Modern cpp 30 lectures — compilation template

Tuesday, December 22, 2020 8:28 AM

编译期计算

C++模板是图灵完全的：使用C++模板，可以在编译期间模拟一个完整的图灵机，可以完成任何计算任务.

Eg.

```
template <int n>
struct factorial {
  static const int value =
    n * factorial<n-1>::value;
};

template <>
struct factorial<0> {
   static const int value = 1;
};
```

上面定义了一个递归的阶乘函数，

$0! = 1$

$n! = n \times (n-1)!$

可以使用static_assert，确保参数永远不会是负数.

```
template <int n>
struct factorial {
  static_assert(
     n >= 0,
    "Arg must be non-negative.");

  static const int value =
     n * factorial<n-1>::value;
};
```

要进行编译期编程，最重要的是把计算转变成类型推导：

Eg. 下面的模板可以代表条件语句.

```
template <bool bond,
          typename Then,
          typename Else >
struct If;
```

If 模板有3个参数，第一个是布尔值，后面两个代表不同分支计算的类型.

struct 声明规定了模板的形式.

```
template <typename Then,
          typename Else>
struct If<true, Then, Else> {
  typedef Then type;
};
```

第一个特化是真的情况，定义结果type为Then的分支.

```
template <typename Then,
          typename Else >
struct If< false, Then, Else> {
  typedef Else type;
};
```

第二个特化是假的情况，定义结果type为Else的分支.

循环：

```
template <bool Condition,
          typename Body>
struct While Loop;

template <typename Body>
struct WhileLoop <true, Body> {
   typedef typename WhileLoop<
     Body:: cond_value,
     typename Body::next_type >:: type
     type;
};

template <typename Body>
struct While Loop <false, Body> {
  typedef
    typename Body::res_type type;
};
```

约定循环体类型，必须提供一个静态数据成员cond_value，以及两个子类型的定义：res_type和next_type：

- cond_value代表循环条件（真/假）.
- res_type代表退出循环时的状态.
- next_type代表下面循环执行一次时的状态.

```
template <typename Body>
struct While {
  typedef typename WhileLoop<
    Body::cond_value, Body>:: type
    type;
};
```

通用地代表整数常数的模板：

```
template <class T, T v>
struct integral_constant {
  static const T value = v;
  typedef T value_type;
  typedef integral_constant type;
};
```

用循环模板来完成从1加到n的计算：

```
template < int result, int n>
struct SumLoop {
  static const bool cond_value = n != 0;
  static const int res_value = result;
  typedef integral_constant <
    int, res_value>
    res_type;
  typedef SumLoop <result+n, n-1>
    next_type;
};

template <int n>
struct Sum {
  typedef SumLoop <0, n> type;
};
```

使用 While<Sum<10>::type>::type::value 就可得从1加到10的结果. 相当于：

```
int result = 0;
while (n != 0) {
  result = result + n;
  n = n-1;
}
```

通用的fmap函数模板：

map-reduce 中，map，reduce 都来自函数式编程，在C++中用fmap代替map：

```
template <
  template <typename, typename>
  class OutContainer = vector,
  typename F, class R>
auto fmap(F&& f, R&& inputs).
{
  typedef decay_t <decltype (
    f(*inputs.begin()) >
    result_type;
  OutContainer <
    result_type,
    allocator <result_type>>
    result;
  for(auto&& item : inputs) {
    result.push_back (f(item));
  }
  return result;
}
```

缺省使用vector作为返回值的容器.

用decltype来获得用f来调用inputs元素的类型

用decay_t把获得的类型变成一个普通的值类型.

使用for循环遍历inputs.

不是右值引用，是转发引用.

验证功能：

```
vector<int> v{1,2,3,4,5};
int add_1 (int x)
{
  return x+1;
}
auto result = fmap(add_1, v);
```

fmap执行后，result是{2,3,4,5,6}.

Tips：

- 模板元编程的本质：把计算过程用编译期的类型推导和类型匹配表达出来.
- 下面的函数和模板等价：

```
int foo(int n)
{
  if (n== 2 || n == 3 || n== 5) {
    return 1;
  } else {
    return 2;
  }
}

template <int n>
struct Foo {
  typedef typename If <
    (n == 2 || n == 3 || n == 5),
    integral_constant <int, 1>,
    integral_constant<int, 2>>:: type
    type;
};
```

可以输出 foo(3)，或者 Foo<3>::type::value.

- 模板实际上就是一种展开.