Modern cpp 30 lectures — boost

Friday, December 25, 2020 10:40 AM

Boost.TypeIndex
如何精确地知道一个表达式或变量的类型.

```
#include <iostream>
#include <typeinfo>
#include <utility>
#include <vector>
#include <boost/type_index.hpp>

using namespace std;
using boost::typeindex::type_id;
using boost::typeindex::type_id_with_cvr;

int main()
{
  vector<int> v;
  auto it = v.cbegin();

  cout << "*** Using typeid \n";
  cout << typeid(const int).name()
       << endl;
  cout << typeid(v).name() << endl;
  cout << typeid(it).name() << endl;

  cout << "*** Using type_id \n";
  cout << type_id<const int>() << endl;
  cout << type_id<decltype(v)>()
       << endl;
  cout << type_id<decltype(it)>()
       << endl;
```

模板参数必须包含引用类型,所以用了decltype((v)),而不是decl-type(v).

```
  cout << "*** Using "
       " type_id_with_cvr \n";
  cout << type_id_with_cvr<const int>() << endl;
  cout << type_id_with_cvr<decltype((v))>() << endl;
  cout << type_id_with_cvr<decltype(move((v)))>() << endl;
  cout << type_id_with_cvr<decltype((it))>() << endl;
}
```

上面代码展示了标准的typeid和Boost的type_id和type_id_with_cvr的使用. 区别:

- typeid是标准C++的关键字,可以应用到变量或类型上,返回一个std::type_info.
- type_id是Boost的函数模板,必须提供类型作为模板参数,所以对于表达式和变量需要使用decltype.
- type_id_with_cvr和type_id相似.

Boost.Core
Core里面提供了一些通用的工具, Eg.

- addressof 获得对象的地址.
- enable_if
- is_same, 判断两个类型是否相同. C++11开始在<type_traits>中定义.
- ref.

boost::noncopyable
提供了一种非常简单也很直白地把类声明为不可拷贝的方式. Eg.

```
#include <boost/core/noncopyable.hpp>
class shape_wrapper
  : private boost::noncopyable {
  ...
};
```

当然,也可以自己把拷贝构造和拷贝赋值函数声明成 = delete.

boost::swap
标准做法:

```
{
  using std::swap;
  swap(lhs, rhs);
}
```

需要在某个小作用域里引入std::swap,然后让编译器在看得到std::swap的情况下编译swap命令,
根据ADL,如果在被交换的对象所属类型的名空间下有swap函数,
那个函数会被优先使用,否则会用std::swap.
使用boost,可以一行搞定:

```
#include <boost/core/swap.hpp>
boost::swap(lhs, rhs);
```

Boost.Conversion
可满足常用类型的转换.

```
#include <iostream>
#include <stdexcept>
#include <string>
#include <boost/lexical_cast.hpp>

using namespace std;
using boost::bad_lexical_cast;
using boost::lexical_cast;

int main()
{
  //整数到字符串的转换.
  int d = 42;
  auto d_str = lexical_cast<string>(d);
  cout << d_str << endl;

  //字符串到浮点数的转换.
  auto f = lexical_cast<float>(d_str) / 4.0;
  cout << f << endl;

  //测试 lexical_cast 的转换异常.
  try {
    int t = lexical_cast<int>("x");
    cout << t << endl;
  }
   catch(bad_lexical_cast& e) {
     cout << e.what() << endl;
   }

   //测试标准库stoi的转换异常.
   try {
     int t = std::stoi("x");
     cout << t << endl;
   }
   catch(invalid_argument& e) {
     cout << e.what() << endl;
   }
}
```

Boost.Program_options
处理命令行参数.

```
#include <iostream>
#include <string>
#include <stdlib.h>
#include <boost/program_options.hpp>

namespace po = boost::program_options;
using std::cout;
using std::endl;
using std::string;

string locale;
string lang;
 int width = 72;
bool keep_indent = false;
bool verbose = false;

int main(int argc, char* argv[])
{
  po::options_description desc(
    "Usage: breaktext [OPTION]..."
    "<Input File> [Output File] \n"
    "\n"
    "Avaiable options");

  desc.add_options()
    ("locale, L",
      po::value<string>(&locale),
     "locale of the console (system locale by default)")

    ("lang, l",
      po::value<string>(&lang),
     "language of input (assume no language by default)")

   ("width, w",
     po::value<int>(&width),
   "width of output text (72 by default)")

   ("help, h", "Show this message and exit")

   (",i",
    po::bool_switch(&keep_indent),
   "keep space indentation")

   (",v",
    po::bool_switch(&verbose),
   "Be verbose");

  po::variables_map vm;
  try{
    po::store(
      po::parse_command_line(
        argc, argv, desc),
      vm);
  }
  catch(po::error& e) {
    cout << e.what() << endl;
    exit(1);
  }
  vm.notify();

  if(vm.count("help")) {
    cout << desc << endl;
    exit(1);
  }
}
```

- options_description是基本的选项描述对象的类型,构造时我们给出选项的基本描述.
- variables_map变量映射表,用来存储对命令行的扫描结果,继承了标准的std::map.
- notify成员函数用来把变量映射表的内容实际传送到选项值描述里提供的那些变量里
- count成员函数继承自std::map.