Lambda-Capture Expressions

An **init capture** expression enables a lambda to add a data member, initialized with an arbitrary expression, to its closure.

Description

In C++11, lambda expressions can capture variables in the surrounding scope either *by copy* or *by reference*:

```
void test0()
{
    int i = 0;
    auto f0 = [i]{ };   // Capture i by copy.
    auto f1 = [&i]{ };  // Capture i by reference.
}
```

Here, we use the familiar C++11 feature **auto** (see Section 2.1."**auto** Variables" on page 195) to deduce a closure's type since there is no way to name such a type explicitly.

Although one could specify *which* and *how* existing variables were captured, the programmer had no control over the creation of new variables within a **closure** (see Section 2.1. "Lambdas" on page 573). C++14 extends the **lambda-introducer** syntax to support implicit creation of data members inside a closure using an arbitrary initializer:

```
auto f2 = [i = 10]{ /* body of closure */ };
    // Synthesize an int data member, i, copy-initialized with 10.

auto f3 = [c{'a'}]{ /* body of closure */ };
    // Synthesize a char data member, c, direct-initialized with 'a'.
```

Note that the identifiers `i` and `c` above do not refer to any existing variables; they are specified by the programmer creating the closure. For example, the closure type bound to `f2` above is similar in functionality to an **invocable** **struct** containing an **int** data member:

```
struct f2LikeInvocableStruct
{
    auto i = 10;  // The type int is deduced from the initialization expression.
    auto operator()() const { /* closure body */ }  // The struct is invocable.
};
```

The type of the data member is deduced from the initialization expression provided as part of the capture in the same vein as **auto** (see Section 2.1."**auto** Variables" on page 195) type deduction; hence, it's not possible to synthesize an uninitialized closure data member: