Document number: **P0082R2**
Date: 2017-02-06
Prior versions: N3587, P0082R0, P0082R1
Audience: Evolution Working Group
Reply to: **Alan Talbot**
cpp@alantalbot.com

For Loop Exit Strategies (Revision 3)

Abstract

This proposal describes an enhancement to the iteration statements that allows the specification of two optional blocks of code, one that executes on normal completion of the loop (when the loop condition is no longer met), and one that executes on early termination (when the loop is exited with a **break**).

Changes in Revision 3 since P0082R1

This version has been revised to reflect feedback from the EWG discussion in Jacksonville (March 2016). The use of existing keywords has been dropped in favor of using new keywords to identify the blocks. I have also pointed out the relationship between this proposal and P0305R1.

Changes in Revision 2 since P0082R0

This version has again been considerably rewritten to reflect feedback from the EWG discussion in Kona (October 2015). In particular, the **if for** construct has been dropped in favor of a cleaner solution using existing keywords to identify the blocks. I have also removed all the alternative approaches that were considered; they may be found in P0082R0.

The Problem

On a fairly regular basis I find myself writing code that looks something like this:

```
auto it = get_begin(. . .);         // Unfortunate that ‘it’ has to be out here.
auto end = get_end(. . .);          // Unfortunate that ‘end’ has to be out here.
for (; it != end; ++it)
{
   if (some_condition(*it)) break;
   do_something(*it);
}
if (it == end)                      // Extra test here.
   do_stuff();
else
   do_something_else(*it);
```

This is rather annoying, involves an unnecessary test, and hoists the loop iterator and (sometimes) the terminator out into the surrounding scope. One could of course avoid the scope problem by putting an extra set of braces around the code, but this makes the code even harder to read and is too easy to forget or simply neglect in the interests of readability.

If you have a situation where you can't evaluate the loop condition twice, then you have to introduce a flag to keep track of how the loop exited. For example:

```
bool early = false;
while (some_condition())
{
   . . .
   if (test1()) { early = true; break; }
   . . .
   if (test2()) { early = true; break; }
   . . .
   if (test3()) { early = true; break; }
   . . .
}
if (early)
{ . . . }
else
{ . . . }
```

Assuming that `some_condition` and the tests cannot be called again outside the loop, either for performance or logical reasons, you need a flag in the outer scope and must remember to set it every time you break. This simple example could be written as a function with returns instead of breaks, but there are often reasons why doing so would be difficult or less clear.

The problem gets even worse with range-based **for** loops. Because the loop variable cannot be hoisted out of the loop, it is not possible to test for normal completion as you can with most iterator-based loops, so again you need a flag, or you have to resort to **goto** hopscotch. And if you want to know the value of the loop variable when you exit prematurely (as you almost always do) you will have to make a copy of it (if possible):

```
something_t last;                   // Extra construction here.
bool early = false;
for (auto&& element : container)
{
   if (some_condition(element))
   {
      last = element;               // Extra copy here
      early = true;
      break;
   }
   do_something(element);
}
if (early)
   do_something_else(last);
else
   do_stuff();
```

The extra construction in the outer scope requires stating a type which might be hard to state, or might not be copyable or movable (or even default constructible). Regardless, this is clearly not an improvement over a conventional **for** statement, so the advantage of range-based **for** has been lost. (In this example, the need for `last` could be eliminated by calling `do_something_else` from inside the loop, but that can become impractical if there are a number of early exit points and the early termination code is not a simple function call.

The solution is for the language to provide a way to catch either normal or early termination. This is especially important now that C++ has range-based **for** loops.

The Solution

Overview

The proposed solution is a pure extension to the language requiring two new keywords. The iteration statements (**for**, **while** and **do**) will have an optional **on_complete** block to catch the normal termination case and an optional **on_break** block to catch the early termination case.

Here are the three earlier examples with the benefit of this feature:

```
for (auto it(get_begin(. . .)), end(get_end(. . .)); it != end; ++it)
{
   if (some_condition(*it)) break;
   do_something(*it);
}
on_complete
   do_stuff();
on_break
   do_something_else(*it);
```

```
while (some_condition())
{
   . . .
   if (test1()) break;
   . . .
   if (test2()) break;
   . . .
   if (test3()) break;
   . . .
}
on_break
{ . . . }
on_complete
{ . . . }
```

```
for (auto&& element : container)
{
   if (some_condition(element)) break;
   do_something(element);
}
on_complete
   do_stuff();
on_break
   do_something_else(element);
```

Any declared variables remain in scope in both the normal termination and early termination blocks, and only one of the blocks is executed. Control transfers to the normal termination block if and when the loop condition is no longer met (even if the loop body is never entered), and to the early termination block if and only if the loop exits with a **break**. Neither block is ever required, and the blocks may appear in either order.

Multiple breaks

The early termination block also provides for graceful multiple breaks. Suppose you want to iterate over a three-dimensional table and choose a particular cell. Today you might write something like this:

```
vector<vector<vector<. . .>>> table = . . .;
for (auto& x : table)
   for (auto& y : x)
      for (auto& z : y)
         if (some_condition(z))
         {
               do_something(z);
               goto DONE;
         }
DONE:
```

This isn't too bad, but it gets worse if you have different exit situations. This particular problem could be solved by putting the loops in a function, then returning from the innermost loop, but not all algorithms are so easily encapsulated. With early termination blocks, you can do this:

```
for (auto& x : table)
   for (auto& y : x)
      for (auto& z : y)
         if (some_condition(z))
         {
               do_something(z);
               break;
         }
      on_break break;
   on_break break;
```

This scales well to more complicated cases since you can either continue or break on either termination condition. I would expect that the compiler could collapse the repeated breaks into a single jump, so the efficiency of the **goto** solution would be preserved.

