Document number: **P0082R0**
Date: 2015-09-24
Project: Programming Language C++
Reference: N4527, N3587
Reply to: **Alan Talbot**
cpp@alantalbot.com

For Loop Exit Strategies (Revision 1)

Abstract

This proposal suggests an enhancement to the iteration statements to allow the specification of two blocks of code that execute on completion of a loop: one for normal termination (when the loop condition is no longer met) and the other for early termination (when the loop is exited with a **break**).

Changes in Revision 1

This version has been considerably rewritten, and now includes suggestions from several people who commented on N3587. The examples have been clarified and the pseudo-wording has been removed. I have also added **if do** and **if while**.

The Problem

I often find myself writing code that looks something like this:

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

This is rather annoying, involves an unnecessary test, and hoists `it` out into the surrounding scope. The problem gets much worse with range-based **for** loops. There it is not possible to declare the loop variable outside the **for** statement, so the best I can do is something like this:

```
something_t last;                   // Extra construction here.
for (auto&& elem : cont)
{
   if (some_condition(elem))
   {
      last = elem;                  // Extra copy here
      goto EARLY;
   }
   do_something(elem);
}
do_stuff();
goto DONE;
EARLY:
do_something_else(last);
DONE:
```

This is pretty awful. Note the extra construction in the outer scope that requires stating the type. That might not even be possible (if the type isn't default constructible or copyable). This is clearly not an improvement over the conventional **for** version, so the point of range-based **for** has been lost. In this case I would simply use the conventional version above.

In some cases I could eliminate the `last` variable and call `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 several lines of code rather than a simple function call. And I would still need the **goto**.

What I'd really like to do is have the language provide me a way to optionally catch the two cases, normal and early termination. This would be especially useful with range-based **for** statements.

A Solution

Overview

Here is a solution that I think would be quite natural, and would be a simple, pure extension to the language:

```
if
   for (auto&& elem : cont)
   {
      if (some_condition(elem)) break;
      do_something(i);
   }
{        // Normal termination: the loop condition failed.
   do_stuff();
}
else     // Early termination: a break was encountered.
{
   do_something_else(elem);  // Note that elem is in scope here.
}
```

This is a very intuitive construct: the semantics of the **if for** statement retain the exact sense of the **if** statement. The syntax is not too foreign—there are other examples of intermingled statements in C++, e.g. the function try block, and Duff's device. Note that the **for** statement is not in parentheses—it is not an expression.

The declared variable remains in scope in both the normal termination and early termination (**else**) blocks, and only one of the termination 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 executed), and to the early termination block if the loop exits with a **break**.

Python

Niels Dekker (quoting Sam Saariste) pointed out that Python has exactly this construct, but without the early termination clause, and with **else** introducing the normal termination clause. However, I do not recommend following the Python syntax because it is counterintuitive, to the point that Summerfield calls the Python normal termination clause "rather confusingly named."[1]

Are braces required?

An interesting question is whether the normal termination block braces should be required. Leaving them off could be confusing for the human reader, and Clark Nelson suggested that they be required. However, they are not required in any other similar situation, so that could also be confusing. Without them, this example would be legal:

```
if
   for (int i = 0; i < 10; ++i)
      if (foo(i)) break;           // Loop body if statement.
   cout << "no foo" << '\n';       // Normal termination statement.
else
   cout << "foo at " << i << '\n'; // Early termination statement.
```

And the same care would have to be taken as with any nested if statements:

```
if
   for (int i = 0; i < 10; ++i)
      if (foo(i)) break;           // Loop body if statement.
      else bar(i);                 // Loop body else clause.
else
   cout << "foo at " << i << '\n'; // Early termination statement.
cout << "done" << '\n';            // Next statement after the if-for.
```

Removing the first else gives a completely different meaning.

```
if
   for (int i = 0; i < 10; ++i)
      if (foo(i)) break;           // Loop body if statement.
else
   cout << "foo at " << i << '\n'; // Loop body else clause.
cout << "done" << '\n';            // Normal termination statement.
```

I recommend not requiring them, but I'm happy to change the proposal to require them if there is consensus that they improve readability.

