Labeled Break and Continue inside labeled block scopes

Document #:	P230217R0
Date:	2023-02-17
Project:	Programming Language C++
Audience:	Core Language Evolution (EWG), SG19, SG22

Reply-to:	Hypatia of Sva <hypatia.sva@posteo.eu>

1 Introduction

This paper proposes two new statements, the labeled break and continue statement, as well as a corresponding change in code generation, introducing two label-like sequence points generated per scope which has a label on the same line as the label inducing opening brace, which are only jumpable to from within the scope. (In this sense, it might be called a "local" or "scoped goto".)

2 Motivation and Scope

A longstanding goal of C++, and of higher level languages in general, is the elimination of goto statements whereever possible. In this regard, C++ could get rid of a whole lot of cases, in which C has to refer to goto+ statements, like "goto error" for error handling with exceptions (although ther is still much debate about error handling, as exceptions are not always the right tool depending on the domain).

However, there is one use of goto, that is necessary to this day in C++, and that is the continuing of an outer loop within an inner loop without introducing a flag variable. There might be a number of reasons, why a variable is not desired, but the most obvious is code clarity. Examples are provided in the comparisons, there it is also visible that they are useful for more than just loop continuation, although that is the primary purpose of this proposal.

The primary point of comparison for this proposal is the named CYCLE and EXIT statment of Fortran 90 and newer (cf. 8.1.4.4.3-4, p.99 of [N692]). The statement introduced here is broader, but should be compatible to it in order to facilitate translation from fortran programs to C++.

The benefit is, that we don't expose labels to the compiler that could be called even from outside the encompassing scope. This is a serious danger of a "break_outer" label, and is not by defaut catchable by tooling, since a label indicates a manual change of control flow. However, breaking or continuing a scope is in line with standard control flow and shouldn't break it, this is fixed by standardizing these constructs in a uniform manner.

Because this is a low level language construct, it seems not a bad idea to coordingate this change with a similar change of the C language in WG14, however, if this is not feasible, then C++ should adopt this first and propose it to C later via SG22 or other means if possible.

3 Before/After Comparisons

3.1 Exiting a loop early with a flag

Before / After

```
for(int i = 0; i < n; i++) {
    flag = true;
    for(int j = 0; j < n; j++) {
      flag = f(i,j);
      if(flag) break;
    }
    if(flag) break;
}
```

```
        for(int i = 0; i < n; i++)
outer:  {
            for(int j = 0; j < n; j++) {
                if(f(i,j)) break outer;
            }
        }
```

3.2 Exiting a loop early with a jump

Before / After

```
//goto break_outer can be called here to skip the loop

for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      if(f(i,j)) goto break_outer;
    }
}
break_outer:
```

```
//break outer can't be called here to skip the loop

        for(int i = 0; i < n; i++)
outer:  {
            for(int j = 0; j < n; j++) {
                if(f(i,j)) break outer;
            }
        }
```

3.3 Continuing a loop from within an inner loop

Before / After

```
outer_loop: do {
    initialize_transformation(my_object);
    for(int j = 0; j < n; j++) {
      some_transformation(my_object);
      if(have_to_rewind(my_object)) goto outer_loop;
    }
} while (!done(my_object));

//goto outer_loop can be called here
```

```
        do
outer:  {
            initialize_transformation(my_object);
            for(int j = 0; j < n; j++) {
              some_transformation(my_object);
              if(have_to_rewind(my_object)) continue outer;
            }
            break;
        } while (!done(my_object));

//continue outer can't be called here
```

3.4 Continuing a loop from within a switch

Before / After

```
outer_loop: for(int i = 0; i < n; i++) {
    switch(i) {
      case 3: goto outer_loop; break;
      case 5: f(x); break;
      default: break;
    }
}

//goto outer_loop can be called here
```

```
        for(int i = 0; i < n; i++)
outer:  {
            switch(i) {
              case 3: continue outer; break;
              case 5: f(x); break;
              default: break;
            }
        }


//continue outer can't be called here
```

3.5 Error handling with an error block

Before / After / After with single return

```
void do_something(int i) {
  if(!f(i)) goto error;
  if(!g(i)) goto error;
  do {
    bool sucess = true;
    // some complicated loop that might indicate its not sucessful
    if(!sucess) goto error;
    switch(result) {
        //...
        case 27:
          //special error case by an external API
          log_error("Error 27!");
          goto error;
          break;
        //...
    }
  } while(determine_condition(i));
  return;

  error:
  cleanup(i);
  // more cleanup code
}
```

```
void do_something(int i) {
  main_block: {
    if(!f(i)) break main_block;
    if(!g(i)) break main_block;
    do {
      bool sucess = true;
      // some complicated loop that might indicate its not sucessful
      if(!sucess) break main_block;
      switch(result) {
          //...
          case 27:
            //special error case by an external API
            log_error("Error 27!");
            break main_block;
            break;
          //...
      }
    } while(determine_condition(i));

    return;
  }

  cleanup(i);
  // more cleanup code
}
```

