Finally, the purpose of a closure is to be invoked. It can be invoked immediately by supplying arguments for each of its parameters:

```
#include <iostream>  // std::cout
void f3()
{
    [](const char* s) { std::cout << s; }("hello world\n");
        // equivalent to std::cout << "hello world\n";
}
```

The closure object, in this example, is invoked immediately and then destroyed, making the above just a complicated way to say `std::cout << "hello world\n";`. More commonly, the lambda expression is used as a local function for convenience and to avoid clutter:

```
#include <cmath>  // std::sqrt

double hypotenuse(double a, double b)
{
    auto sqr = [](double x) { return x * x; };
    return std::sqrt(sqr(a) + sqr(b));
}
```

Note that the closure's call operator cannot be **overloaded**. There is no syntax to express multiple call operators on the same closure type.

The most common use of a lambda expression, however, is as a callback to a function template, e.g., as a functor argument to an algorithm from the Standard Library:

```
#include <algorithm>  // std::partition

template <typename FwdIt>
FwdIt oddEvenPartition(FwdIt first, FwdIt last)
{
    using value_type = decltype(*first);
    return std::partition(first, last, [](value_type v) { return v % 2 != 0; });
}
```

The `oddEvenPartition` function template moves odd values to the start of the sequence and even values to the back. The closure object is invoked repeatedly within the `std::partition` algorithm.

Lambda capture and lambda introducer

The purpose of the lambda capture is to make certain local variables from the environment available to be used (or, more precisely, **ODR-used**, which means that they are used in a **potentially evaluated** context) within the **lambda body**. Each local variable