```
struct __lambda_2  // compiler-generated name; not visible to the user
{
    template <typename __T>
    __T operator()(__T x) const { return x; }
    // ...
};

__lambda_1 identityInt = __lambda_1();
__lambda_2 identity    = __lambda_2();
```

Note that the names `__lambda_1`, `__lambda_2`, and `__T` are for descriptive purpose and are not available to the user; the compiler might choose any name or no name for these **entities**.

A generic lambda is any lambda expression having one or more parameters declared using the placeholder type **`auto`**. The compiler generates a **template parameter** type for each **`auto`** parameter in the generic lambda, and that type is substituted for **`auto`** in the function-call operator's parameter list. In the `identity` example above, **`auto`** `x` is replaced with `__T x`, where `__T` is a new template parameter type. When user code subsequently calls, e.g., `identity(42)`, normal template **type deduction** takes place, and **`operator()<int>`** is instantiated.

Lambda capture and mutable closures

The closure type produced by a generic lambda is not a **class template**. Rather, its function-call operator and its conversion-to-function-pointer **operator** (as we'll see later in *Conversion to a pointer to function* on page 974) are **function templates**. In particular, the **lambda capture**, which creates **data members** within the closure type, has the same syntax and semantics for all lambda expressions, **generic** or not. Similarly, the **mutable qualifier** has the same effect for generic lambdas as for nongeneric lambdas:

```
#include <algorithm>  // std::for_each
#include <iterator>   // std::next

template <typename FwdIter>
auto secondBiggest(FwdIter begin, FwdIter end)
    // Return the second-largest element in the range [begin, end),
    // assuming at least two elements and that all values in the range
    // are distinct.
{
    auto second = std::next(begin);  // Refer to second element.
    auto ret = *second;              // Set to second element.
    std::for_each(second, end,
        [biggest = *begin, &ret](const auto& element) mutable
        {
            if (biggest <= element) {
```