as though the function call operator were a static member function template of the **closure** and the conversion operator returned a pointer to that member function.

Use Cases

Reusable lambda expressions

One of the benefits of lambda expressions is that they can be defined within a function, close to the point of use. Saving a lambda expression in a variable allows it to be reused within the function. This reusability is greater for generic lambdas than for nongeneric lambdas, just as function templates are more reusable than ordinary functions. Consider, for example, a function that partitions a vector of strings and a vector of vectors based on the length of each element:

```
#include <vector>     // std::vector
#include <string>     // std::string
#include <algorithm>  // std::partition

void partitionByLength(std::size_t                    pivotLen,
                       std::vector<std::vector<int>>& v1,
                       std::vector<std::string>&      v2)
{
    auto condition = [pivotLen](const auto& e) { return e.size() < pivotLen; };

    std::partition(v1.begin(), v1.end(), condition);
    std::partition(v2.begin(), v2.end(), condition);
}
```

The `condition` generic lambda can be used to partition both `vector`s because its function call operator can be instantiated on either element type. The capture of `pivotLen` is performed only once, when the lambda expression is evaluated to yield `condition`.

Applying a lambda to each element of a tuple

An `std::tuple` is a collection of objects having heterogeneous types. We can apply a functor to each element of a `tuple` by using some **metaprogramming** features of the C++14 Standard Library:

```
#include <utility>  // std:::index_sequence, std::make_index_sequence
#include <tuple>    // std::tuple, std::tuple_size, std::get

template <typename Tpl, typename F, std::size_t... I>
void visitTupleImpl(Tpl& t, F& f, std::index_sequence<I...>)
{
```