```
auto y1 = [](const auto& r) { };  // Match anything (read only).
auto y2 = [](auto&& r)      { };  // Match anything (forwarding reference).
auto y3 = [](auto& r)       { };  // Match only lvalues.
auto y4 = [](auto* p)       { };  // Match only pointers.
auto y5 = [](auto(*p)(int)) { };  // Match only pointers to functions.
auto y6 = [](auto C1::* pm) { };  // Match only pointers to data members of C1.

void g2()
{
    int       i1 = 0;
    const int i2 = 1;

    y1(i1);        // OK, r has type const int&.
    y2(i1);        // OK, r has type int&.

    y3(5);         // Error, argument is not an lvalue.
    y3(i1);        // OK, r has type int&.
    y3(i2);        // OK, r has type const int&.

    y4(i2);        // Error, i2 is not a pointer.
    y4(&i2);       // OK, p has type const int*.

    y5(&f1);       // OK, p has type double (*)(int).

    y6(&C1::d_i);  // OK, pm has type double C1::*.
}
```

To understand how `y1` and `y2` match any argument type, recall that **`auto`** is a placeholder for a template type argument, say, `__T`. As usual, **`const`** `__T& r` can bind to a **`const`** or non**`const`** *lvalue* or a temporary value created from an *rvalue*. The argument `__T&& r` is a **forwarding reference** (see Section 2.1."Forwarding References" on page 377); `__T` will be deduced to an *rvalue* if the argument to `y2` is an *rvalue* and to an *lvalue* reference otherwise. Because the parameter type for `r` is unnamed — we invented the name `__T` for descriptive purposes only — we must use **`decltype`**`(r)` to refer to the type of `r`:

```
#include <utility>  // std::move, std::forward
#include <cassert>  // standard C assert macro

struct C2
{
    int d_value;

    explicit C2(int i)      : d_value(i)                { }
    C2(const C2& original) : d_value(original.d_value) { }
    C2(C2&& other)          : d_value(other.d_value)    { other.d_value = 99; }
};
```

```
void g3()
{
    auto y1 = [](const auto& a) { C2 v(a); };
    auto y2 = [](auto&&      a) { C2 v(std::forward<decltype(a)>(a)); };

    C2 a(1);

    y1(a);             assert(1  == a.d_value);  // copies from a
    y1(std::move(a)); assert(1  == a.d_value);  //    "     "  a
    y2(a);             assert(1  == a.d_value);  //    "     "  a
    y2(std::move(a)); assert(99 == a.d_value);  // moves    "  a
}
```

In this example, `y1` always invokes the copy constructor for `C2` because `a` has type **`const`** `C2&` regardless of whether we instantiate it with an *lvalue* or *rvalue* reference to `C2`. Conversely, `y2` forwards the value category of its argument to the `C2` constructor using `std::forward` according to the common idiom for forwarding references. If passed an *lvalue* reference, the copy constructor is invoked; otherwise, the move constructor is invoked. We can tell the difference because `C2` has a move constructor that puts the special value `99` into the moved-from object.

The **`auto`** placeholder in a generic lambda parameter cannot be a type argument in a template specialization, a parameter type in the prototype of a function reference or function pointer, or the class type in a pointer to member[1]:

```
#include <vector>  // std::vector
auto y7 = [](const std::vector<auto>& x) { };  // Error, invalid use of auto
auto y8 = [](double (*f)(auto)) { };           // Error,    "     "   "   "
auto y9 = [](int auto::* m) { };               // Error,    "     "   "   "
```

Because of this restriction, there are no contexts where more than one **`auto`** is allowed to appear in the declaration of a single lambda parameter. Template parameters *are* allowed in these contexts for regular function templates, so generic lambdas are less expressive than handwritten functor objects in this respect:

```
struct ManualY7
{
    template <typename T>
    void operator()(const std::vector<T>& x) const { }  // OK, can deduce T
};

struct ManualY8
{
    template <typename T>
```

[1]GCC 10.2 (c. 2020) does allow **`auto`** in both template arguments and function prototype parameters and deduces the template parameter type in the same way as for a regular function template. MSVC 19.29 (c. 2021) allows **`auto`** in the parameter list for a function reference or function pointer but not in the other two contexts.