常用链接

联系方式:me115在126.com

木书架网，专注于互联网阅读分享，关注以下领域：互联网，程序员，时间管理及个人提升。
http://www.me115.com

我的开源书籍

《Linux工具快速教程》
在线阅读 GitHub地址

《图说设计模式》
在线阅读 GitHub地址

C++11带来的优雅语法

内容目录：

内容目录：

自动类型推导 auto
萃取类型 decltype
返回类型后置语法 Trailing return type
空指针标识 nullptr
区间迭代 range-based for loop
去除右尖括号的蹩脚语法 right angle brackets
lambda表达式的引入
原生字符串 Raw string literals
非成员begin()和end()
初始化列表及统一初始化方法 Initializer lists
参考

C++11带来的优雅语法

自动类型推导 auto

auto的自动类型推导，用于从初始化表达式中推断出变量的数据类型。通过auto的自动类型推导，可以简化我们的编程工作;
auto是在编译时对变量进行了类型推导，所以不会对程序的运行效率造成不良影响;
另外，似乎auto也并不会影响编译速度，因为编译时本来也要右侧推导然后判断与左侧是否匹配。

```
auto a; // 错误，auto是通过初始化表达式进行类型推导，如果没有初始化表达式，就无法确定a
的类型
auto i = 1;
auto d = 1.0;
auto str = "Hello World";
auto ch = 'A';
```

auto对引用的推导默认为值类型，可以指定引用修饰符设置为引用:

```
int x = 5;
int & y = x;
```

随笔档案


```
auot  z = y ;// z 为int
auto & z = y; // z的类型为 int&
```

对指针的推导默认为指针类型，当然，也可以指定*修饰符（效果一样）：

```
int  *px = &x;
auto py = px;
auto*py = px;
```

推导常量

```
const int *px = &x;
auto py = px; //py的类型为 const int *
const auto py = px ; //py的类型为const int *
```

萃取类型 decltype

decltype实际上有点像auto的反函数，使用auto可以用来声明一个指定类型的变量，而decltype可以通过一个变量（或表达式）得到类型；

```
#include <vector>
int main() {
    int x = 5;
    decltype(x) y = x; //等于 auto y = x;
    const std::vector<int> v(1);
    auto a = v[0];        // a has type int
    decltype(v[1]) b = 1; // b has type const int&, the return type of
                          //   std::vector<int>::operator[](size_type) const
    auto c = 0;           // c has type int
    auto d = c;           // d has type int
    decltype(c) e;        // e has type int, the type of the entity named by c
    decltype((c)) f = c;  // f has type int&, because (c) is an lvalue
    decltype(0) g;        // g has type int, because 0 is an rvalue
}
```

有没有联想到STL中的萃取器？写模版时有了这个是不是会方便很多；

返回类型后置语法 Trailing return type

C++11支持返回值后置
例如：

```
int adding_func(int lhs, int rhs);
```

可以写为：

```
auto adding_func(int lhs, int rhs) -> int
```

auto用于占位符，真正的返回值在后面定义；
这样的语法用于在编译时返回类型还不确定的场合；
比如有模版的场合中，两个类型相加的最终类型只有运行时才能确定：

```
template<class Lhs, class Rhs>
auto adding_func(const Lhs &lhs, const Rhs &rhs) -> decltype(lhs+rhs)
{return lhs + rhs;}
cout << adding_func<double,int>(dv,iv) << endl;
```

auto用于占位符，真正的返回值类型在程序运行中，函数返回时才确定；

不用auto占位符，直接使用decltype推导类型：

```
decltype(lhs+rhs) adding_func(const Lhs &lhs, const Rhs &rhs)
```

这样写，编译器无法通过，因为模版参数lhs和rhs在编译期间还未声明；
当然，这样写可以编译通过：

```
decltype( (*(Lhs*)0) + (*(Rhs*)0) ) adding_func(const Lhs &lhs, const Rhs &rhs)
```

但这种形式实在是不直观，不如auto占位符方式直观易懂；

空指针标识 nullptr

空指针标识(nullptr)（其本质是一个内置的常量）是一个表示空指针的标识，它不是一个整数。这里应该与我们常用的NULL宏相区别，虽然它们都是用来表示空置针，但NULL只是一个定义为常整数0的宏，而nullptr是C++11的一个关键字，一个内建的标识符。
nullptr和任何指针类型以及类成员指针类型的空值之间可以发生隐式类型转换，同样也可以隐式转换为bool型（取值为false）。但是不存在到整形的隐式类型转换。
有了nullptr，可以解决原来C++中NULL的二义性问题；

```
voidF(int a){
    cout<<a<<endl;
}
voidF(int*p){
    assert(p != NULL);
    cout<< p <<endl;
}

int main(){
    int*p = nullptr;
    int*q = NULL;
    bool equal = ( p == q ); // equal的值为true，说明p和q都是空指针
    int a = nullptr; // 编译失败，nullptr不能转型为int
    F(0); // 在C++98中编译失败，有二义性；在C++11中调用F（int）
    F(nullptr);

    return 0;
}
```

