语言新特性

模板表达式中的空格

在C++11之前,多层模板参数在嵌套时,最后的两个`>`之间要加一个空格,以避免和`>>`运算符混淆;C++11之后就不需要这样做了

```
vector<list<int> >;     // 左右版本均能通过编译
vector<list<int>>;      // C++11之后能通过编译
```

`nullptr`和`std::nullptr_t`

C++11之后允许将变量`nullptr`赋值给未指向任何对象的指针(C++11之前使用`0`或`NULL`).这个新特性可以避免将空指针转为整数`0`带来的错误.

```
// 函数f的两个重载版本
void f(int);
void f(void*);

// 调用函数
f(0);       // 调用 f(int)
f(NULL);    // 若NULL被定义为0,则会调用 f(int),产生错误
f(nullptr); // 调用 f(void*)
```

`nullptr`可以被自动转换为任意指针类型,但不会被转换为整型数.`nullptr`的类型为`std::nullptr_t`,定义在头文件`cstddef`中.

```
#if defined(_ cplusplus) & cplusplus >= 201103L
#ifndef_ GXX NULLPTR T
#define_ GXX NULLPTR T
    typedef decltype(nullptr) nullptr_t;
```

使用`auto`自动推断类型

C++11之后允许使用`auto`定义变量二不需要显式指定其类型,其类型由编译器自动推断得到.

```
auto i= 42;         // i has type int
double f();
auto d= f();        // d has type double
```

`auto`尤其适用于变量类型名特别长或难以写出的情况,如容器迭代器和lambda函数.

```
vector<string> v;

auto pos = v.begin();       // pos 的类型为 vector<string>::iterator
auto l= [](intx)-> bool {   // l 的类型为一个接收int参数返回int变量的lambda函数
    // ...
};
```

uniform initialization

在C++11之前,变量的初始化方式多种多样,有可能是有括号`()`,花括号`{}`和赋值运算符`=`.因此C++11引入了一种uniform initialization机制,上述用到的初始化方式都可以用一个花括号代替.

```
int values[]{1, 2, 3};
vector<int> v{2, 3, 5, 7, 11, 13, 17};
vector<string> cities{"Berlin", "New York", "London", "Braunschweig" "Cairo",
"Cologne"};
complex<double> c{4.0, 3.0};    // 等价于 c(4.0, 3.0)
```

uniform initialization是值初始化,未定义的基础数据类型的变量值设为`0`(或`nullptr`).

```
int i;      // i has undefined value
int j{};    // j is initialized by 0
int* p;     // P has undefined value
int* q{};   // q is initialized by nullptr
```

uniform initialization还具有防止窄化的功能,当自动类型转换可能使变量降低精度时报错.

```
int x1(5.3);         // OK, but OUCH: x1 becomes 5
int x2 = 5.3;        // OK, but OUCH: x2 becomes 5
int x3{5.0};         // ERROR: narrowing
int x4 = {5.3};      // ERROR: narrowing
char c1{7};          // OK: even though 7 is an int, this is not narrowing
char c2{99999};      // ERROR: narrowing (if 9999 doesn 'fut into a char)
std::vector<int> v1{1, 2, 4, 5};         // OK
std::vector<int> v2{1, 2.3, 4, 5.6};     // ERROR: narrowing
```

---

uniform initialization底层依赖于模板类`initializer_list<T>`,该类封装了一个`array<T,n>`.调用函数时该`array`内的元素可被编译器分解逐一传递给函数.若函数参数是`initializer_list<T>`,则传入的数据不会被拆解.

```
void print(std::initializer_list<int> vals) {
    for (auto p = vals.begin(); p != vals.end(); ++p) { //a list of values
        std::cout << *p << endl;
    }
}

print({12, 3, 5, 7, 11, 13, 17});       // pass a list of values to print()
```

