《现代C++编程指南》：尽可能使用 auto 类型占位符

自动类型推导是现代 C++ 中最重要且广泛使用的特性之一。新的 C++ 标准使得在各种上下文中可以使用 auto 作为类型的占位符，并让编译器推导出实际的类型。在 C++11 中，auto 可用于声明局部变量以及具有尾随返回类型的函数的返回类型。在 C++14 中，auto 可用于无需指定尾随类型的函数的返回类型以及 lambda 表达式中的参数声明。未来的标准版本可能会将 auto 的使用扩展到更多的情况。

在这些上下文中使用 auto 有几个重要的好处，都将在“工作原理……”部分中讨论。开发者应该意识到它们，并在可能的情况下优先使用 auto。Andrei Alexandrescu 提出了一个专门的术语，并由 Herb Sutter 推广——几乎总是 auto（AAA）。

如何使用

考虑在以下情况中使用 auto 作为实际类型的占位符：

- 在不想明确指定具体类型时，使用 `auto name = expression` 的形式来声明局部变量。

```
auto i = 42; // int
auto d = 42.5; // double
auto s = "text"; // char const *
auto v = { 1, 2, 3 }; // std::initializer_list<int>
```

- 当需要明确指定具体类型时，使用 `auto name = type - id { expression }` 的形式来声明局部变量。

```
auto b = new char[10]{ 0 }; // char*
auto s1 = std::string {"text"}; // std::string
auto v1 = std::vector<int> { 1, 2, 3 }; // std::vector<int>
auto p = std::make_shared<int>(42); // std::shared_ptr<int>
```

- 当定义命名 lambda 函数时，使用 `auto name = lambda - expression` 的形式，除非该 lambda 函数需要被传递给函数或者从函数中返回。

```
auto upper = [](char const c) {return toupper(c); };
```

- 用于声明 lambda 表达式的参数和返回值。

```
auto add = [](auto const a, auto const b) {return a + b;};
```

- 当你不想明确指定具体类型时，用于声明函数的返回类型。

```
template <typename F, typename T>
auto apply(F&& f, T value)
{
   return f(value);
}
```

工作原理

auto 关键字本质上是一个实际类型的占位符。使用 auto
时，编译器会根据以下情况推导出实际类型：

- 当使用 auto 声明变量时，从用于初始化变量的表达式的类型推导。
- 当 auto
  用作函数返回类型的占位符时，从函数的尾随返回类型或者返回表达式的类型推导。

在一些情况下，必须明确指定一个特定的类型。例如，在上一节中的第一个示例里，编译器推导出 s 的类型为 `char const*`。如果意图是得到一个 std::string
类型，那么必须在右侧明确指定类型。同样地，v 的类型被推导为
std::initializer_list，但意图可能是得到一个 std::vector
类型，在这种情况下，必须在赋值右侧明确指定类型。

使用 auto 关键字而不是实际类型有一些重要的好处；以下是一些可能最重要的好处：

- 不可能出现变量未初始化的情况。这是开发者在声明指定实际类型的变量时经常犯的一个常见错误。然而，使用 auto
  时，由于需要通过初始化变量来推导类型，所以不可能出现这种情况。
- 使用 auto 能确保总是使用正确的类型，并且不会发生隐式转换。考虑以下示例，我们获取一个向量的大小到一个局部变量中。在第一种情况下，变量的类型为 int，尽管 size() 方法返回 size_t 类型。这意味着将会发生从 size_t 到 int 的隐式转换。然而，使用 auto 作为类型将会推导出正确的类型，即 size_t 类型。

  ```
  auto v = std::vector<int>{ 1, 2, 3 };

  // implicit conversion, possible loss of data
  int size1 = v.size();

  // OK
  auto size2 = v.size();

  // ill-formed (warning in gcc/clang, error in VC++)
  auto size3 = int{ v.size() };
  ```

- 使用 auto 有助于推广良好的面向对象编程实践，例如相比于实现更倾向于接口。指定的类型越少，代码就越通用，对未来变化的适应性就越强，这是面向对象编程的一个基本原则。
- 这意味着更少的打字量，并且减少对那些我们实际上并不关心的实际类型的关注。通常情况下，即使我们明确指定了类型，实际上也并不关心它。一个非常常见的情况是迭代器，但还有很多其他情况也是如此。当你想要遍历一个范围时，你并不关心迭代器的实际类型。你只对迭代器本身感兴趣；所以，使用 auto 可以节省输入可能很长的类型名称所花费的时间，并且有助于你将注意力集中在实际的代码上而不是类型名称上。在以下示例中，在第一个 for 循环里，我们明确地使用了迭代器的类型。这需要输入大量的文字；冗长的语句实际上可能会降低代码的可读性，并且你还必须要知道那些实际上并不关心的类型名称。而第二个带有 auto
  关键字的循环看起来更简洁，并且让你免去了输入和对实际类型的关注：

```
std::map<int, std::string> m;
for (std::map<int, std::string>::const_iterator
    it = m.cbegin();
    it != m.cend(); ++it)
{ /*...*/ }

for (auto it = m.cbegin(); it != m.cend(); ++it)
{ /*...*/ }
```