区间迭代 range-based for loop

C++11扩展了for的语法，终于支持区间迭代，可以便捷的迭代一个容器的内的元素；

```
int my_array[5] = {1, 2, 3, 4, 5};
// double the value of each element in my_array:
for (int &x : my_array) {
    x *= 2;
}
```

当然，这时候使用auto会更简单；

```
for (auto &x : my_array) {
    x *= 2;
}
```

如果有更为复杂的场景,使用auto的优势立刻体现出来：

```
map<string,int> map;
map.insert<make_pair<>>("ss",1);
for(auto &x : my_map)
{
   cout << x.first << "/" << x.second;
}
```

去除右尖括号的蹩脚语法 right angle brackets

在C++98标准中，如果写一个含有其他模板类型的模板：

```
vector<vector<int> > vector_of_int_vectors;
```

你必须在结束的两个’>‘之间添加空格。这不仅烦人，而且当你写成>>而没有空格时，你将得到困惑和误导的编译错误信息。产生这种行为的原因是C++词法分析的最大匹配原则（maximal munch rule)。一个好消息是从今往后，你再也不用担心了：

```
vector<vector<int>> vector_of_int_vectors;
```

在C++98中，这是一个语法错误，因为两个右角括号(‘>’)之间没有空格（译注：因此，编译器会将它分析为”>>”操作符）。C++0x可以正确地分辨出这是两个右角括号(‘>’)，是两个模板参数列表的结尾。

为什么之前这会是一个问题呢？一般地，一个编译器前端会按照“分析/阶段”模型进行组织。简要描述如下：

词法分析（从字符中构造token）
语法分析（检查语法）
类型检查（确定名称和表达式的类型）
这些阶段在理论上，甚至在某些实际应用中，都是严格独立的。所以，词法分析器会认为”>>”是一个完整的token（通常意味着右移操作符或是输入），而无法理解它的实际意义（译注：即在具体的上下文环境下，某一个符号的具体意义）。特别地，它无法理解模板或内置模板参数列表。然而，为了使上述示例“正确”，这三个阶段必须进行某种形式的交互、配合。解决这个问题的最关键的点在于，每一个C++ 编译器已完整理解整个问题（译注：对整个问题进行了全部的词法分析、符号分析及类型检测，然后分析各个阶段的正确性），从而给出令人满意的错误消息。

lambda表达式的引入

对于为标准库算法写函数/函数对象（function object）这个事儿大家已经抱怨很久了（例如Cmp）。特别是在C++98标准中，这会令人更加痛苦，因为无法定义一个局部的函数对象。

首先，我们需要在我们实现的逻辑作用域（一般是函数或类）外部定义比较用的函数或函数对象，然后，才能使用：

```
bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;

int main()
{
    int myints[] = {32,71,12,45,26,80,53,33};
    std::vector<int> myvector (myints, myints+8);
     // using function as comp
    std::sort (myvector.begin(), myvector.end(), myfunction);
      // using function object as comp
      std::sort (myvector.begin(), myvector.end(), myobject);
}
```

不过现在好多了，lambda表达式允许用”inline”的方式来写函数了：

```
sort(myvector.begin(), myvector.end(), [](int i, int j) { return i< j; });
```

真是亲切！lambda的引入应该会增加大家对STL算法的使用频率；

原生字符串 Raw string literals

比如，你用标准regex库来写一个正则表达式，但正则表达式中的反斜杠’\’其实却是一个“转义(escape)”操作符(用于特殊字符)，这相当令人讨厌。考虑如何去写“由反斜杠隔开的两个词语”这样一个模式(\w\\w):

```
string s = "\\w\\\\\\w"; // 不直观、且容易出错
```

请注意，在正则表达式和普通C++字符串中，各自都需要使用连续两个反斜杠来表示反斜杠本身。然而，假如使用C++11的原生字符串，反斜杠本身仅需一个反斜杠就可以表示。因而，上述的例子简化为：

```
string s = R"(\w\\w)"; // ok
```

非成员begin()和end()

非成员begin()和end()函数。他们是新加入标准库的，除了能提高了代码一致性，还有助于更多地使用泛型编程。它们和所有的STL容器兼容。更重要的是，他们是可重载的。所以它们可以被扩展到支持任何类型。对C类型数组的重载已经包含在标准库中了。

在这个例子中我打印了一个数组然后查找它的第一个偶数元素。如果std::vector被替换成C类型数组。代码可能看起来是这样的：

```
int arr[] = {1,2,3};
std::for_each(&arr[0], &arr[0]+sizeof(arr)/sizeof(arr[0]), [](int n) {std::cout << n << std::endl;});

auto is_odd = [](int n) {return n%2==1;};
auto begin = &arr[0];
auto end = &arr[0]+sizeof(arr)/sizeof(arr[0]);
auto pos = std::find_if(begin, end, is_odd);
```

```
if(pos != end)
std::cout << *pos << std::endl;
```

