need to consider. The storage occupied by the jump table sometimes needs to be taken into account. In a simple implementation it is proportional to the difference between the maximum and minimum values appearing in the **case** labels (which may not be considered an efficient use of storage if there are only a few case labels used within this range). A more sophisticated technique than using a series of **if** statements is to create a binary tree of **case** label values and jump addresses. The value of the controlling expression being used to walk this tree to obtain the destination address. Some optimizers split the implementation into a jump table for those **case** label values that are contiguous and a binary tree for the out lying values.

Translator vendors targeting modern processors face an additional problem. Successful processors often contain a range of different implementations, creating a processor family, (e.g., the Intel Pentium series). These different processor implementations usually have different performance characteristics, and in the case of the **switch** statement different levels of sophistication in branch prediction. How does a translator make the decision on whether to use a jump table or **if** statements when the optimal code varies between different implementations of a particular processor?

A study by Uh and Whalley[6] compared (see Table 1753.1) the performance of a series of **if** statements and the equivalent jump table implementation. For three of the processors it was worth using a jump table when there were more than two **if** statements were likely to be executed. In the case of the UltraSPARC-1 the figure was more than eight **if** statements executed (this was put down to the lack hardware support for branch prediction of indirect jumps).

Table 1753.1: Performance comparison (in seconds) of some implementation techniques for a series of **if** statements (contained in a loop that iterated 10,000,000 times) using (1) linear search (LS), or (2) indirect jump (IJ), for a variety of processors in the SPARC family. *br* is the average number of branches per loop iteration. Based on Uh and Whalley.[6]

Processor Implementation	2.5br LS	4.5br LS	8.5br LS	2.5br IJ	4.5br IJ	8.5br IJ
SPARCstation-IPC	3.82	5.53	8.82	2.61	2.71	2.76
SPARCstation-5	1.03	1.65	2.74	0.63	0.76	0.76
SPARCstation-20	0.93	1.60	2.65	0.87	0.93	0.94
UltraSPARC-1	0.50	1.16	1.56	1.50	1.51	1.51

A **case** or **default** label is accessible only within the closest enclosing **switch** statement. 1754

Commentary

This requirement needs to be explicitly stated because there is no syntactic association between **case** labels and their controlling **switch** statement.

Coding Guidelines

statement visual layout

The issue most likely to be associated with a nested **switch** statement is source layout (because the amount of indentation used is often greater than in nested **if** statements). However, nested **switch** statements are relatively uncommon. For this reason the issue of the comprehension effort needed for this form of nested construct is not discussed.

The integer promotions are performed on the controlling expression. 1755

Commentary

integer promotions

The rationale for performing the integer promotions is the same as that for the operands within expressions.

Common Implementations

When the controlling expression is denoted by an object having a character type the possible range of values is known to fit in a byte. Even relatively simple optimizers often check for, and make use of, this special case.

1756 The constant expression in each **case** label is converted to the promoted type of the controlling expression.

Commentary

Prior to this conversion the type of the constant expression associated with each **case** label is derived from the form of the literals and result type of the operators it contains. The relationship between the value of a **case** label and a controlling expression is not the same as that between the operands of an equality operator. The conversion may cause the rank of the case label value to be reduced. If the types of both expressions are unsigned it is possible for the **case** label value to change (e.g., a modulo reduction). Like all integer conversions undefined behavior may occur for some values and types.

Other Languages

Many languages have a single integer type, so there is no conversion to perform for **case** label values. Strongly typed languages usually require that the type of the **case** label value be compatible with the type of the controlling expression, there is not usually any implicit conversions. Enumerated constants are often defined to be separate types, that are not compatible with any integer type.

Coding Guidelines

This C sentence deals with the relationship between individual **case** label values and the controlling expression. The following points deal with the relationship between different **case** label values within a given **switch** statement:

- Mixing **case** labels whose values are represented using both character constants and integer constants is making use of representation information (in this context the macro `EOF` might be interpreted in its symbolic form of representing an end-of-file character, rather than an integer constant). There does not appear to be a worthwhile benefit in having a deviation that permits the use of the integer constant `0` rather than the character constant `'\0'`, on the grounds of improved reader recognition performance. The character constant `'\0'` is the most commonly occurring character constant (10% of all character constants in the visible form of the `.c` files, even if it only represents 1% of all constant tokens denoting the value 0).
- Mixing **case** labels whose values are represented using both enumeration constants and some other form of constant representation (e.g., an integer constant) is making use of the underlying representation of the enumerated constants. The same is also true if enumerated constants from different enumerations types are mixed.
- Mixing integer constants represented using decimal, hexadecimal, or octal lexical forms. The issue of visually mixing integer constants having different lexical forms is discussed elsewhere. form of representation mixing

