The `callFuncPtr` function takes a callback in the form of a pointer to function. Even though it is not a template, it can be called with a lambda argument having the same parameter types, the same return type, and an empty lambda capture; the closure object is converted to an ordinary pointer to function. This conversion is *not* available in the second call to `callFuncPtr` because the lambda capture is not empty.

Conversion to function pointer is considered a user-defined conversion operator and thus cannot be implicitly combined with other conversions on the same expression. It can, however, be invoked *explicitly*, as needed:

```
using Fp2 = int(*)(int);  // function-pointer type

struct FuncWrapper
{
    FuncWrapper(Fp2) { /*...*/ }  // implicit conversion from function-pointer
    // ...
};

int f2(FuncWrapper);
int i2 = f2([](int x) { return x; });  // Error, two user-defined conversions
int i3 = f2(static_cast<Fp2>([](int x) { return x; }));  // OK, explicit cast
int i4 = f2(+[](int x) { return x; });  // OK, forced conversion
```

The first call to `f2` fails because it would require two implicit user-defined conversions: one from the closure type to the `Fp2` function-pointer type and one from `Fp2` to `FuncWrapper`. The second call succeeds because the first conversion is made explicit with the **`static_cast`**. The third call is an interesting shortcut that takes advantage of the unary **`operator+`** being defined as the identity transformation for pointer types. Thus, the closure-to-pointer conversion is invoked for the operand of **`operator+`**, which returns the unchanged pointer, which, in turn, is converted to `FuncWrapper`; the first and third steps of this sequence use only one user-defined conversion each. The Standard Library `std::function` class template provides another way to pass a function object of unnamed type, one that does not require the lambda capture to be empty; see *Use Cases — Use with `std::function`* on page 601.

The compile-time and runtime phases of defining a closure type and constructing a closure object from a single lambda expression resembles the phases of calling a function template; what looks like an ordinary function call is actually broken down into a compile-time instantiation and a runtime call. The closure type is deduced when a lambda expression is encountered during compilation. When the control flow passes through the lambda expression at run time, the closure object is *constructed* from the list of captured local variables. In the `numAboveAverageSalaries` example on page 576, the `SalaryIsGreater` class can be thought of as a closure type — created by hand instead of by the compiler — whereas the call to `SalaryIsGreater(average)` is analogous to constructing the closure object at run time.