若函数同时有**接收多个参数**的重载版本和**接收initializer list**的重载版本,则优先调用**接收initializer list**的重载版本.

```
class P {
public:
    // 有两个重载版本的构造函数,uniform initialization时优先调用接收initializer list的重载版本
    P(int a, int b) {
        cout << "P(int, int), a=" << a << ", b=" << b << endl;
    }

    P(initializer_list<int> initlist) {
        cout << "P(initializer list<int>), values= ";
        for (auto i : initlist)
            cout << i << ' ';
        cout << endl;
    }
};

P p(77, 5);      // P(int, int), a=77, b=5
P q{77, 5};      // P(initializer list<int>), values= 77 5
P r{77, 5, 42}; // P(initializer list<int>), values= 77 5 42
P s = {77, 5};  // P(initializer list<int>), values= 77 5
```

STL中的大部分容器和算法相关函数均有接收initializer list的重载版本,以`vector`、`min`和`max`为例:

```
#include <initializer_list>

vector(initializer_list<value_type> __l,
       const allocator_type &__a = allocator_type())
    : _Base(a)
```

```
    { _M_range_initalize(__l.begin(), __l.end(),
random_access_iterator_tag()); }

vector &operator=(initalizer_list <value_type> __l) {
    this->assign(__l.begin(), __l.end());
    return *this;
}

void insert(iterator __position, initializer_list<value_type> __l) {
    this->insert(__position, __l.begin(), __l.end());
}

void assign(initializer_list<value_type> __l) {
    this->assign(__l.begin(), __l.end());
}
```

```
vector<int> v1{2, 5, 7, 13, 69, 83, 50};
vector<int> v2({2, 5, 7513, 69, 83, 50});
vector<int> v3;
v3 = {2, 5, 7, 13, 69, 83, 50};
v3.insert(v3.begin() + 2, {0, 1, 2, 3, 4});

for (auto i : v3)
    cout << i << ' ';
cout << endl; // 2 5 0 1 2 3 4 7 13 69 83 50

cout << max({string("Ace"), string("Stacy"), string("Sabrina"),
string("Barkley")}); //Stacy
cout << min({string("Ace"), string("Stacy"), string("Sabrina"),
string("Sarkley")}); //Ace
cout << max({54, 16, 48, 5}); //54
cout << min({54, 16, 48, 5});//5
```

`=default`、`=delete`

- 使用`=default`使得编译给类加上默认的构造函数、析构函数、拷贝构造函数、拷贝赋值函数、移动构造函数等.

  在C++11之前,我们都是手动给类添加空的构造函数等函数,但是这样手动添加的函数与编译器生成的默认构造函数是不同的,一个影响就是使类不再是POD类型,减少了编译器对其优化的可能性.

  ```
  class A {
  public:
      A() {}          // 手动添加的空参构造函数
      A(int mem) : member(mem) {}
  private:
      int member;
  };

  class B {
  public:
      B() = default;  // 使用编译器生成的空参构造函数
      B(int mem) : member(mem) {}
  private:
      int member;
  };
  ```

```
16
17 int main() {
18     cout << std::is_pod<A>::value << endl;      // false
19     cout << std::is_pod<B>::value << endl;      // true
20     return 0;
21 }
```

- `=delete`(或简写为`=0`)表示删除该函数,使得该类不具有对应的构造、析构、拷贝构造、拷贝赋值、析构等功能.

  在C++11之前的做法通常是将这些函数声明为`private`函数,这样外界就不能调用这些函数了.但是这种做法对友元的支持不好.

```
1 struct NoCopy {
2     NoCopy() = default;                           // use the synthesized
default constructor
3     NoCopy(const NoCopy &) = delete;              // no copy
4     NoCopy &operator=(const NoCopy &) = delete; // no assignment
5     ~NoCopy() = default;                          // use the synthesized
destructor
6 // other members
7 };
8
```

```
1 struct NoDtor {
2     NoDtor() = default; // use the synthesized default constructor
3     ~NoDtor() = delete; // we can't destroy objects of type NoDtor .
4 };
5
6 NoDtor nd;                  //error: NoDtor destructor is deleted
7 NoDtor *p = new NoDtor();   // ok: but we can't delete p
8 delete p;                  //error: NoDtor destructor is deleted
```

```
1 class PrivateCopy {
2 private:
3     // C++11之前的做法,拷贝赋值函数仅能被内部和友元调用
4     PrivateCopy(const PrivateCopy &);
5     PrivateCopy &operator=(const PrivateCopy &);
6     // other members
7 public:
8     PrivateCopy() = default;    // use the synthesized default
constructor
9     ~PrivateCopy();             // users can define objects of this type
but not copy them
10 };
```

alias template

alias template使用关键字`using`,其用法类似于`typedef`.

```cpp
template<typename T>
using Vec = std::vector<T, MyAlloc<T>>;     // 使用alias template语法定义含有自定义分配器的vector

Vec<int> container;     // 使用Vec类型
```