如果使用非成员的begin()和end()来实现，就会是以下这样的：

```
int arr[] = {1,2,3};
std::for_each(std::begin(arr), std::end(arr), [](int n) {std::cout << n << std::endl;});

auto is_odd = [](int n) {return n%2==1;};
auto pos = std::find_if(std::begin(arr), std::end(arr), is_odd);
if(pos != std::end(arr))
std::cout << *pos << std::endl;
```

这基本上和使用std::vecto的代码是完全一样的。这就意味着我们可以写一个泛型函数处理所有支持begin()和end()的类型。

初始化列表及统一初始化方法 Initializer lists

在C++98中，对vector的多个初始化，我们需要这样：

```
int myints[] = { 10, 20, 30, 30, 20, 10, 10, 20 };
std::vector<int> myvector (myints, myints+8);
```

现在，我们可以这样：

```
std::vector<int> second ={10, 20, 30, 30, 20, 10, 10, 20};
```

初始化表有时可以像参数那样方便的使用。看下边这个例子（x,y,z是string变量，Nocase是一个大小写不敏感的比较函数）：
auto x = max({x,y,z},Nocase());

初始化列表不再仅限于数组。对于常见的map、string等，我们可以使用以下语法来进行初始化：

```
int arr[3]{1, 2, 3};
vector<int> iv{1, 2, 3};
map<int, string>  m{{1, "a"}, {2, "b"}};
string str{"Hello World"};
```

可以接受一个“{}列表”对变量进行初始化的机制实际上是通过一个可以接受参数类型为std::initializer_list的函数（通常为构造函数）来实现的。例如：

```
void f(initializer_list<int>);
f({1,2});
f({23,345,4567,56789});
f({});  // 以空列表为参数调用f()
f{1,2}; // 错误：缺少函数调用符号( )
years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});
```

初始化列表可以是任意长度，但必须是同质的（所有的元素必须属于某一模板类型T, 或可转化至T类型的)。

容器可以用如下方式来实现“初始化列表构造函数”：

```
template<class E> class vector {
    public:
        // 初始化列表构造函数
                vector (std::initializer_list<E> s)
        {
                // 预留出合适的容量
                  reserve(s.size());    //
              // 初始化所有元素
                uninitialized_copy(s.begin(), s.end(), elem);
         sz = s.size(); // 设置容器的size
        }
        // ... 其他部分保持不变 ...
    };
```

使用“{}初始化”时，直接构造与拷贝构造之间仍有细微差异，但不再像以前那样明显。例如，std::vector拥有一个参数类型为int的显式构造函数及一个带有初始化列表的构造函数：

```
vector<double> v1(7); // OK: v1有7个元素<br />
v1 = 9;         // Err: 无法将int转换为vector
```

```
vector<double> v2 = 9;    // Err: 无法将int转换为vector

void f(const vector<double>&);
f(9);              // Err: 无法将int转换为vector

vector<double> v1{7};     // OK: v1有一个元素，其值为7.0
v1 = {9};           // OK: v1有一个元素，其值为9.0
vector<double> v2 = {9};  // OK: v2有一个元素，其值为9.0
f({9});  // OK: f函数将以列表{9}为参数被调用

vector<vector<double>> vs = {
    vector<double>(10), // OK, 显式构造(10个元素，都是默认值0.0)
    vector<double>{10}, // OK：显式构造(1个元素，值为10.0)
        10          // Err ： vector的构造函数是显式的
};
```

函数可以将initializer_list作为一个不可变的序列进行读取。例如：

```
void f(initializer_list<int> args)
{
    for (auto p=args.begin(); p!=args.end(); ++p)
        cout << *p << "\n";
}
```

仅具有一个std::initializer_list的单参数构造函数被称为初始化列表构造函数。

标准库容器，string类型及正则表达式均具有初始化列表构造函数，以及（初始化列表）赋值函数等。初始化列表亦可作为一种“序列”以供“序列化for语句”使用。

参考

http://www.stroustrup.com/C++11FAQ.html
https://www.chenlq.net/books/cpp11-faq

Posted by: 大CC | 11SEP,2015
博客： blog.me115.com [订阅]
Github： 大CC

分类: C++编程

标签： c++11

« 上一篇：C++11在时空性能方面的改进
» 下一篇：异步和非阻塞

posted @ 2015-09-11 14:10 大CC 阅读(2031) 评论(0) 编辑 收藏

最新IT新闻:

· Capital One：只有windows手机份额达到10% 我们才会推出APP
· 苹果遭受零日漏洞攻击，手机黑客新时代来到
· 美国研制管状机器人能够分析污水中细菌指数
· 无人驾驶汽车如何应对道德困境 谷歌也没有答案
· 德国软件公司SAP 收购大数据创业公司Altiscale
» 更多新闻...

最新知识库文章:

· 程序猿媳妇儿注意事项
· 可是姑娘，你为什么要编程呢?
· 知其所以然（以算法学习为例）
· 如何给变量取个简短且无歧义的名字
· 编程的智慧
» 更多知识库文章...