```
int    local2b = func2();     // Valid? Call one of the definitions of func2.
char   local2c = func3();     // OK, call to inline function func3
char   local2d = func4("a");  // OK, call to instantiation func4<const char>

// file3.cpp:
#include <file1.h>            // Error, IFNDR, redefinition of func2
auto func1() { return 1.2; }  // OK, defined to return double
double local3a = func1();     // OK
int    local3b = func2();     // Valid? Call one of the definitions of func2.
char   local3c = func3();     // OK, call to inline function func3
char   local3d = func4("b");  // OK, call to instantiation func4<const char>
```

Because `func1` is declared in `file1.h` but defined in `file3.cpp`, `file2.cpp` does not have enough information to deduce its return type. Conversely, `func2` has the reverse problem: there is an ODR violation because `func2` is redefined in every translation unit that has `#include <file1.h>`. The compiler is not required to diagnose most ODR violations, but linkers will typically complain about multiply-defined public symbols. Finally, `func3` is `inline` and `func4` is a template; like `func1`, their definitions are visible in each translation unit, making the deduced return type available, but unlike `func2`, they do not create an ODR violation.

Placeholders in trailing return types

If **`auto`** or **`decltype(auto)`** is used in a trailing return type, the meaning is the same as using the same placeholder as a leading return type:

```
auto f1() -> auto;
auto f2() -> decltype(auto);
auto f3() -> const auto&;

auto           f1();  // OK, compatible redeclaration of f1
decltype(auto) f2();  // OK, compatible redeclaration of f2
const auto&    f3();  // OK, compatible redeclaration of f3
```

When any trailing return type is specified, the leading return-type placeholder must be plain **`auto`**:

```
decltype(auto) f4() -> auto;  // Error, decltype(auto) with trailing return
auto&          f5() -> int&;  // Error, auto& with trailing return
```

Deduced return types for lambda expressions

As described in Section 2.1."Lambdas" on page 573, the return type of a **closure** call operator can be deduced automatically from its **`return`** statement(s):

```
auto y1 = [](int& i)                  { return i += 1; };  // Deduce int.
```