Modern cpp 30 lectures — template and tuple

Thursday, December 24, 2020 8:39 AM

可变模板.
可以在模板参数里表达不定个数和类型的参数. 用途:
- 在通用工具模板中转发参数到另外一个函数.
- 在递归的模板中表达通用的情况.

转发用法.
以标准库里的 make_unique为例,定义:

```
template <typename T,
          typename... Args>
inline unique_ptr<T>
make_unique(Args&&... args)
{
  return unique_ptr<T>(
    new T(forward<Args>(args)...));
}
```

这样,它就可以把传递给自己的全部参数转发到模板参数类的构造函数上去.
注意:在这种情况下,通常会用std::forward,确保参数转发时仍然保持正确的左值或右值引用类型.
... 的含义:
- typename... Args 声明了一系列的类型 —— class... 或 typename... 表示后面的标识符代表了一系列的类型.
- Args&&... args 声明了一系列的形参args, 其类型是 Args&&.
- forward<Args>(args)... 会在编译时实际逐项展开 Args 和 args, 参数有多少项,展开后就是多少项.

例如,要在堆上传递一个 vector<int>,希望初始构造的大小为 100,每个元素为 1, 可以写:

```
make_unique<vector<int>>(100, 1);
```

模板实例化后会得到相当于下面的代码:

```
template <>
inline unique_ptr<vector<int>>
make_unique(int&& arg1, int&& arg2)
{
  return unique_ptr<vector<int>>(
    new vector<int>(
      forward<int>(arg1),
      forward<int>(arg2)));
}
```

递归用法

```
template <typename T>
constexpr auto sum(T x)
{
  return x;
}

template <typename T1, typename T2,
          typename... Targ>
constexpr auto sum(T1 x, T2 y,
                   Targ... args)
{
  return sum(x+y, args...);
}
```

如果sum得到的定义只有一个,就会用上面的重载,如果有两个或更多参数,编译器就会选下面的重载,执行一次加法,随后参数数量就少了一个,因而递归总会终止到上面那个重载,结束计算.
可以写下面的函数调用

```
auto result = sum(1, 2, 3.5, x);
```

模板会依次展开:

```
sum(1+2, 3.5, x)
sum(3+3.5, x)
sum(6.5+x)
6.5+x.
```

函数的组合:如果有函数 f 和函数 g,要得到函数的联用 g∘f,
$(g\circ f)(x) = g(f(x))$.
写出递归的终结情况,单个函数"组合":

```
template <typename F>
auto compose(F f)
{
  return [f](auto&&... x) {
    return f(
      forward<decltype(x)>(x)...);
  };
}
```

上面仅返回一个泛型 lambda,保证参数可转发到f,
下面是有正常组合的情况.

```
template <typename F,
          typename... Args>
auto compose(F f, Args... other)
{
  return [f,
          other...](auto&&... x) {
    return f(compose(other...)(
      forward<decltype(x)>(x)...));
  };
}
```

返回一个lambda表达式,用f捕捉第一个函数对象,用args...捕捉后面的函数对象. 然后把结果传到f里面.

验证 compose:写一个对输出范围中每一项都进行平方的函数对象:

```
auto square_list =
  [](auto&& container) {
    return fmap(
      [](int x) { return x*x; },
      container);
  };
```

fmap是函数式的接口,不修改参数的内容,结果完全在返回值中.

再写一个求和的函数对象.

```
auto sum_list =
  [](auto&& container) {
    return accumulate(
      container.begin(),
      container.end(), 0);
  };
```

那先平方再求和,可以定义为:

```
auto squared_sum = compose(sum_list, square_list);
```

验证:

```
vector v{1, 2, 3, 4, 5};
cout << squared_sum(v) << endl;
```

Output: 55.

tuple
上面写法的缺陷:被 compose的函数除了第一个,其它函数只能接收一个参数,
C++中,要通用地用一个变量表达多个值,就得用多元组— tuple 模板了.
tuple 算是 C++98 里pair类型的一般化,可表达任意多个固定数量,固定类型的值的组合.

```
#include <algorithm>
#include <iostream>
#include <string>
#include <tuple>
#include <vector>

using namespace std;

// 整数,字符串,字符串的三元组.
using num_tuple = tuple<int, string, string>;

ostream&
operator<<(ostream& os,
           const num_tuple& value)
{
  os << get<0>(value) << ','
     << get<1>(value) << ','
     << get<2>(value);
  return os;
}

int main()
{
  // 阿拉伯数字,英文,法文.
  vector<num_tuple> vn{
    {1, "one", "un"},
    {2, "two", "deux"},
    {3, "three", "trois"},
    {4, "four", "quatre"}};

  // 修改第0项的法文
  get<2>(vn[0]) = "une";

  // 按法文进行排序
  sort(vn.begin(), vn.end(),
       [](auto&& x, auto&& y) {
         return get<2>(x) < get<2>(y);
       });

  // 输出内容.
  for(auto&& value : vn) {
    cout << value << endl;
  }

  // 输出多元组项数
  constexpr auto size =
    tuple_size_v<num_tuple>;
  cout << "Tuple size is " << size << endl;
}
```

Output:

```
2, two, deux
4, four, quatre
3, three, trois
1, one, une
Tuple size is 3.
```

- tuple的成员数量由尖括号里写的类型数量决定
- 可用get函数对 tuple的内容进行读写.
- 可用 tuple_size_v 在编译期取得多元组里的项数.

如果用一个三项的 tuple 调用一个函数,可以写:

```
template <class F, class Tuple>
constexpr decltype(auto) apply(
  F&& f, Tuple&& t)
{
  return f(
    get<0>(forward<Tuple>(t)),
    get<1>(forward<Tuple>(t)),
    get<2>(forward<Tuple>(t)));
}
```

数值预算
我们希望快速地计算一串二进制中1比特的数量, E.g. 如果有十进制的 31和254,转换成二进制是 0001 1111 和 1111 1110,应得 5+7=12.

每个数字去数很慢,应该预先把每个字节的256种情况记录下来.可以让编译器在编译时帮我们计算数值.

```
constexpr int
count_bits(unsigned char value)
{
  if (value == 0) {
    return 0;
  } else {
    return (value & 1) +
           count_bits(value >> 1);
  }
}
```

为了避免把计算语句写256遍,我们定义一个模板,它的参数是一个序列,在初始化时这个模板会对参数里的每一项计算比特数,并放到数组成员里.

```
template <size_t... V>
struct bit_count_t {
  unsigned char
    count[sizeof...(V)] = {
      static_cast<unsigned char>(
        count_bits(V))...};
};
```

sizeof...(v)可获得参数的个数. 如果模板参数传0,1,2,3,结果里就会有含4项元素的数组,分别是对 0,1,2,3 的比特计数.

然后是用 make_index_sequence 展开计算:

```
template <size_t... V>
bit_count_t<V...> get_bit_count(
  index_sequence<V...>)
{
  return bit_count_t<V...>();
}

auto bit_count = get_bit_count(
  make_index_sequence<256>());
```

得到bit_count后,计算一个序列里的比特数就是查表了.