翻书

C++11 带来的新特性 （2）—— 统一初始化（Uniform Initialization）

1 统一初始化（Uniform Initialization）

在C++ 11之前，所有对象的初始化方式是不同的，经常让写代码的我们感到困惑。C++ 11努力创造一个统一的初始化方式。
其语法是使用{}和std::initializer_list，先看示例。

```
int values[]{ 1, 2, 3 };
std::vector<int> v{ 2, 3, 6, 7 };
std::vector<std::string> cities{
    "Berlin", "New York", "London",  "Braunschweig"
};
std::comples<double> c{4.0, 3.0}; //等价于 c(4.0, 3.0)
auto ar = { 1, 2, 3 };  // ar 是一个std::initializer_list<int>类型

std::vector<int> v = { 1, 2, 3 };
std::list<int> l = { 1, 2, 3 };
std::set<int> s = { 1, 2, 3 };
std::map<int, std::string> m = { {1, "a"}, {2, "b"} };
```

2 原理

针对形如"{ 1, 2, 3 }"的参数列表,系统会首先自动调用参数初始化(value initialization)，将其转换成一个std::initializer_list，它将用于对变量（可能是简单类型，或者对象类型）初始化。比如"{ 1, 2, 3 }"会首先生成一个std::initializer_list，然后用于生成一个int数组values。c{4.0, 3.0}会生成一个std::initializer_list,然后调用std::comples类的构造函数std::comples(double,double)。
我们通过一个例子来分析具体细节：

```
std::vector<int> v{ 2, 3, 6, 7 };
```

- 首先，将参数列表{ 2, 3, 6, 7 }转换成std::initializer_list。
  从stl源码中可以看出initializer_list的带参数构造函数是个私有函数，它只能由编译器调用。

```
private:
  iterator          _M_array;
  size_type         _M_len;

  // The compiler can call a private constructor.
  constexpr initializer_list(const_iterator __a, size_type __l)
  : _M_array(__a), _M_len(__l) { }
```

- 其次，使用std::initializer_list对象来初始化std::vector类的构造函数。下面是构造函数源码。

```
  vector(initializer_list<value_type> __l,
     const allocator_type& __a = allocator_type())
  : _Base(__a)
  {
_M_range_initialize(__l.begin(), __l.end(),
            random_access_iterator_tag());
  }
```

3 未赋值的初始化

如果使用了std::initializer_list，但是没有指定参数值，结果会怎样？直接看示例。

```
int i;      //i值未定义
int j{};    //j=0
int *p;     //p值未定义
int *q{};   //q=nullptr
```

4 在构造函数中显示使用std::initializer_list

我们可以在构造函数中主动使用std::initializer_list，这时外部调用{}初始化时，会优先调用包含std::initializer_list参数的构造函数。请看下例子。

```
class P
{
public:
    P(int, int){std::cout<<"call P::P(int,int)"<<std::endl;}
    P(std::initializer_list<int>){
        std::cout<<"call P::P(initializer_list)"<<std::endl;
    }
};
P p(77,5);     // call P::P(int,int)
P q{77,5};     // call P::P(initializer_list)
P r{77,5,42};  // call P::P(initializer_list)
P s = {77, 5}; // call P::P(initializer_list)
```

5 拒绝隐式调用构造函数

我们知道，C++会先使用{}中的参数生成一个std::initializer_list对象，然后调用赋值对象的构造函数。
有一种特殊情形，我们如果希望对象的某个构造函数必须要被显示调用，如何做到呢？向其中添加一个explicit关键字。在stl库中出现大量的这种使用方式。请看下例：

```
class P
{
  public:
    P(int a, int b){...}
    explicit P(int a, int b, int c){...}
};

P x(77,5);    //OK
P y{77,5};    //OK
P z{77,5,42}; //OK
p v = {77,5}; //OK,(implicit type conversion allowed)
P w = {77,5,42};//Error,必须要显示调用该构造函数

void fp(const P&);

fp({47,11});    //OK
fp({47,11,3});  //Error
fp(P{47,11});   //OK
fp(P{47,11,3}); //OK
```

6 局限 —— Narrowing Initializations

统一初始化用起来很舒爽，那它有什么局限呢?
有，在一种场景下无法使用，那就是Narrowing Initializations。
Narrowing Initializations，我翻译为“精度截断”。比如float转换为int，double转换为float。统一初始化，完全不允许精度阶段的发生，更进一步，要求参数列表中的所有参数的精度一样。请看以下示例。

```
int x1(5.3);    //OK, x1 = 5.3
int x3{5.0};    //Error
int x4 = {5.3}; //Error
char c1{7};     //OK
char c2{99999}; //Error
std::vector<int> v1{ 1, 2, 4, 5};    //OK
std::vector<int> v2{ 1, 2,3, 4, 5.6};//Error
```

但是如果实际工程允许精度截断的发生，那么我们应该怎么完成初始化。可以使用()来完成初始化，它会调用赋值操作或者相应的构造函数。

```
int x3{5.0};    //Error
int x2(5.2);    //OK, x2 = 5
```

分类：C && C++

翻书
关注 - 3
粉丝 - 26
推荐博客
+加关注

« 上一篇： C++11 带来的新特性 （1）
» 下一篇： C++11 带来的新特性 （3）—— 关键字noexcept

posted @ 2018-11-26 12:28 翻书 阅读(3145) 评论(0) 编辑 收藏 举报