Document number: **P0082R1**
Date: 2016-02-14
Prior versions: N3587, P0082R0
Audience: Evolution Working Group
Reply to: **Alan Talbot**
cpp@alantalbot.com

For Loop Exit Strategies (Revision 2)

Abstract

This proposal suggests an enhancement to the iteration statements to allow the specification of two 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 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.

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. In this simple example, all this could be put into a function with returns instead of breaks, but often there are reasons why that would be difficult or less clear.

The problem is 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 have to have a flag or resort to nasty **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). You end up with something like this:

```
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();
```

Note that the extra construction in the outer scope requires stating the type. That type might be hard to state. Or the type might not be copyable or movable (or even default constructible). Anyway, this is clearly not an improvement over the conventional **for** version, so the advantage of range-based **for** has been lost.

In this simple example, the `last` variable could be eliminated by calling `do_something_else` from inside the loop, but that becomes impractical if there are a number of early exit points and `do_something_else` is actually a bunch of code rather than a simple function call.

What I would really like—especially when I'm using range-based **for** loops—is for the language to provide a way to catch either normal or early termination.

The Solution

Overview

The proposed solution requires no new keywords and is a fairly simple, pure extension to the language. The iteration statements (**for**, **while** and **do**) will have an optional **catch default** block to catch the normal termination case and an optional **catch 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);
}
catch default
   do_stuff();
catch break
   do_something_else(*it);
```

```
while (some_condition())
{
   . . .
   if (test1()) break;
   . . .
   if (test2()) break;
   . . .
   if (test3()) break;
   . . .
}
catch break
{ . . . }
catch default
{ . . . }
```

```
for (auto&& element : container)
{
   if (some_condition(element)) break;
   do_something(element);
}
catch default
   do_stuff();
catch 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.

Is the early termination block required?

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.

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;
         }
      catch break
         break;
   catch 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.

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 believe we want the early termination block.

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 am proposing that braces not be required for termination blocks. Given their similarity to **else** blocks, I think this provides the greatest consistency.

Specifics

I will provide formal wording for section 6.5 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:

catch default *statement*

catch break *statement*

Both the normal (**catch default**) termination block and the early (**catch 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.

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.

I also did an informal survey of the code base I work on, and I found quite a number of cases where I was doing exactly what the first example above does, and observed 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 with the feature available, I would use it in many more places.

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 made many helpful comments and suggestions. Chandler Carruth pointed out the case where the condition cannot be evaluate twice. Chandler and James Touton made very compelling arguments for following the Python design of **else** as the normal termination block (but unfortunately without the **if for** construct, **else** cannot be used). 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 this latest draft and providing many helpful suggestions.