- 使用 auto 声明变量能够在赋值的右侧始终如一地保持一致的编码风格（类型总是出现在右侧）。如果动态分配对象，就需要在赋值的左右两侧都写出类型，例如 `int* p = new int(42)`。而使用 auto 时，类型仅在右侧指定一次。

然而，在使用 auto 时存在一些需要注意的地方：

- auto 关键字仅仅是一个类型的占位符，而不是针对
  const（常量）、volatile（易失性）和引用限定符的占位符。如果需要一个 const/volatile 和/或引用类型，那么就需要显式地指定它们。
- 在以下示例中，foo.get() 返回一个对 int 的引用；当变量 x
  从返回值初始化时，编译器推导出的类型是 int，而不是 int&。因此，对 x
  所做的任何改变都不会传播到 foo.x_ 中。为了使改变能够传播，我们应该使用 auto&。

```
class foo {
    int x_;
public:
    foo(int const x = 0) :x_{ x } {}
    int& get() { return x_; }
};
```

```
foo f(42);
auto x = f.get();
x = 100;
std::cout << f.get() << '₩n'; // prints 42
```

- 不可能将 auto 用于不可移动的类型。

  ```
  auto ai = std::atomic<int>(42); // error
  ```

- 不可能将 auto 用于多单词的类型，例如 `long long`、`long double` 或者结构体 foo。然而，在第一种情况下（`long long` 和 `long double`），可能的解决方法是用字面量或者类型别名；至于第二种情况（结构体 foo），在 C++ 中以这种形式使用结构体/类仅仅是为了与 C 兼容，并且无论如何都应该避免使用。

  ```
  auto l1 = long long{ 42 }; // error
  using llong = long long;
  auto l2 = llong{ 42 }; // OK
  auto l3 = 42LL; // OK
  ```

- 如果你使用 auto 关键字但仍需要知道类型，在大多数集成开发环境（IDE）中，可以通过将光标悬停在变量上来查看类型。然而，一旦离开 IDE，就不再可能这样做，唯一知道实际类型的方法就是根据初始化表达式自己推导，这可能意味着要在代码中搜索函数返回类型。

auto 可以用于指定函数的返回类型。在 C++11 中，这需要在函数声明中使用尾随返回类型。在 C++14 中，这一要求已经放宽，编译器会根据返回表达式推导出返回值的类型。如果有多个返回值，它们的类型应该相同：

```
// C++11
auto func1(int const i) -> int
{ return 2*i; }

// C++14
auto func2(int const i)
{ return 2*i; }
```

如前所述，auto 不保留 const/volatile 和引用限定符。这就给 auto 作为函数返回类型的占位符带来了问题。为了解释这一点，让我们考虑前面提到的 foo.get() 的例子。这一次，我们有一个名为 proxy_get() 的包装函数，它接受一个对 foo 的引用，调用

`get()` 并返回 `get()` 返回的值，这个值是一个 int&。然而，编译器会将 `proxy_get()` 的返回类型推导为 int，而不是 int&。试图将该值赋给一个 int& 会导致错误。

```
class foo
{
   int x_;
public:
   foo(int const x = 0) :x_{ x } {}
   int& get() { return x_; }
};

auto proxy_get(foo& f) { return f.get(); }
auto f = foo{ 42 };
auto& x = proxy_get(f); // cannot convert from  'int' to 'int &'
```

要解决这个问题，我们需要实际返回 `auto&`。然而，在模板和完美转发返回类型而不知道它是值还是引用的情况下，这是一个问题。在 C++14 中解决这个问题的方法是使用 `decltype(auto)`，它将正确推导出类型：

```
decltype(auto) proxy_get(foo& f)
{ return f.get(); }

auto f = foo{ 42 };
decltype(auto) x = proxy_get(f);
```

decltype 关键字用于检查实体或表达式的声明类型。当使用标准记法声明类型很麻烦或者根本不可能声明时，它大多很有用。例如声明 lambda 类型和依赖模板参数的类型等情况。

auto 可以使用的最后一个重要情况是与 lambda 表达式一起使用。从 C++14 开始，lambda 表达式的返回类型和参数类型都可以是 auto。这样的 lambda 表达式被称为泛型 lambda 表达式，因为由 lambda
表达式定义的闭包类型有一个模板化的调用操作符。下面展示了一个泛型 lambda
表达式，它接受两个 auto 参数，并返回对实际类型应用 operator+ 的结果：

```
auto ladd = [] (auto const a, auto const b) { return a + b; };
struct
{
  template<typename T, typename U>
  auto operator () (T const a, U const b) const { return a+b; }
} L;
```

这个 lambda 表达式可以用于对任何定义了 operator+
的东西进行相加操作，如下代码片段所示：

```
auto i = ladd(40, 2); // 42
auto s = ladd("forty"s, "two"s); // "fortytwo"s
```

在这个示例中，我们使用 ladd lambda 表达式来对两个整数相加以及对 std::string 对象（使用 C++14 的用户定义字面量操作符 "s"）进行连接操作。