Lambdas Chapter 2   Conditionally Safe Features

Note that a variable named in a lambda capture isn't automatically *captured*. A variable is captured only if it is ODR-used within the lambda expression:

```cpp
#include <algorithm>  // std::min

void f9()
{
        int a = 0;  // a is not a compile-time constant.
    const int b = 2;  // b is     a compile-time constant.

    auto c1 = [&]{ return 2 * a; };       // a is ODR-used; implicitly captured.
    auto c2 = [&]{ return sizeof(a); };  // a is not ODR-used; not captured.
    auto c3 = [&]{ return 2 * b; };       // b is not ODR-used; not captured.
    auto c4 = [&]{ return &b; };          // b is ODR-used; implicitly captured.
    auto c5 = [&]{ std::min(b, 5); };     // b is ODR-used; implicitly captured.
}
```

In the above example, the lambda body for c1 ODR-uses a by reading its value and thus captures a. Conversely, c2 does *not* capture a despite its name being used in the lambda body because it is only used in the **unevaluated operand** of the **sizeof operator**, which does not constitute the variable's ODR-use. Similarly, c3 does *not* capture b because (1) b is a **compile-time constant** and (2) c3 only uses b's value, which also does not constitute ODR-use of b (see Section 2.1."**constexpr** Variables" on page 302). Finally, taking the address of or binding a reference to a variable *always* constitutes the variable's ODR-use; hence, both c4, which directly takes the address of b, and c5, which passes b by **const**& to std::min, capture b.

Finally, a lambda capture within a **variadic function template** (see Section 2.1."Variadic Templates" on page 876) may contain a **pack expansion**:

```cpp
#include <utility>  // std::forward

template <typename... ArgTypes>
int f10(const char* s, ArgTypes&&... args);

template <typename... ArgTypes>
int f11(ArgTypes&&... args)
{
    const char* s = "Introduction";
    auto c1 = [=]{ return f10(s, args...); };  // OK, args... captured by copy
    auto c2 = [s,&args...]{ return f10(s, std::forward<ArgTypes>(args)...); };
        // OK, explicit capture of args... by reference
}
```

In the example above, the variadic arguments to f11 are implicitly captured using capture by copy in the first lambda expression. Capturing by copy means that, regardless of the **value category** (*rvalue*, *lvalue*, and so on) of the original arguments, the captured variables are all *lvalue* members of the resulting closure. Conversely, the second lambda expression captures