Floating point literals are very rarely seen in **case** labels. The guideline recommendation dealing with exact comparison of floating-point values is applicable to this usage. ?? equality operators not floating-point operands

Example

```
#include <limits.h>

enum {red, green, blue};

extern int glob;

void f(unsigned char ch)
```

```
{
switch (ch)
   {
   case 'a': glob++;
             break;

   case green: glob+=2;
               break;

   case (int)7.0: glob--;
                  break;

   case 99: glob -= 9;
            break;

   case ULONG_MAX: glob *= 3;
                   break;
   }
}
```

If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **case** label. 1757

Commentary

A **case** label can appear on any statement in the **switch** body.

```
switch (x)
   default : if (prime(x))
                case 2: case 3: case 5: case 7:
                   process_prime(x);
             else
                case 4: case 6: case 8: case 10:
                   process_composite(x);
```

Duff's Device

There can be more practical uses for this functionality.

Coding Guidelines

Experience suggests that developers treat the case label value as being the result of evaluating the expression appearing in the source (i.e., that no conversion, driven by the type of the controlling expression, takes place). A conversion that causes a change of value is very suspicious. However, no instances of such an event occur in the Usage `.c` files or have been experienced by your author. Given this apparent rarity no guideline recommendation is made here.

Otherwise, if there is a **default** label, control jumps to the labeled statement. 1758

Commentary

A **switch** statement may be thought of as a series of **if** statements with the **default** label representing the final **else** arm (although other **case** labels may label the same statement as a **default** label).

Common Implementations

Having a **default** label may not alter the execution time performance of the generated machine code. All of the tests necessary to determine that the default label should be jumped to are the same as those necessary to determine that no part of the switch should be executed (if there is no **default** label).

If no converted **case** constant expression matches and there is no **default** label, no part of the switch body is executed. 1759

Commentary

This behavior is the same as that of a series of nested **if** statements. If all of their controlling expressions are false and there is no final **else** arm, none of the statement bodies is executed.

Other Languages

Some languages require that there exist a **case** label value, or **default**, that matches the value of the controlling expression. If there is no such matching value the behavior may be undefined (e.g., Pascal specifies it is a *dynamic-violation*) or even defined to raise an exception (e.g., Ada).

Coding Guidelines

The coding guideline issue of always having a **default** label is discussed elsewhere.

1751 default label at most one

Implementation limits

1760 As discussed in 5.2.4.1, the implementation may limit the number of **case** values in a **switch** statement.

limit case labels

Commentary

This observation ought really to be a Further reference subclause.

1761 133) 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.

footnote 133

Commentary

If the declaration is not followed by any **case** or **default** labels, all references to the identifier it declares can only occur in the statements that follow it (which can only be reached via a jump to preceding **case** or **default** labels, unless a **goto** statement jumps to an ordinary label within the statement list occurs).

1762 EXAMPLE In the artificial program fragment

EXAMPLE case fall through

```
switch (expr)
{
        int i = 4;
        f(i);
case 0:
        i = 17;
        /* falls through into default code */
default:
        printf("%d\n", i);
}
```

the object whose identifier is **i** exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the **printf** function will access an indeterminate value. Similarly, the call to the function **f** cannot be reached.

Commentary

Objects with static storage duration are initialized on program startup.

static storage duration initialized before startup

```
switch (i)
   {
   static char message[] = "abc"; /* Not dependent on control flow. */
   case 0:
        f(message);
        break;
   case 1:
        /* ... */
   }
```

Other issues associated with constructs contained in this example are discussed elsewhere.

case fall through

1749 switch past variably modified type

References

1. Altrium BV. *C166/ST10 v8.0 C Cross-Compiler User's Guide*. Altrium BV, 2003.
2. J. Engblom. Static properties of commercial embedded real-time and embedded systems. Technical Report ASTEC Technical Report 98/05, Uppsala University, Sweden, Nov. 1998.
3. Hitachi Ltd. *H8S, H8/300 Series C Compiler User's Manual*. Hitachi Ltd, 4.0 edition, Sept. 1998.
4. L. Rosler. The evolution of C–past and future. *AT&T Bell Laboratories Technical Journal*, 63(8):1685–1699, 1984.
5. D. A. Spuler. Compiler code generation for multiway branch statements as a static search problem. Technical Report Technical Report 94/03, James Cook University, Jan. 1994.
6. G.-R. Uh and D. B. Whalley. Effectively exploiting indirect jumps. *Software–Practice and Experience*, 29(12):1061–1101, Oct. 1999.