Evaluating a lambda expression creates a temporary **closure** object of an unnamed type called the **closure type**. Each part of a lambda expression is described in detail in the subsections below.

Closures

A lambda expression looks a lot like an unnamed function definition, and it is often convenient to think of it that way, but a lambda expression is actually more complex than that. First, a lambda expression, as the name implies, is an *expression* rather than a *definition*. The result of evaluating a lambda expression is a special function object called a closure[1]; it is not until the closure is *invoked* — which can happen immediately but often occurs later (e.g., as a callback) — that the *body* of the lambda expression is executed.

Evaluating a lambda expression creates a temporary **closure object** of an unnamed type called the closure type. The closure type encapsulates captured variables (see Section 2.2. "Lambda Captures" on page 986) and has a call operator that executes the body of the lambda expression. Each lambda expression has a unique closure type, even if it is identical to another lambda expression in the program. If the lambda expression appears within a template, the closure type for each instantiation of that template is unique. Note, however, that, although the closure object is an unnamed temporary object, it can be saved in a named variable whose type can be queried. Closure types are copy constructible and move constructible, but they have no other constructors and have deleted assignment operators.[2] Interestingly, it is possible to *inherit* from a closure type, provided the derived class constructs its closure type base class using only the copy or move constructors. This ability to derive from a closure type allows taking advantage of the empty-base optimization (EBO):

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

template <typename Func>
int callFunc(const Func& f) { return f(); }

void f1()
{
    int   i   = 5;
    auto  c1  = [i]{ return 2 * i; };   // OK, deduced type for c1
    using C1t = decltype(c1);           // OK, named alias for unnamed type
    C1t   c1b = c1;                     // OK, copy of c1
    auto  c2  = [i]{ return 2 * i; };   // OK, identical lambda expression
    using C2t = decltype(c2);
    C1t   c2b = c2;                     // Error, different types, C1t & C2t
    using C3t = decltype([]{/*...*/});  // Error, lambda expr within decltype
```

[1]The terms *lambda* and *closure* are borrowed from *Lambda Calculus*, a computational system developed by Alonzo Church in the 1930s. Many computer languages have features inspired by Lambda Calculus, although most (including C++) take some liberties with the terminology. See **rojas15** and **barendregt84**.

[2]C++17 provides default constructors for captureless lambdas, which are assignable in C++20.