上述功能使用**宏定义**或`typedef`都不能实现

- 要想使用宏定义实现该功能,从语义上来说,应该这样实现:

  ```cpp
  #define Vec<T> std::vector<T, MyAlloc<T>>       // 理想情况下应该这样写,但不能通过编译
  Vec<int> container;
  ```

  但是`define`不支持以小括号定义参数,要想符合语法,需要这样写

  ```cpp
  #define Vec(T) std::vector<T, MyAlloc<T>>       // 能通过编译,但是使用小括号失去了泛型的语义
  Vec(int) container;
  ```

  这样可以通过编译,但是`Vec(int)`这种指定泛型的方式与原生指定泛型的方式不一致.

- `typedef`根本不接受参数,因此也不能实现上述功能.

模板模板参数也需要通过alias template指定其它模板参数的初值:

```cpp
template<typename T, template<typename U> class Container>
class XCls {
private:
    Container<T> c;
public:
    // ...
};

// 错误写法:
XCls<string, list> mylst2;      // 错误:虽然list的第二模板参数有默认值,但是其作模板模板参数时不能自动推导

// 正确写法: 使用alias template指定第二模板参数
template<typename T>
using LST = list<T, allocator<T>>
XCls<string, list> mylst2;      // 正确:模板LST只需要一个模板参数
```

decltype

`decltype`实现了`typeof`语法,可以推断出表达式的类型.

```cpp
map<string, float> coll;

map<string, float>::value_type elem;    // C++11以前的写法
decltype(coll)::value_type elem;        // 使用decltype,就不用在程序中写死变量coll的类型了
```

`decltype`语法常用于**声明返回值类型、元编程**和**代指lambda函数的类型**上.

下面程序使用`decltype`声明函数`add`的返回值类型:

```
template <typename T1, typename T2>
decltype(x+y) add(T1 x, T2 y);          // error: 'x' and 'y' was not
declared in this scope
```

从语法上来说,上述程序是错误的,因为变量`x`和`y`在函数外访问不到,因此需要使用C++11声明返回值类型的新语法:

```
template <typename T1, typename T2>
auto add(T1 x, T2 y) -> decltype(x+y);
```

`decltype`语法进一步增强了模板语法的灵活性:

```
template <typename T>
void test_decltype(T obj) {

    map<string, float>::value_type elem1;

    typedef typename decltype(Obj)::iterator iType;
    typedef typename T::iterator iType;

    decltype(obj) anotherObj(obj);
}
```

`decltype`语法也可用于代指lambda函数的类型:

```
// 定义lambda函数,lambda函数作为变量的变量类型较复杂,因此使用auto进行推断
auto cmp = [](const Person &p1, const Person &p2) {
    return p1.lastname() < p2.lastname() ||
           (p1.lastname() == p2.lastname() && p1.firstname() <
p2.firstname());
};

// 使用decltype语法推断lambda函数cmp的类型
std::set<Person, decltype(cmp)> coll(cmp);
```

lambda函数

lambda函数既可以用作变量,也可以立即执行:

```
[] {
    std::cout << "hello lambda" << std::endl;
};

// 用作变量
auto l = [] {
    std::cout << "hello lambda" << std::endl;
};
```

```
l();

// 直接执行
[] {
    std::cout << "hello lambda" << std::endl;
}();
```

---

lambda函数的完整语法如下:

$$[...](...)mutable_{opt}\ \ throwSpeC_{opt}\ \rightarrow\ retType_{opt}\{...\}$$

其中$mutable_{opt}$、$throwSpeC_{opt}$和$retType_{opt}$都是可选的.

$[...]$部分指定可以在函数体内访问的外部非static对象,可以通过这部分访问函数作用域外的变量.

