

## Trailing Function Return Types

This syntactically more convenient yet semantically equivalent alternative of using `->` to declare a function’s return type *after* its parameter list enables that type to refer to the individual parameters by name along with other class or namespace members without explicit qualification.

### Description

C++11 offers an alternative function-declaration syntax in which the return type of a function is located to the right of its **signature** (name, parameters, and qualifiers), offset by the arrow token (`->`); the function itself is introduced by the keyword **auto**, which acts as a type placeholder:

```
auto f() -> void; // equivalent to void f();
```

When using the alternative, trailing-return-type syntax, any **const**, **volatile**, and reference qualifiers (see Section 3.1. “Ref-Qualifiers” on page 1153) are placed to the left of the `-> *<return-type>*`, and any contextual keywords, such as **override** and **final** (see Section 1.1. “**override**” on page 104 and Section 3.1. “**final**” on page 1007), are placed to its right:

```
struct Base
{
    virtual int e() const; // const qualifier
    virtual int f() volatile; // volatile qualifier
    virtual int g() &; // lvalue-reference qualifier
    virtual int h() &&; // rvalue-reference qualifier
};

struct Derived : Base
{
    auto e() const -> int override; // override contextual keyword
    auto f() volatile -> int final; // final " "
    auto g() & -> int override; // override " "
    auto h() && -> int final; // final " "
```

Using a trailing return type allows the parameters of a function to be named as part of the specification of the return type, which can be useful in conjunction with **decltype**:

```
auto g(int x) -> decltype(x); // equivalent to int g(int x);
```

When using the trailing-return-type syntax in a member function definition outside the class definition, names appearing in the return type, unlike with the classic notation, will be looked up in class scope by default: