Notice that when we invoke the recursive lambda, we pass it as an argument to itself, both to the external call and to the internal recursive calls. To avoid this somewhat awkward interface, a special function object called a Y Combinator can be used.[3] The Y Combinator object holds the closure object to be invoked recursively and passes it to itself:

```
#include <utility>  // std::move, std::forward

template <typename Lambda>
class Y_Combinator {
    Lambda d_lambda;

public:
    Y_Combinator(Lambda&& lambda) : d_lambda(std::move(lambda)) { }

    template <typename... Args>
    decltype(auto) operator()(Args&&...args) const
    {
        return d_lambda(*this, std::forward<Args>(args)...);
    }
};

template <typename Lambda>
Y_Combinator<Lambda> Y(Lambda lambda) { return std::move(lambda); }
```

The function call operator for `Y_Combinator` is a variadic function template (see Section 2.1. "Variadic Templates" on page 873) that passes itself to the stored closure object, `d_lambda`, along with zero or more additional arguments supplied by the caller. Thus, `d_lambda` and the `Y_Combinator` are mutually recursive functors. The `Y` function template constructs a `Y_Combinator` from a lambda expression.

To use a `Y_Combinator`, pass a recursive generic lambda to `Y`; the resulting object is the one that we would call from code:

```
auto fib2 = Y([](auto self, int n) -> int
{
    if (n < 2) { return n; }
    return self(n - 1) + self(n - 2);
});

int fib8 = fib2(8);  // returns 21
```

Note that the recursive lambda still needs to take `self` as an argument, but because `self` is a `Y_Combinator`, it does not need to pass `self` to itself. Unfortunately, we must now specify the return type of the lambda because the compiler cannot deduce the return type of the mutually recursive invocations of `self`. The usefulness of a Y Combinator in C++

[3] **hindley86**