```
    void operator()(double (*f)(T)) const { }  // OK, can deduce T
};

struct ManualY9
{
    template <typename R, typename T>
    void operator()(R T::* m) const { }  // OK, can deduce R and T
};
```

`ManualY7`, `ManualY8`, and `ManualY9` benefit of type deduction for their template arguments, whereas `y7`, `y8`, and `y9` are unable to deduce **auto**. In addition, `ManualY9` deduces two template arguments for a single function parameter. There is a trade-off between the benefits of lambda expressions, such as **defining** a function **in place** at the point of use, and the pattern-matching power of manually written function templates. See *Annoyances — Cannot use full power of template-argument deduction* on page 981.

A default value on an **auto** parameter, while allowed, is not useful because it **defaults** only the *value* and not the *type* of the parameter. Invocation of such a generic lambda requires the programmer to either supply a value for the argument, which defeats the point of a defaulted argument, or explicitly instantiate **operator()**, which is gratuitously awkward:

```
void g4()
{
    auto y = [](auto a = 3) { return a * 2; };
    y(5);                     // OK, returns an int with value 10
    y();                      // Error, cannot deduce type for parameter a
    y.operator()<int>();      // OK, returns an int with value 6
    y.operator()<double>();   // OK, returns a double with the value 6.0
}
```

Variadic generic lambdas

If a placeholder argument to a generic lambda is followed by an ellipsis (...), then the parameter becomes a variadic **parameter pack**, and the function-call operator becomes a **variadic function template**; see Section 2.1."Variadic Templates" on page 873:

```
#include <tuple>  // std::tuple, std::make_tuple
auto y11 = [](int i, auto&&... args)
{
    return std::make_tuple(i, std::forward<decltype(args)>(args)...);
};

std::tuple<int, const char*, double> tpl1 = y11(3, "hello", 1.2);
```

The `y11` closure object forwards all of its arguments to `std::make_tuple`. The first argument must have type convertible to **int**, but the remaining arguments can have any type. Assuming the invented name `__T` for the **template parameter pack**, the generated function-call operator for `y11` would have a variadic **template parameter list**: