Document Number: N3879
Date: 2014-01-17
Project: Programming Language C++
Evolution Working Group
Reply to: Andrew Tomazos
andrewtomazos@gmail.com

Explicit Flow Control: break label, goto case and explicit switch

0.1 Proposal [proposal]

0.1.1 Informal Summary [proposal.summary]

We propose adding to C++ some new jump statements and making available an explicit-specifier for switch statements.

The new jump statements are `break` *label*, `continue` *label* (same as Java), `goto case` *constant-expression* and `goto default` (same as C#).

An `explicit switch` statement (same as C#) causes each case block to have its own block scope, and to never flow off the end. That is, each case block must be explicitly exited. (The implicit fallthrough semantic between two consecutive case blocks can be expressed in an `explicit switch` using a `goto case` statement instead.)

0.1.2 Existing Practice [proposal.existing]

Each proposed addition has already been present in either Java or C# for many years, and so has been extensively tested by millions of programmers.

0.1.3 Motivation and Examples [proposal.motivation]

1 The break label and continue label forms are used to easily break or continue on an outer enclosing statement:

```
loop_ts:
  for (T t : ts)
    for (U u : us)
      if (f(u,v))
      {
        g(u,v);
        break loop_ts;
      }
```

2 The goto case statement is used to transfer control between case blocks in a switch:

```
switch (cond)
{
case foo:
  do_foo();
  break;

case bar:
  do_bar();
```

```
  goto case foo;

case baz:
  do_baz();
};
```

3 An explicit-specified switch is used to avoid the deadly accidental implicit fallthrough bug, and to declare local variables without adding redundant braces:

```
explicit switch (digit)
{
case 0:
case 1:
case 2:
  T t = f(0,2); // OK: see below
  return t.low();

case 4:
case 8:
  if (x % 2 == 0)
  {
    g();
    // ERROR: potential flow off end of explicit-switch case statement, use "goto default" instead
  }
  else
    throw logic_error("x must be even");

default:
  T t = f(4); // OK: The two names ‘t‘ are in different scope
  return t.high();
}
```

4 (Fun Historical Footnote: C++ was derived from C which was derived from B which was derived in part from BCPL. BCPL had the goto case statement semantic in the form of a docase statement.)

0.2 Technical Specifications [proposal.techspecs]

0.2.1 Grammar Additions [proposal.grammar]

labeled-statement:
  attribute-specifier-seq$_{opt}$ *identifier* : *statement*
  attribute-specifier-seq$_{opt}$ <u>*case-label*</u> : *statement*

<u>*case-label*</u>*:*
  `case` *constant-expression*
  `default`

jump-statement:
  `break` <u>*identifier$_{opt}$*</u> ;
  `continue` <u>*identifier$_{opt}$*</u> ;
  `return` *expression$_{opt}$*;
  `return` *braced-init-list* ;
  `goto` *identifier* ;
  `goto` <u>*case-label*</u> ;

selection-statement:
`if (` *condition* `)` *statement*
`if (` *condition* `)` *statement* `else` *statement*
`switch (` *condition* `)` *statement*
`explicit switch (` *condition* `) {` *case-statement-seq* `}`

case-statement-seq:
case-statement
case-statement-seq case-statement

case-statement:
case-label-seq statement-seq

case-label-seq:
attribute-specifier-seq$_{opt}$ *case-label* `:` *case-label-seq*$_{opt}$

0.3 Labeled statement [stmt.label]

1 A statement can be labeled.

labeled-statement:
attribute-specifier-seq$_{opt}$ *identifier* `:` *statement*
attribute-specifier-seq$_{opt}$ *case-label* `:` *statement*

case-label:
`case` *constant-expression*
`default`

The optional *attribute-specifier-seq* appertains to the label. An identifier label declares the identifier. The only use of an identifier label is as the target of a `goto`. The scope of an identifier label is the function in which it appears. Identifier labels shall not be redeclared within a function. An identifier label can be used in a `goto` statement before its definition. Identifier labels have their own name space and do not interfere with other identifiers.

2 A *case-label* shall only occur in an enclosing switch statement. A *case-label* is associated with its smallest enclosing `switch` statement.

