3. 语言新特性

3. 语言新特性

C++语言内核与标准库，二者的标准化通常并行进行。这么一来，标准库可受益于语言的强化，而语言可受益于标准库实现过程中获得的经验。这导致C++标准库往往会用上若干“很可能在前一版标准中并不存在”的语言特性。

3.1 C++11 语言新特性

3.1.1 微小但重要的语法提升

nullptr 和 std::nullptr_t

C++11允许你使用nullptr取代0或NULL，用来表示一个pointer（指针）指向所谓的no value （此不同于拥有一个不确定值）。这个新特性能够帮助你在“null pointer被解释为一个整数值”时避免误解。例如：

```
1  void f(int);
2  void f(void*);
3  f(0);          // calls f(int)
4  f(NULL);       // calls f(int) if NULL is 0, ambiguous otherwise
5  f(nullptr);    // calls f(void*)
```

nullptr是个新关键字。它被自动转换为各种pointer类型，但不会被转换为任何整数类型。它拥有类型std::nullptr_t，定义于＜cstddef＞（5.8.1节），所以你现在甚至可以重载函数令它们接受null pointer。注意，std::nullptr_t被视为一个基础类型 （5.4.2节）。

3.1.2 auto 完成类型自动推导

C++11允许你声明一个变量或对象（object）而不需要指明其类型，只需说它是auto。以auto声明的变量，其类型会根据其初值被自动推导出来，因此一定需要一个初始化操作。

3.1.3 一致性初始化（Uniform Initialization）与初值列（Initializer List）

是初学者，很容易被这个问题混淆：如何初始化一个变量或对象。初始化可因为小括号、大括号或赋值操作符（assignment operator）的出现而发生。

C++11引入了“一致性初始化”（uniform initialization）概念，意思是面对任何初始化动作，你可以使用相同语法，也就是使用大括号。

```
int values[] { 1, 2, 3 };
std::vector<int> v { 2, 3, 5, 7, 11, 13, 17 };
std::vector<std::string> cities {
    "Berlin", "New York", "London", "Braunschweig", "Cairo", "Cologne"
};
std::complex<double> c{4.0,3.0}; // equivalent to c(4.0,3.0)
```

初值列（initializer list）会强行 value initialization ，意思是即使某个local变量属于某种基础类型（通常会有不明确的初值）也会被初始化为0（或nullptr——如果它是个pointer）：

```
int i;      // i has undefined value
int j{};    // j is initialized by 0
int* p;     // p has undefined value
int* q{};   // q is initialized by nullptr
```

注意，窄化（narrowing，精度降低或造成数值变动）对大括号而言是不可成立的。

```
int x1(5.3);      // OK, but OUCH: x1 becomes 5
int x2 = 5.3;     // OK, but OUCH: x2 becomes 5
int x3{5.0};      // ERROR: narrowing
int x4 = {5.3};   // ERROR: narrowing
char c1{7};       // OK: even though 7 is an int, this is not narrowing
char c2{99999};  // ERROR: narrowing (if 99999 doesn't fit into a char)
std::vector v1 { 1, 2, 4, 5 };  // OK
std::vector v2 { 1, 2.3, 4, 5.6 }; //ERROR: narrowing doubles to ints
```

为了检查是否窄化，如果当前值在编译期可获得的话，它也可能被考虑。判断是否窄化转换，C++11用以避免许多不兼容的做法是，依赖初值设定的实际值不只是依赖类型。如果一个值可被 target type 精确表述，其间的转换就不算窄化。浮点数转换至整数，永远是一种窄化——即使7.0转为7。

为了支持 “用户自定义类型之初值列” （initializer lists for user-defined types）概念，C++11 提供了 class template std::initializer_list<>，用来支持以一系列值进行初始化。

```
 void print (std::initializer_list vals)
```

```
 {
     for (auto p=vals.begin(); p!=vals.end(); ++p) { // process a list of values
         std::cout << *p << "\n";
     }
 }

 print ({12,3,5,7,11,13,17}); // pass a list of values to print()
```

当 “指明实参个数” 和 “指明一个初值列” 的构造函数（ctor）同时存在，带有初值列的版本优先。

explicit构造函数如果接受的是个初值列，会失去“初值列带有0个、1个或多个初值”的隐式转换能力。

3.1.4 Range-Based for 循环

基于范围的 for 循环可以逐一迭代某个给定区间、数组、集合内的每个元素，也称 for each 循环。

```
for (decl : coll) {
    statement
}
```

如果要将vector vec的每个元素elem乘以3，可以这么做：

```
std::vector<double> vec;
...
for (auto& elem : vec) {
    elem *= 3;
}
```

声明 elem 为一个引用很重要。不然，for循环的语句会作用在元素的一份 local copy 上。

为了避免调用每个元素的复制构造函数和析构函数，通常应该声明当前元素为一个 const reference。 “打印某集合内所有元素” 的泛型函数应该写成这样：

```
template <typename T>
void printElements(const T& coll)
{
    for (const auto& elem : coll) {
        std::cout << elem << std::endl;
```

```
    }
}
```

Range-based for 语句等同于：

```
{
    for(auto _pos=coll.begin(); _pos != coll.end(); ++_pos) {
        const auto& elem = *_pos;
        std::cout << elem << std::endl;
    }
}
```

元素在for循环中被初始化为decl，不得有任何显式类型转换（explicit type conversion）。下面的代码无法通过编译：

```
class C
{
public:
    explicit C(const std::string& s)
    ...
};

std::vector<std::string> vs;
for(const C& elem : vs) {   //ERROR
    std::cout << elem << std::endl;
}
```