Are braces required?

An interesting question is whether the termination blocks should require braces like **try**/**catch** blocks, or be consistent with the rest of the language and not require them. Personally I am not fond of the brace requirement for **try**/**catch**, and I propose that braces not be required for termination blocks. Given their similarity to **else** blocks, I think this provides the greatest consistency.

Is the early termination block necessary?

I feel that the early termination case is very important. In an informal look at my current code base, I found that the number of cases where I needed the early termination block was about equal to the number of cases where I didn't.

Similarity to Python

Python has half of this feature. It uses **else** for the normal termination block, but lacks the early termination block. Unfortunately we can't use **else** in C++ because it would change the meaning of existing code, and I am convinced that the early termination block is important.

Importance

It is quite reasonable to ask if this feature is worth it. Are the instances where it improves readability, encapsulation and performance sufficiently common and compelling to motivate a language change? The reaction to the first version of this paper tells me that the answer is resoundingly yes. People really seem to like this idea.

This feature also fits very well with the new C++17 selection statement initializers (P0305R1). The motivation for selection statement initializers is that they limit the scope of variables without requiring extra blocks, which degrade readability and are easy to forget. Termination blocks offer the same advantages.

I did an informal survey of the code I work on and found a number of cases that match the first example above, and found that the cases requiring only normal termination were about as numerous as those requiring early termination. My search looked only at **for** statements with no loop variable declaration. I did not look for **while** or **do** cases, nor did I look for places where the logic could be reorganized and simplified by this feature. I suspect that if the feature were available, I would find a use for it in many more places.

Syntax

The syntax of this proposal has changed several times as a result of input from EWG. There were significant objections to approaches that avoided keywords or used existing keywords in new ways. There was strong agreement that new keywords made for a cleaner and clearer solution. There was a discussion about two-word (whitespace) keywords, but this approach also raised some objections.

Another quite different approach that was suggested by several people was making the iteration statements in effect return a value. This is a very different and more complicated design that does not in my opinion add enough value to be worth the complication, and is harder to understand. There were considerable objections to this as well.

The spelling of the keywords is a matter that I believe worthy of some consideration. Choosing terms is often referred to as “bike shedding”, but *that* term was coined to refer to Parkinson's law of triviality, and I do not believe terminology in programming is a trivial issue. I feel choosing the correct terms for things is an important aspect of software engineering.

I have proposed a pair of keywords that I believe will work well, but I welcome a discussion on the matter and would be glad to change them. For example, the underscores could be removed (**oncomplete** and **onbreak**), and **on_default** (or **ondefault**) has been proposed instead of **on_complete**.

Specifics

I will provide formal wording in a future revision of this proposal. Meanwhile here are the basics. Note that this is just for exposition and not meant to be a draft of the final version of the grammar or wording.

iteration-statement:

while (*condition*) *statement termination-block*$_{opt}$ *termination-block*$_{opt}$

do *statement* **while** (*expression*) **;** *termination-block*$_{opt}$ *termination-block*$_{opt}$

for (*for-init-statement condition*$_{opt}$ **;** *expression*$_{opt}$) *statement termination-block*$_{opt}$ *termination-block*$_{opt}$

for (*for-range-declaration* **:** *for-range-initializer*) *statement termination-block*$_{opt}$ *termination-block*$_{opt}$

termination-block:

on_complete *statement*

on_break *statement*

Both the normal (**on_complete**) termination block and the early (**on_break**) termination block are optional. There may be at most one of each type of termination block. The termination blocks may appear in either order.

If a loop exits normally because the loop condition fails, the normal termination block will be executed and the early termination block will not be executed. If a loop exits prematurely because of a break statement, the early termination block will be executed and the normal termination block will not be executed.

If a **for** or **while** statement declares a loop variable or variables, the scope of the name(s) declared includes the termination blocks. In the case of a range-based **for** loop, the value of the loop variable is undefined in the normal termination block. In all other cases the value of the loop variable(s) will be the terminating value when entering the normal termination block, and the value at the time of the break when entering the early termination block.

Acknowledgements

Beman Dawes reviewed an early draft of this proposal and suggested several excellent clarifications. Clark Nelson reviewed the final draft of the first version and caught several mistakes. Thanks again for your help.

A number of people made insightful comments about the first version of this proposal. Thanks to Niels Dekker, Niall Douglas, Folkert van Heusden, Nick Maclaren, Sarfaraz Nawaz, Dwayne Robinson, Sam Saariste, Diego Sánchez, Mike Spertus, and Daveed Vandevoorde for their contributions. P0082R0 has more details of their contributions.

Many people in Kona (October 2015) made helpful comments and suggestions. Chandler Carruth pointed out the case where the condition cannot be evaluated twice. Chandler and James Touton made very compelling arguments for following the Python design of **else** as the normal termination block. I'm sure I will miss someone if I try to list everyone else who commented, but the rest of you know who you are and have my thanks.

Thanks very much to JC van Winkel, Walter Brown and Paul Sepe for reviewing Revision 2 and providing many helpful suggestions.