known return type (**double**). The second lambda expression returns a value by brace initialization, which is insufficient for deducing a return value. Again, the issue is resolved by specifying the return type explicitly. Note that, unlike ordinary functions, a lambda expression cannot have a return type specified before the lambda introducer or lambda declarator:

```
auto c5 = int []     ()        { return 0; };  // Error, return type misplaced
auto c6 =     [] int ()        { return 0; };  // Error, return type misplaced
auto c7 =     []     () -> int { return 0; };  // OK, trailing return type
```

Attributes (see Section 1.1.“Attribute Syntax” on page 12) that appertain to the *type* of call operator can be inserted in the lambda declarator just before the trailing return type. If there is no trailing return type, the attributes can be inserted before the open brace of the lambda body. Unfortunately, these attributes do not appertain to the call operator itself, but to its type, ruling out some common attributes:

```
#include <cstdlib>  // std::abort
auto c1 = []() noexcept [[noreturn]] {  // Error, [[noreturn]] on a type
    std::abort();
};
```

Lambda body

Combined, the lambda declarator and the lambda body make up the declaration and definition of an **inline** member function that is the call operator for the closure type. For the purposes of name lookup and the interpretation of **this**, the lambda body is considered to be in the context where the lambda expression is evaluated (independent of the context where the closure’s call operator is invoked).

Critically, the set of entity names that can be used from within the lambda body is not limited to captured local variables. Types, functions, templates, constants, and so on — just like for any other member function — do not need to be captured and, in fact, *cannot* be captured in most cases. To illustrate, let’s create a number of entities in multiple scopes:

```
#include <iostream>  // std::cout

namespace ns1
{
    void f1() { std::cout << "ns1::f1" << '\n'; }
    struct Class1 { Class1() { std::cout << "ns1::Class1()" << '\n'; } };
    int g0 = 0;
}

namespace ns2
{
    void f1() { std::cout << "ns2::f1" << '\n'; }

    template <typename T>
    struct Class1 { Class1() { std::cout << "ns2::Class1()" << '\n'; } };
```