```
    auto  c4 = []{ return 2; };         // OK, captureless lambda expression
    using C4t = decltype(c4);

    class C4Derived : public C4t        // OK, inherit from closure type.
    {
        int d_auxValue;
    public:
        C4Derived(C4t c4, int aux) : C4t(std::move(c4)), d_auxValue(aux) { }
        int aux() const { return d_auxValue; }
    };

    static_assert(sizeof(C4Derived) == sizeof(int), "");  // OK, EBO applied

    int ret = callFunc([i]{ return 2 * i; });  // OK, deduced arg type, Func

    c1b = c1;  // Error, assignment of closures is not allowed.
}
```

The types of c1 and c2, above, are different, even though they are token-for-token identical. As there is no way to explicitly name a closure type, we use **auto** in the case of c1 and c2 in f1 (see Section 2.1.“**auto** Variables” on page 195) or template-argument deduction in the case of f in callFunc to create variables directly from the lambda expression, and we use **decltype** (see Section 1.1.“**decltype**” on page 25) to create aliases to the types of existing closure variables (C1t and C2t). Note that using **decltype** directly on a lambda expression is ill formed, as shown with C3t~~, because~~ there would be no way to construct an object of the resulting unique type.[3] The derived class, C~~1~~Derived, uses the type alias C~~1~~t to refer to its base class. Note that its constructor forwards its first argument to the base-class move constructor.

There is no way to specify a closure type prior to creating an actual closure object of that type. Consequently, there is no way to declare callFunc with a parameter of the actual closure type that will be passed; hence, it is declared as a template parameter. As a special case, however, if the lambda capture is *empty* (i.e., the lambda expression begins with []; see Section 2.2.“Lambda Captures” on page 986), then the closure is implicitly convertible to an ordinary function pointer having the same signature as its call operator:

```
char callFuncPtr(char (*f)(const char*)) { return f("x"); }  // not a template

char c = callFuncPtr([](const char* s) { return s ? s[0] : '\0'; });
    // OK, closure argument is converted to function-pointer parameter.

char d = callFuncPtr([c](const char* s) { /*...*/ });
    // Error, lambda capture is not empty; no conversion to function pointer.
```

[3] Since C++20, lambda expression are allowed to appear in unevaluated contexts, including operands of **decltype** and **sizeof**.