- `[=]`表示使用值传递变量.
- `[&]`表示使用引用传递变量.

```
int id = 0;
auto f = [id]() mutable {
    std::cout << "id:" << id << std::endl;
    ++id;
};
id = 42;
f();                            // id:0
f();                            // id:1
f();                            // id:2
std::cout << id << std::endl;   // 42
```

lambda函数使用时相当于仿函数(functor),$[...]$中传入的对象相当于为仿函数的成员变量.

```
class Functor {
    private:
    int id; // copy of outside id
    public:
    void operator()() {
        std::cout << "id: " << id << std::endl;
        ++id; // OK
    }
};
Functor f;
```

与STL结合时,相比于仿函数,lambda函数通常更优雅:

```
// lambda函数充当predict谓词
vector<int> vi{5, 28, 50, 83, 70, 590, 245, 59, 24};
int x = 30;
int y = 100;
remove_if(vi.begin(), vi.end(),
            [x, y](int n) { return x < n && n < y; });
```

```
// 仿函数充当predict谓词
```

```
class LambdaFunctor {
public:
    LambdaFunctor(int a, int b) : m_a(a), m_b(b) {}

    bool operator()(int n) const {
        return m_a < n && n < m_b;
    }

private:
    int m_a;
    int m_b;
};

remove_if(vi.begin(), vi.end(),
            LambdaFunctor(x, y));
```

variadic template

variadic template是C++11引入的最重要的新特性之一,可以用来实现递归,一般形式如下:

```
void func() { /*... */}        // 空函数,用于结束递归

template<typename T, typename... Types>
void func(const T &firstArg, const Types &... args) {
    // 将输入参数分为一个和一包
    // 先处理第一个参数firstArg,再递归调用func函数处理余下的一包
    func(args...);
}
```

下面6个示例演示variadic template的强大之处

示例1: 定义`printX()`函数输出任意数目任意类型的变量

```
// 重载版本1,用于结束递归
void printX() {
}

// 重载版本2,先输出第一个参数,再递归调用自己处理余下的参数
template<typename T, typename... Types>
void printX(const T &firstArg, const Types &... args) {
    cout << firstArg << endl;
    printX(args...);
}

// 重载版本3,可以与重载版本1并存么?
template<typename... Types>
void printX(const Types &... args) {
}
```

语句`printX(7.5, "hello", bitset<16>(377), 42);`依次输出所有参数,各函数依次调用的顺序如下:

1. `printX(7.5, "hello", bitset<16>(377), 42)`,重载版本2
2. `printX("hello", bitset<16>(377), 42)`,重载版本2
3. `printX(bitset<16>(377), 42)`,重载版本2
4. `printX(42)`,重载版本2

5. `printX()`,重载版本1

在上面程序中,重载版本1和重载版本3的两个`printX`函数可以并存,但重载版本3的函数不会被调用,因为重载版本1的函数比重载版本3的**更特化**;当多个重载版本均满足输入参数时,编译器会优先调用更特化的版本.

示例2: 重写`printf()`函数

```
void printf(const char *s) {
    while (*s) {
        if (*s == '%' && *(++s) != '%')
            throw std::runtime_error("invalid format string: missing arguments");
        std::cout << *s++;
    }
}

template<typename T, typename... Args>
void printf(const char *s, T value, Args... args) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') {
            std::cout << value;
            printf(++s, args...); // call even when *s = 0 to detect extra arguments
            return;
        }
        std::cout << *s++;
    }
    throw std::logic_error("extra arguments provided to printf");
}
```

执行下面语句,可以看到程序正常输出:

```
int* pi = new int;
printf("params:%d %s %p %f \n", 15, "This is Ace.", pi, 3.141592653);
```

示例3: 重写`max()`函数接收任意参数

若`max()`函数的所有参数的类型相同的话,直接使用`initializer_list`传递参数即可.

```
std::max({10.0, 20.0, 4.5, 8.1});
```

若`initializer_list`中的参数类型不同则会报错

```
std::max({10, 20, 4.5, 8.1});       // ERROR: no matching function for call to 'max(<brace-enclosed initializer list>)'
```

使用variadic template重写`max`函数使之接受任意参数:

```cpp
int maximum(int n) {
    return n;
}

template<typename... Args>
int maximum(int n, Args... args) {
    return std::max(n, maximum(args...));
}
```