3.1.5 Move 语义和 Rvalue Reference

C++11 的一个最重要的特性就是，支持 move semantic（语义）。这项特性用以避免非必要拷贝和临时对象。

```
void createAndInsert(std::set<X>& coll)
{
    X x;
    ...
    coll.insert(x);
}
```

在这里将新对象插入集合中，后者提供一个成员函数可为传入的元素建立一份内部拷贝（internal copy）：

```
coll.insert(x+x);    // inserts copy of temporary rvalue
coll.insert(x);      // inserts copy of x(although x is not used any longer)
```

上面被传入值不再被调用者使用，那么 coll 内部就无须为它建立一份 copy 且 “以某种方式move其内容到新建元素中”。当 x 复制成本高昂时，例如它是个巨大的 string 集合，这会带来很大的性能提升。

C++11 开始，我们可以通过自行指明 move。虽然编译器自身也有可能找出这个情况，允许程序员执行这项工作可使这个特性被用于逻辑上任何适当之处。先前的代码只需简单改成这样：

```
coll.insert(x+x);     // moves (or copies) contents of temporary rvalue
coll.insert(std::move(x)); // moves (or copies) contents of x into coll
```

声明在 <utility> 的 std::move()，x 可被 moved 而不再被 copied。但是 std::move() 本身不做任何 moving 工作，它只是将其实参转成一个 rvalue reference，一种被声明为 X&& 的类型。这种类型表示 rvalue（匿名临时对象只能出现在赋值操作的右侧）可被改变内容。这份 contract 的含义是，这是个不再被需要的（临时）对象，所以你可以偷取其内容或资源。

现在，集合可以提供 insert() 的重载版本，它处理右值引用。

```
namespace std {
    template <typename T, ...> class set {
        public:
        ... insert(const T& x);    //for lvalues:copies the value
        ... insert(T&& x);         //for rvalues:moves the value
    };
}
```

现在可以优化右值引用的版本，实现窃取 x 的内容。要做到这点，需要 x 类型的帮助，只有 x 类型才能访问其内部。因此，可以使用 x 的内部数组和指针来初始化插入的元素。

3.1.6 新的字符串字面常量（String Literal）

Raw String Literals

允许定义字符序列，可以节省修饰特殊字符的符号。

```
//“两个反斜线和一个n” 的寻常字面量定义如下:
"\\\\n"

//也可定义为 raw string literal:
R"(\\n)"
```

Encoded String Literals

使用编码前缀（econding prefix），就可以为 string literal 定义一个特殊的字符编码。

3.1.7 关键字 noexcept

noexcept 用于指定函数不能抛出异常，或者不准备抛出异常。

```
void foo() noexcept;
```

如果 foo() 内部没有处理异常，因此，若 foo() 抛出异常，程序会终止，调用 std::terminate()，它默认情况下会调用 std::abort()。

noexcept 针对（空）异常规范的许多问题。

- 运行期检查（Runtime checking）：C++ 异常规范被检查是在运行期而非编译期，所以它无法保证每个异常都被处理。runtime failure mode（调用 std::unexpected()）已经不能恢复。
- 运行期开销（Runtime overhead）：运行期检查会令编译器产出额外代码且妨碍优化。
- 无法用于泛型代码（Unusable in generic code）：泛型代码往往不知道哪一类异常可能被 “模板实参的操作” 的函数抛出，所以我们无法写出正确的异常规范。

实际上只有两种形式的异常抛出保证是有用的：一是操作可能抛出异常（任何异常），二是操作绝不会抛出任何异常。前者以完全省略异常规范（exception-specification）。后者表现为 throw()，但基于性能考虑很少如此。

由于 noexcept 不需要栈展开（stack unwinding），可以表现 nothrow 而不需要额外开销。C++11起不再鼓励使用异常规范。

还可以指明在某种条件下函数不抛出异常。例如，对于任意类型 Type，全局的 swap() 通常被定义如下：

```
void swap(Type& x, Type& y) noexcept(noexcept(x.swap(y)))
{
    x.swap(y);
}
```

在 noexcept(...) 中可以指定一个 Boolean 条件，若符合就不抛出异常。

再举个例子，接受 value pair 为实参的 move assignment 操作符声明如下；

```
pair& operator=(pair&& p)
        noexcept(is_nothrow_move_assignable<T1>::value &&
                 is_nothrow_move_assignable<T2>::value);
```

这里用到了 is_nothrow_move_assignable type trait，用来检查针对被传入的类型，是否可能存在一个不抛出异常的 move assignment 操作符。

noexcept 以下面保守方式引入程序库中：

3.1.8 关键字 constexpr

constexpr 用于编译时计算表达式。

3.1.9 Template 特性

Variadic Template（可变参数模版）

Alias Template

3.1.10 Lambda

3.1.11 关键字 decltype

decltype 可让编译器找出表达式类型。typeof 的特性体现。

decltype 应用之一是声明返回类型，另一用途是在 metaprogramming（5.4.1节）或用来传递一个 lambda 类型（10.3.4节）

3.1.12 新的函数声明语法（New Function Declaration Syntax）

C++11 可以将一个函数的返回类型转而声明于参数列之后：

```
1 template<typename T1, typename T2>
2 auto add(T1 x, T2 y) -> decltype(x+y);
```

这种写法所采用的语法，和为lambd声明返回类型是一样的（3.1.10节）。

3.1.13 带域的（Scoped）Enumeration

3.1.14 新的基础类型（New Fundamental Data Type）

3.2 虽旧犹新的语言特性

3.2.1 基础类型的明确初始化（Explicit Initialization for Fundamental Type）

3.2.2 main() 定义式