Multiple breaks

The **if for** statement also provides for a graceful multiple break. Suppose I want to iterate over a three-dimensional table and choose a particular cell. Today I would probably do 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 is a little ugly, and gets even worse if you have different exit situations. (I could solve this particular problem by writing a function that returns from the inner loop, but not all such constructs are easily put into a function.) With **if for** you can do this:

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

Do and while

Diego Sánchez pointed out that this approach works equally well with **while** and **do**:

```
if
   while (auto p = get_next())
   {
      if (some_condition(p)) break;
      do_something(p);
   }
else
   do_something_else(p);
```

Specifics

I am proposing to add a new **if** form to the iteration statements in section 6.5. I will provide formal wording in a revision of this proposal if there is sufficient interest to proceed.

```
if
   for (...)    // Normal for loop, either conventional or range-based.
   {
      // For loop block.
      // Braces are not required for a single statement.
   }
{  // Normal termination block.
      // Executed if loop "succeeds" by exiting normally.
      // May be omitted entirely if the else is present.
      // Loop iteration variable is in scope.
      // (In a range-based for statement, it's value is undefined here.)
      // Braces are not required for a single statement.
}
else
{  // Early termination block.
      // Executed if loop "fails" by exiting prematurely with a break.
      // May be omitted entirely if the normal termination block is present.
      // Loop iteration variable is in scope.
      // Braces are not required for a single statement.
}
```

If and only if the early termination (**else**) substatement is present, then the normal termination substatement may be omitted. If the **for** statement declares a loop variable or variables, the scope of the name(s) declared includes the normal termination substatement and the early termination substatement. In the case of a range-based for loop, the value of the loop variable is undefined in the normal termination substatement. (It is in scope simply for consistency with conventional **if for** statements.)

```
if
   while (...)
   {
   }
{
}
else
{
}
```

The while statement has the same semantics and syntax as the **for** statement, including the scope of a loop variable, if any.

```
if
   do
   {
   } while (...);
{
}
else
{
}
```

The **do** statement has the same semantics and syntax as the **for** statement.

Other Possible Solutions

Then

If C++ had a **then** keyword, a very clean syntax would be possible:

```
for (auto&& elem : cont)
{
   if (some_condition(elem)) break;
   do_something(i);
}
then   // Normal termination: the loop condition failed.
{
   do_stuff();
}
else   // Early termination: a break was encountered.
{
   do_something_else(elem);  // Note that elem is in scope here.
}
```

Unfortunately it is very difficult to add a (reasonably spelled) new keyword to the language. And adding **then** (regardless of spelling) poses the problem that people would expect it to work with **if** statements. Sarfaraz Nawaz suggested using **do** rather than **then**. This avoids these problems, but makes the construct confusing to use with **do** loops. Dwayne Robinson suggested using **finally**. The main problems with this are that it is a new keyword, and that its meaning would be subtly different from the meaning in Java.

Statements as expressions

Niall Douglas, Mike Spertus and others have suggested that the **for** statement in effect become a boolean expression. This is certainly intriguing. It would make the syntax slightly more "normal" than this proposal, while retaining the same basic syntax, and it allows for some interesting constructs, such as assigning the "result" of a loop statement to a variable.

However, it introduces a concept that is completely foreign to C++ (statements as expressions), which would presumably have wide-ranging and subtle consequences. It also seems more complicated than the proposed solution. For these reasons, I am not recommending this approach, but I have no objection to it, and if there is interest I am happy to explore it further.

Catching breaks

Another possibility suggested by Daveed Vandevoorde, Nick Maclaren and others is to allow break and continue "catch" blocks:

```
for (auto&& elem : cont)
{
   if (some_condition(elem)) break ONE;
   if (another_condition(elem)) break TWO;
   do_something(i);
}
continue  // Normal termination: the loop condition failed.
{
   do_stuff();
}
break ONE // Early termination: break ONE was encountered.
{
   do_action_one(elem);             // Would elem be in scope here?
}
break TWO // Early termination: break TWO was encountered.
{
   do_action_two(elem);             // Would elem be in scope here?
}
```

This has several advantages. It allows for multiple breaks with different behavior, and it would be useful with **switch** statements. It would be very nice if the name could be optional if there is only one break catch block. For this to be fully useful, the loop variable would need to be in scope in all the catch blocks.

My main concern is that it is a fairly significant change to the language. However, my personal needs would be very well met by this approach, and I would be happy to go in this direction if there is consensus that it is preferred.

Importance

It is a good question to ask, is this worth it? Are the instances where this construct improves readability, encapsulation and performance sufficiently common and compelling? The reaction to the first version of this paper tells me that the answer is resoundingly yes. People really seem to like this idea, and every person who read the paper and responded represents many, many others out there who are not involved in this process.

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.

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.

Notes

1: Summerfield, Mark. Programming in Python 3 – A Complete Introduction to the Python Language, p. 151, Addison-Wesley, 2009.