0.4 Selection statements [stmt.select]

1 Selection statements choose one of several flows of control.

selection-statement:
`if (` *condition* `)` *statement*
`if (` *condition* `)` *statement* `else` *statement*
`switch (` *condition* `)` *statement*
`explicit switch (` *condition* `) {` *case-statement-seq* `}`

case-statement-seq:
case-statement
case-statement-seq case-statement

case-statement:
case-label-seq statement-seq

case-label-seq:
attribute-specifier-seq$_{opt}$ *case-label* : *case-label-seq*$_{opt}$

condition:
expression
attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator* = *initializer-clause*
attribute-specifier-seq$_{opt}$ *decl-specifier-seq declarator braced-init-list*

See [dcl.meaning] for the optional *attribute-specifier-seq* in a condition.

In Clause [stmt.stmt], the term *substatement* refers to the contained statement or statements that appear directly in the *selection-statement* syntax notation and to each *case-statement* of an `explicit switch` statement. Each substatement of a *selection-statement* implicitly defines a block scope. If a substatement of a *selection-statement* is a single statement, and not a *compound-statement* or a *case-statement,* it is as if it was rewritten to be a compound-statement containing the original substatement.

[*Example:*

```
if (x)
  int i;
```

can be equivalently rewritten as

```
if (x) {
  int i;
}
```

Thus after the `if` statement, `i` is no longer in scope. — *end example*]

2 The rules for conditions apply both to *selection-statements* and to the `for` and `while` statements (**??**). The declarator shall not specify a function or an array. If the `auto` *type-specifier* appears in the *decl-specifier-seq*, the type of the identifier being declared is deduced from the initializer as described in **??**.

3 A name introduced by a declaration in a condition (either introduced by the *decl-specifier-seq* or the declarator of the condition) is in scope from its point of declaration until the end of the substatements controlled by the condition. If the name is re-declared in the outermost block of a substatement controlled by the condition, the declaration that re-declares the name is ill-formed. [*Example:*

```
if (int x = f()) {
  int x;            // ill-formed, redeclaration of x
}
else {
  int x;            // ill-formed, redeclaration of x
}
```

— *end example*]

4 The value of a condition that is an initialized declaration in a statement other than a `switch` statement is the value of the declared variable contextually converted to `bool` (Clause **??**). If that conversion is ill-formed, the program is ill-formed. The value of a condition that is an initialized declaration in a `switch` statement is the value of the declared variable if it has integral or enumeration type, or of that variable implicitly converted to integral or enumeration type otherwise. The value of a condition that is an expression is the value of the expression, contextually converted to `bool` for statements other than `switch`; if that conversion is ill-formed,

the program is ill-formed. The value of the condition will be referred to as simply "the condition" where the usage is unambiguous.

5 If a condition can be syntactically resolved as either an expression or the declaration of a block-scope name, it is interpreted as a declaration.

6 In the *decl-specifier-seq* of a *condition*, each *decl-specifier* shall be either a *type-specifier* or `constexpr`.

0.4.1 The `switch` statement [stmt.switch]

1 The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

2 The condition shall be of integral type, enumeration type, or class type. If of class type, the condition is contextually implicitly converted (Clause **??**) to an integral or enumeration type. Integral promotions are performed. Any statement within the `switch` statement can be labeled with one or more case labels as follows:

`case` *constant-expression* `:`

where the *constant-expression* shall be a converted constant expression (**??**) of the promoted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion to the promoted type of the switch condition.

3 An `explicit switch` statement is a `switch` statement. Each *case-statement* within it is considered a single compound statement and defines a block scope. Each *case-label* in the *case-label-seq* of a *case-statement* is associated with the `explicit switch` statement and labels the *case-statement*. If a *case-label* can be syntactically resolved as labeling a *case-statement* or a *labeled-statement*, it is interpreted as labeling a *case-statement*.