示例4: 重载`tuple`的`<<`运算符,以异于一般的方式处理头尾元素

```cpp
// helper: print element with index IDX of the tuple with MAX elements
template<int IDX, int MAX, typename... Args>
struct PRINT_TUPLE {
    static void print(ostream &os, const tuple<Args...> &t) {
        os << get<IDX>(t) << (IDX + 1 == MAX ? "" : ",");
        PRINT_TUPLE<IDX + 1, MAX, Args...>::print(os, t);
    }
};

// partial specialization to end the recursion
template<int MAX, typename... Args>
struct PRINT_TUPLE<MAX, MAX, Args...> {
    static void print(std::ostream &os, const tuple<Args...> &t) {
    }
};

// output operator for tuples
template<typename... Args>
ostream &operator<<(ostream &os, const tuple<Args...> &t) {
    os << "[";
    PRINT_TUPLE<0, sizeof...(Args), Args...>::print(os, t);
    return os << "]";
}
```

示例5: 递归继承实现`tuple`容器

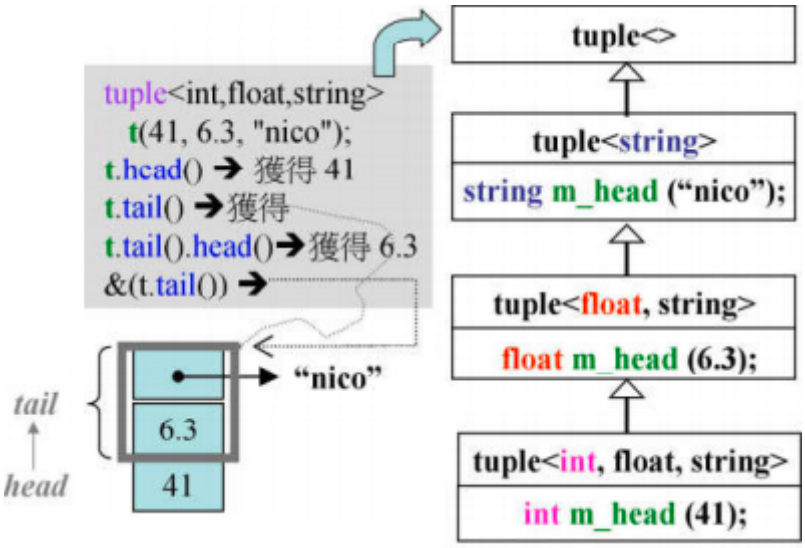
```cpp
// 定义 tuple类
template<typename... Values>
class tuple;

// 特化模板参数: 空参
template<>
class tuple<> {};

// 特化模板参数
template<typename Head, typename... Tail>
class tuple<Head, Tail...> :
        private tuple<Tail...>          // tuple类继承自tuple类,父类比子类少了一个模板参数
{
    typedef tuple<Tail...> inherited;   // 父类类型
protected:
    Head m_head;                        // 保存第一个元素的值
public:
```

```
    tuple() {}
    tuple(Head v, Tail... vtail)        // 构造函数：将第一个元素赋值给m_head,使用其他元素构建父类tuple
        : m_head(v), inherited(vtail...) {}

    Head head() { return m_head; }      // 返回第一个元素值
    inherited &tail() { return *this; } // 返回剩余元素组成的tuple(将当前元素强制转换为父类类型)
};
```



示例6: 递归复合实现`tuple`容器

```
template<typename... Values>
class tup;

template<>
class tup<> {};

template<typename Head, typename... Tail>
class tup<Head, Tail...> {
    typedef tup<Tail...> composited;
protected:
    composited m_tail;
    Head m_head;
public:
    tup() {}
    tup(Head v, Tail... vtail) : m_tail(vtail...), m_head(v) {}

    Head head() { return m_head; }
    composited &tail() { return m_tail; }
};
```

标准库新特性

move语义

C++11引入的move语义可以加速容器操作.

