```
struct __lambda_3
{
    template <typename... __T> auto operator()(int i, __T&&... args) const
    {
        return std::make_tuple(i, std::forward<decltype(args)>(args)...);
    }
};
```

The standard limitations on variadic function templates apply. For example, only a variadic parameter pack at the end of the parameter list will match function call arguments at the point of invocation. In addition, because **auto** is not permitted in a template-specialization parameter, the usual methods of defining function templates with multiple variadic parameter packs do not work for generic lambdas:

```
// Attempt to define a lambda expression with two variadic parameter packs.
auto y12 = [](std::tuple<auto...>&, auto...args) { };
    // Error, auto is a template argument in tuple specialization.
```

Conversion to a pointer to function

A nongeneric lambda expression with an empty lambda capture can be converted implicitly to a function pointer with the same signature. A generic lambda with an empty lambda capture can similarly be converted to a regular function pointer, where the parameters in the prototype of the target pointer type drive deduction of the appropriate **auto** parameters in the generic lambda signature:

```
auto y1 = [](int a, char b) { return a; };   // nongeneric lambda
int (*f1)(int, char) = y1;                   // OK, conversion to pointer

auto y2 = [](auto a, auto b) { return a; };  // generic lambda
int    (*f2)(int, int)    = y2;  // OK, instantiates operator()<int, int>
double (*f3)(double, int) = y2;  // OK, instantiates operator()<double, int>
char   (*f4)(int, char)   = y2;  // Error, incorrect return type
```

If the function target pointer is a variable template (see Section 1.2."Variable Templates" on page 157), then the deduction of the arguments is delayed until the variable template itself is instantiated:

```
template <typename T> int (*f5)(int, T) = y2;  // variable template
int (*f6)(int, short) = f5<short>;             // instantiate f5<short>
```

Each function pointer is produced by calling a conversion operator on the closure object. In the case of a generic lambda, the conversion operator is also a template. Template-argument deduction and return-type deduction are performed on the conversion operator; then the conversion operator instantiates the function-call operator. Intuitively, it is as