Before / After / After with single return

```
void do_something(int i) {
  bool error = false;
  main_block: {
    if(!f(i)) {
      error = true;
      break main_block;
    }
    if(!g(i)) {
      error = true;
      break main_block;
    }
    do {
      bool sucess = true;
      // some complicated loop that might indicate its not sucessful and gives a result
      if(!sucess) {
        error = true;
        break main_block;
      }
      switch(result) {
          //...
          case 27:
            //special error case by an external API
            log_error("Error 27!");
            error = true;
            break main_block;
            break;
          //...
      }
    } while(determine_condition(i));
  }

  if(error) {
    cleanup(i);
    // more cleanup code
  }

  //singular return point.
}
```

4 Design Overview

4.1 Syntax

`break` *identifier* `;`
`continue` *identifier* `;`

4.2 Model

Every labeled block

```
label: {
  //...
}
```

shall be expanded to

```
label: {
__BLOCK__BEGIN__label:
  //...
__BLOCK__LOOP__CONTINUE__label:
}
__BLOCK__END__label:
```

where `__BLOCK__BEGIN__label`, `__BLOCK__LOOP__CONTINUE__label` and `__BLOCK__END__label` are compiler generated labels inserted after the opening and closing curly brace, and are uniquely tied to the label. In other words, for a block labeled `something_else:` it would generate the labels `__BLOCK__BEGIN__something_else` and `__BLOCK__END__something_else`.

The statement `break` *label* `;` shall be identical to `goto __BLOCK_END__` *label* `;`.

The statement `continue` *label* `;` shall be identical to `goto __BLOCK_BEGIN__` *label* `;` if the scope is not the body of an iteration statement, and identical to `goto __BLOCK__LOOP__CONTINUE__` *label* `;` if it is.

The case differentiation makes the labeled `continue` statement behave like the ruglar `continue` statement in loops, and treats each other scope like the body of an infinite loop. In this way, every scope can be continued without changing the semantics for loops specifically.

Important: These labels are not accessible to the language user! They don't need to be named like here specified. This model merely represents the way, these statments work, and a possible way for them to be implemented.

The compiler needs to a) implement these statments, as if they were written with these `goto`'s, and b) make sure that no other symbols but the label before the opening curly brace are exported to the user.

4.3 Constraints

The labeled `break` and `continue` statements are only to be used within the block enclosed with the curly braces that match the beginning curly brace after the statement.

Any other use of these statements is a language error. A compiler must not allow them in a compilation mode relating to strict standard compatibility (typically expressed by compiletime options like `-pedantic`) and must give a diagnostics. Any implementation that allows a `break` or `continue` statement out of scope is considered not standard conform or is regarded as a non standard extension.

Also, all language implementation constraints of `goto` statements equally apply to the labeled `break` and `continue` statements. (That is because they are implemented "as if" they were goto statements, whether they are implemented with such a preprocessing step or not.)

5 Proposed Wording

The following is the proposed change to the standard document, based on [N4928].

Change **§8.7 [stmt.jump.general]** by adding the last two options:

jump-statement:
`break` ;
`continue` ;
`return` *expr-or-braced-init-list*$_{opt}$;
coroutine-return-statement
`goto` *identifier* ;
`break` *identifier* ;
`continue` *identifier* ;

Add new sections **§8.7 [stmt.break.block]**:

A scoped `break` statement shall be enclosed by a block scope with the same label before the opening curly brace. It causes contol to pass to the first point after the corresponding closing brace. More precisely, in the statement

```
label: {
  //...
}
__BLOCK__END__label:
```

a `break label;` is equivalent to `goto __BLOCK__END__label;`.

And **§8.7 [stmt.continue.block]**:

A scoped `continue` statement shall be enclosed by a block scope with the same label before the opening curly brace. It causes contol to pass to the first point after that opening brace, or to the end of the scope, if the scope is a loop body, executing the equivalent control flow as if a `continue` statement would be passed on that level of identation. More precisely, in the statement

```
label: {
__BLOCK__BEGIN__label:
  //...
__BLOCK__LOOP__CONTINUE__label:
}
```

a `continue label;` is equivalent to `goto __BLOCK__BEGIN__label;`. if the scope is not the body of an iteration statement, and equivalent to `goto __BLOCK__LOOP__CONTINUE__label;` if it is.

6 References

[N4928] Working Draft, Standard for Programming Language C++.
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/n4928.pdf

[N692] Fortran 90 (ISO/IEC 1539 : 1991 (E)).
https://wg5-fortran.org/N001-N1100/N692.pdf