右值引用

右值不能出现在赋值运算符`=`的左边.

```
int foo() { return 5; }

int x = foo();
int *p = &foo();    // lvalue required as unary '&' operand
foo() = 7;          // lvalue required as left operand of assignment
```

当右值出现在赋值运算符`=`的右侧时,我们认为对其资源进行偷取/搬移(**move**)而非拷贝(**copy**)是合理的,依次:

1. 必须有语法让我们在**调用端**告诉编译器这是一个右值.
2. 必须有语法让我们在**被调用端**写出一个专门处理右值的**移动赋值函数**.

专门处理右值的函数使用`value_type&&`声明参数:

```
iterator insert(const_iterator __position, const value_type& __x);
iterator insert(const_iterator __position, value_type&& __x);
```

perfect forwarding

调用中间函数会改变变量的可变性和左值右值等性质,导致参数的非完美转交(unperfect forwarding),下面程序中的中间转交函数`forward()`破坏了参数本身是一个右值的性质:

```
// 函数process的两个重载版本,分别处理参数是左值和右值的情况
void process(int &i) {
    cout << "process(int&):" << i << endl;
}
void process(int &&i) {
    cout << "process(int&&):" << i << endl;
}

// 中间转交函数forward接收一个右值,但函数内将其作为左值传递给函数process了
void forward(int &&i) {
    cout << "forward(int&&):" << i << ", ";
    process(i);
}
```

```
int a = 0;
process(a);             // process(int&):0      (变量作左值)
process(1);             // process(int&&):1     (临时变量作右值)
process(std::move(a));  // process(int&&):0     (使用std::move将左值改为右值)
forward(2);             // forward(int&&):2, process(int&):2    (临时变量作左值传给forward函数,forward函数体内将变量作为右值传给process函数)
forward(std::move(a));  // forward(int&&):0, process(int&):0    (临时变量作左值传给forward函数,forward函数体内将变量作为右值传给process函数)

forward(a);             // ERROR: cannot bind rvalue reference of type 'int&&' to lvalue of type 'int'
const int &b = 1;
process(b);             // ERROR: binding reference of type 'int&' to 'const int' discards qualifiers
process(move(b));       // ERROR: binding reference of type 'int&&' to 'std::remove_reference<const int&>::type' {aka 'const int'} discards qualifiers
```

使用`std::forward()`函数可以完美转交变量,不改变其可变性和左值右值等性质.

```
// 函数process的两个重载版本,分别处理参数是左值和右值的情况
void process(int &i) {
    cout << "process(int&):" << i << endl;
}
void process(int &&i) {
    cout << "process(int&&):" << i << endl;
}

// 中间转交函数forward使用std::forward()转交变量
void forward(int &&i) {
    cout << "forward(int&&):" << i << ", ";
    process(std::forward<int>(i));
}
```

```
forward(2);             // forward(int&&):2, process(int&&):2    (临时变量作左值传给forward函数,forward函数体内使用std::forward函数包装变量,保留其作为右值的性质)
forward(std::move(a));  // forward(int&&):0, process(int&&):0    (临时变量作左值传给forward函数,forward函数体内使用std::forward函数包装变量,保留其作为右值的性质)
```