4 The implementation shall analyze each but the last *case-statement* of every `explicit switch` statement during translation with some predicate P. P must have the following properties: If it is possible for control to flow off the end of a *case-statement* C, $P(C)$ must be true. If the final statement within a *case-statement* C is a *jump-statement* or a `throw` expression statement, $P(C)$ must be false. For each *case-statement* C with neither property, $P(C)$ is unspecified. If $P(C)$ is true for an analyzed *case-statement*, the program is ill-formed. [*Note:* As a quality of implementation issue, P should be false in as many of the unspecified cases as reasonably possible. — *end note*]

5 There shall be at most one label of the form

`default :`

within a `switch` statement.

6 Switch statements can be nested; a `case` or `default` label is associated with the smallest switch enclosing it.

7 When the `switch` statement is executed, its condition is evaluated and compared with each case constant. If one of the case constants is equal to the value of the condition, control is passed to the statement following the matched case label. If no case constant matches the condition, and if there is a `default` label, control passes to the statement labeled by the default label. If no case matches and if there is no `default` then none of the statements in the switch is executed.

8 `case` and `default` labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see `break`, 0.5.1. [*Note:* Usually, in a non-explicit switch statement the substatement that is the subject of a switch is compound and *case-labels* appear on the top-level statements contained within the (compound) substatement, but this is not required. Declarations can appear in the substatement of a *switch-statement*. — *end note*]

0.5 Jump statements [stmt.jump]

1 Jump statements unconditionally transfer control.

jump-statement:
  `break` *identifier*$_{opt}$;
  `continue` *identifier*$_{opt}$;
  `return` *expression*$_{opt}$;
  `return` *braced-init-list* ;
  `goto` *identifier* ;
  `goto` *case-label* ;

2 On exit from a scope (however accomplished), objects with automatic storage duration (**??**) that have been constructed in that scope are destroyed in the reverse order of their construction. [*Note:* For temporaries, see **??**. — *end note*] Transfer out of a loop, out of a block, or back past an initialized variable with automatic storage duration involves the destruction of objects with automatic storage duration that are in scope at the point transferred from but not at the point transferred to. (See **??** for transfers into blocks). [*Note:* However, the program can be terminated (by calling `std::exit()` or `std::abort()` (**??**), for example) without destroying class objects with automatic storage duration. — *end note*]

0.5.1 The `break` statement [stmt.break]

1 A `break` statement shall occur only in an *iteration-statement* or a `switch` statement. It causes termination of an enclosing *iteration-statement* or `switch` statement; control passes to the statement following the terminated statement, if any. If no identifier is given, the terminated statement is the smallest enclosing *iteration-statement* or `switch` statement. Otherwise, if there is an enclosing *iteration-statement* or `switch` statement labeled by the identifier, this statement is the terminated statement. Otherwise, the program is ill-formed.

0.5.2 The `continue` statement [stmt.cont]

1 The `continue` statement shall occur only in an *iteration-statement* and causes control to pass to the loop-continuation portion of an enclosing *iteration-statement*, that is, to the end of the loop. If no identifier is given, the *iteration-statement* is the smallest one. Otherwise, if there is an enclosing *iteration-statement* labeled by the identifier, this is the one. Otherwise, the program is ill-formed.

More precisely, in each of the statements

```
while (foo) {
  {
    // ...
  }
contin: ;
}
```

```
do {
  {
    // ...
  }
contin: ;
} while (foo);
```

```
for (;;) {
  {
    // ...
  }
contin: ;
}
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto contin`.

0.5.3 The `goto` statement [stmt.goto]

1 The `goto` statement unconditionally transfers control to a target statement.

2 If an identifier is specified, the target statement shall be labeled by that identifier and located in the current function.

3 Otherwise, a *case-label* is specified and the `goto` statement must be enclosed by a `switch` statement. The *case-label* is associated with the smallest enclosing switch statement of the `goto` statement. For `goto case` *constant-expression* - the *constant-expression* is evaluated in the same way as the other *case-labels* associated

with that switch statement. If there is a statement labeled with a *case-label* of the same value and associated with the same switch statement, the target statement is the one so labeled. Likewise for `goto default` and a statement labeled by `default`. If there is no such target statement, the program is ill-formed. [*Note:* A `goto case` statement does not jump to default if the value is not found, cannot jump to an outer switch statement and cannot exit a switch statement. — *end note*]