在传统 C 和 C++中，参数的类型都必须明确定义，这其实对我们快速进行编码没有任何帮助，尤其是当我们面对一大堆复杂的模板类型时，必须明确的指出变量的类型才能进行后续的编码，这不仅拖慢我们的开发效率，也让代码变得又臭又长。

C++11 引入了 `auto` 和 `decltype` 这两个关键字实现了类型推导，让编译器来操心变量的类型。这使得 C++ 也具有了和其他现代编程语言一样，某种意义上提供了无需操心变量类型的使用习惯。

auto

`auto` 在很早以前就已经进入了 C++，但是他始终作为一个存储类型的指示符存在，与 `register` 并存。在传统 C++ 中，如果一个变量没有声明为 `register` 变量，将自动被视为一个 `auto` 变量。而随着 `register` 被弃用，对 `auto` 的语义变更也就非常自然了。

使用 `auto` 进行类型推导的一个最为常见而且显著的例子就是迭代器。在以前我们需要这样来书写一个迭代器：

```
for(vector<int>::const_iterator itr = vec.cbegin(); itr != vec.cend(); ++itr)
```

而有了 `auto` 之后可以：

```
// 由于 cbegin() 将返回 vector<int>::const_iterator
// 所以 itr 也应该是 vector<int>::const_iterator 类型
for(auto itr = vec.cbegin(); itr != vec.cend(); ++itr);
```

```
一些其他的常见用法：
auto i = 5; // i 被推导为 int
auto arr = new auto(10) // arr 被推导为 int *
```

注意：`auto` 不能用于函数传参，因此下面的做法是无法通过编译的（考虑重载的问题，我们应该使用模板）：

```
int add(auto x, auto y);
```

此外，`auto` 还不能用于推导数组类型：

```
#include <iostream>

int main() {
 auto i = 5;

 int arr[10] = {0};
 auto auto_arr = arr;
```

```
 auto auto_arr2[10] = arr;

 return 0;
}
```

decltype

`decltype` 关键字是为了解决 auto 关键字只能对变量进行类型推导的缺陷而出现的。它的用法和 `sizeof` 很相似：

`decltype(表达式)`

有时候，我们可能需要计算某个表达式的类型，例如：

```
auto x = 1;
auto y = 2;
decltype(x+y) z;
```

尾返回类型、auto 与 decltype 配合

你可能会思考，`auto` 能不能用于推导函数的返回类型。考虑这样一个例子加法函数的例子，在传统 C++ 中我们必须这么写：

```
template<typename R, typename T, typename U>
R add(T x, U y) {
 return x+y
}
```

> typename 和 class 在模板中没有区别，在 typename 这个关键字出现之前，都是使用 class 来定义模板参数的

这样的代码其实变得很丑陋，因为程序员在使用这个模板函数的时候，必须明确指出返回类型。但事实上我们并不知道 `add()` 这个函数会做什么样的操作，获得一个什么样的返回类型。

在 C++11 中这个问题得到解决。虽然你可能马上回反应出来使用 `decltype` 推导 `x+y` 的类型，写出这样的代码：

```
decltype(x+y) add(T x, U y);
```

但事实上这样的写法并不能通过编译。这是因为在编译器读到 decltype(x+y) 时，`x` 和 `y` 尚未被定义。为了解决这个问题，C++11 还引入了一个叫做尾返回类型（ trailing return type ），利用 auto 关键字将返回类型后置：

```
template<typename T, typename U>
auto add(T x, U y) -> decltype(x+y) {
 return x+y;
}
```

令人欣慰的是从 C++14 开始是可以直接让普通函数具备返回值推导，因此下面的写法变得合法：

```
template<typename T, typename U>
auto add(T x, U y) {
 return x+y;
}
```