move-aware class

编写一个支持move语义的类`MyString`以演示**移动构造函数**和**移动赋值函数**的写法.

```
#include <cstring>

class MyString {
public:
    static size_t DCtor;    // 累计默认构造函数调用次数
    static size_t Ctor;     // 累计构造函数调用次数
    static size_t CCtor;    // 累计拷贝构造函数调用次数
    static size_t CAsgn;    // 累计拷贝赋值函数调用次数
    static size_t MCtor;    // 累计移动构造函数调用次数
    static size_t MAsgn;    // 累计移动赋值函数调用次数
    static size_t Dtor;     // 累计析构函数调用次数
private:
```

```
13      char *_data;
14      size_t _len;
15
16      void _init_data(const char *s) {
17          _data = new char[_len + 1];
18          memcpy(_data, s, _len);
19          _data[_len] = '\0';
20      }
21
22  public:
23      // 默认构造函数
24      MyString() : _data(nullptr), _len(0) { ++DCtor; }
25
26      // 构造函数
27      MyString(const char *p) : _len(strlen(p)) {
28          ++Ctor;
29          _init_data(p);
30      }
31
32      // 拷贝构造函数
33      MyString(const MyString &str) : _len(str._len) {
34          ++CCtor;
35          _init_data(str._data);
36      }
37
38      // 拷贝赋值函数
39      MyString &operator=(const MyString &str) {
40          ++CAsgn;
41          if (this != &str) {
42              if (_data) delete _data;
43
44              _len = str._len;
45              _init_data(str._data); //COPY!
46          }
47          return *this;
48
49      }
50
51      // 移动构造函数
52      MyString(MyString &&str) noexcept : _data(str._data), _len(str._len) {
53          ++MCtor;
54          str._len = 0;
55          str._data = nullptr;    // 将传入对象的_data指针设为nullptr,防止析构函数多
    次delete同一根指针
56      }
57
58      // 移动赋值函数
59      MyString &operator=(MyString &&str) noexcept {
60          ++MAsgn;
61          if (this != &str) {
62              if (_data) delete _data;
63              _len = str._len;
64              _data = str._data; //MOVE!
65              str._len = 0;
66              str._data = nullptr; // 将传入对象的_data指针设为nullptr,防止析构函数
    多次delete同一根指针
67          }
68          return *this;
```

```
    }

    //dtor
    virtual ~MyString() {
        ++Dtor;
        if (_data)
            delete _data;
    }
};

size_t MyString::DCtor = 0;
size_t MyString::Ctor = 0;
size_t MyString::CCtor = 0;
size_t MyString::CAsgn = 0;
size_t MyString::MCtor = 0;
size_t MyString::MAsgn = 0;
size_t MyString::Dtor = 0;
```

值得注意的有两点:

1. 移动构造函数和移动赋值函数通常不涉及内存操作,不会抛出异常,因此应加以`noexcept`修饰.
2. 在移动构造函数和移动赋值函数中,移动了原对象的数据后,要把原对象的数据指针置空,防止析构函数多次`delete`同一指针.

测试move语义对容器的作用

move语义可以减少深拷贝,可以加速容器操作,编写下述测试函数进行测试:

```
template<typename M, typename NM>
void test_moveable(M c1, NM c2, long &value) {
    char buf[10];

    // 测试保存moveable对象的容器
    typedef typename iterator_traits<typename M::iterator>::value_type V1type;
    clock_t timeStart = clock();
    for (long i = 0; i < value; ++i) {
        snprintf(buf, 10, "%d", rand());    // 向容器内放入随机字符串
        auto ite = c1.end();                // 定位尾端
        c1.insert(ite, V1type(buf));        // 安插於尾端 (對RB-tree和HT這只是hint)
    }
    cout << "construction, milli-seconds: " << (clock() - timeStart) << endl;
    cout << "size()= " << c1.size() << endl;

    output_static_data(*(c1.begin()));
    // 测试容器的std::move()语义
    M c11(c1);
    M c12(std::move(c1));
    c11.swap(c12);

    // 对保存non-movable对象的容器进行上述测试
    // ...
}

template<typename T>
```

```
void output_static_data(const T &myStr) {
    cout << typeid(myStr).name() << "-- " << endl;
    cout << "CCtor=" << T::CCtor
         << " MCtor=" << T::MCtor
         << "Asgn=" << T::CAsgn
         << "MAsgn=" << T::MAsgn
         << "Dtor=" << T::Dtor
         << "Ctor=" << T::Ctor
         << "DCtor=" << T::DCtor
         << endl;
}
```

```
long value = 3000000L;
test_moveable(vector<MyString>(), vector<MyStringNonMovable>(), value);
test_moveable(list<MyString>(), list<MyStringNonMovable>(), value);
test_moveable(deque<MyString>(), deque<MyStringNonMovable>(), value);
test_moveable(multiset<MyString>(), multiset<MyStringNonMovable>(), value);
test_moveable(unordered_multiset<MyString>(),
unordered_multiset<MyStringNonMovable>(), value);
```

测试结果:

- 在插入元素部分,只有`vector`容器的速度受元素是否movable影响大,这是因为只有容器`vector`在增长过程中会发生复制.
- 对于所有容器,其移动构造函数都远快于其拷贝构造函数,容器`vector`的移动复制函数仅仅发生了指针的交换,未发生元素的复制.

容器新特性

这部分内容与侯捷老师另一门课程STL标准库与泛型编程相同,可以参考该课程的笔记.