MyString(const char\* p)

MyString(const MyString& str)

MyString(MyString&& str) noexcept

MyString& operator=(const MyString&str)

MyString& operator=(Mystring&& str) noexcept

virtual ~MyString()

growable containers(会发生memory reallocation)只有两种 vector和deque（从中间插入元素的时候）

You need to inform C++(specifically std::vector)that your move constructor and destructor does not throw.

Then the move constructor will be called when the vector grows.

If the constructor is not noexcept, std::vector can't use it,

since then it can't ensure the exception guarantees demanded by the standard.

noexcept

设计一个类的时候，不知道使用者会把这个类放到哪一种容器之中，如果类有move function，加上noexcept

vector 动态增长

实现是一个数组，两倍成长

lvalue 可以出现于operator=左侧者

rvalue 只能出现于operator=右侧者

rvalue references 是C++0x引进来的一种新的引用类型

解决unnecessary copying问题，enable perfect forwarding

右值引用 rvalue references

rvalue 只能出现于 operator= 右侧

当rvalue出现于operator=(copy assignment)的右侧，我们认为对其资源进行偷取/搬移(move)而非拷贝(copy)是可以的，是合理的

那么

1、必须有语法让我们在调用端告诉编译器“这是个rvalue”

2、必须有语法让我们在被调用端写出一个专门处理rvalue的所谓move assignment函数

临时对象是一个右值

右值不能放在operator=左边

std::forward

std::move

template<typename T1, typename T2>

void functionA(T1&&t1, T2&& t2)

{

functionB(std::foward<T1>(t1), std::forward<T2>(t2));

}

##################################################################

void process(int& i)

{

std::*cout* << "process(int&): " << i << std::*endl*;

}

void process(int&& i)

{

std::*cout* << "process(int&&): " << i << std::*endl*;

}

void forward(int&& i)

{

std::*cout* << "forward(int&&): " << i << ", ";

process(i);

}

class MyString {

public:

static *size\_t* DCtor;//default constructor

static *size\_t* Ctor;//constructor

static *size\_t* CCtor;//copy constructor

static *size\_t* CAsgn;//copy assignment

static *size\_t* MCtor;//move constructor

static *size\_t* MAsgn;//move assignment

static *size\_t* Dtor;//destructor

private:

char\* \_data;

*size\_t* \_len;

void \_init\_data(const char\* s) {

\_data = new char[\_len + 1];

*memcpy*(\_data, s, \_len);

\_data[\_len] = '\0';

}

public:

//default constructor

MyString() : \_data(*NULL*), \_len(0){

++DCtor;

}

//constructor

MyString(const char\* p) : \_len(*strlen*(p)) {

++Ctor;

\_init\_data(p);

}

//copy constructor

MyString(const MyString& str) : \_len(str.\_len) {

++CCtor;

\_init\_data(str.\_data);

}

//move constructor, with "noexcept"

MyString(MyString&& str) noexcept

: \_data(str.\_data), \_len(str.\_len){

++MCtor;

str.\_len = 0;

str.\_data = *NULL*;//important

}

//copy assignment

MyString& operator=(const MyString& str) {

++CAsgn;

if (this != &str){

if (\_data) delete \_data;

\_len = str.\_len;

\_init\_data(str.\_data);

}

else{

}

return \*this;

}

//move assignment

MyString& operator=(MyString&& str) noexcept {

++MAsgn;

if (this != &str){

if (\_data) delete \_data;

\_len = str.\_len;

\_data = str.\_data;

str.\_len = 0;

str.\_data = *NULL*;//important

}

return \*this;

}

//destructor

virtual ~MyString() {

++Dtor;

if (\_data) {

delete \_data;

}

}

bool operator<(const MyString& rhs) const {

return std::*string*(this->\_data) <std::*string*(rhs.\_data);

}

bool operator==(const MyString& rhs) const {

return std::*string*(this->\_data) == std::*string*(rhs.\_data);

}

char\* get() const { return \_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;

#include <typeinfo>

template<typename T>

void output\_static\_data(const T& myStr) {

std::*cout* << typeid(myStr).name() << "--" << std::*endl*;

std::*cout* << "CCtor=" << T::CCtor << std::*endl*

<< "MCotr=" << T::MCtor << std::*endl*

<< "CAsgn=" << T::CAsgn << std::*endl*

<< "MAsgn=" << T::MAsgn << std::*endl*

<< "Dtor=" << T::Dtor << std::*endl*

<< "Ctor=" << T::Ctor << std::*endl*

<< "DCtor=" << T::DCtor

<< std::*endl*;

}

template<typename M>

void test\_moveable(M c,long& value) {

char buf[10];

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*());

auto ite = c.*end*();

c.*insert*(ite, V1type(buf));

}

std::*cout* << "construction, milli-seconds:" << (*clock*() - timeStart) << std::*endl*;

std::*cout* << "size()=" << c.*size*() << std::*endl*;

output\_static\_data(\*(c.*begin*()));

timeStart = *clock*();

M c1(c);

std::*cout* << "copy, milli-seconds: " << (*clock*() - timeStart) << std::*endl*;

timeStart = *clock*();

M c2(std::*move*(c1));

std::*cout* << "move copy, milli-seconds: " << (*clock*() - timeStart) << std::*endl*;

c1.*swap*(c2);

std::*cout* << "swap, milli-seconds: " << (*clock*() - timeStart) << std::*endl*;

}

template<typename M, typename NM>

void test\_moveable(M c1, NM c2, long& value) {

char buf[10];

typedef typename *iterator\_traits*<typename M::*iterator*>::*value\_type* V1type;

std::*cout* << "test, with moveable elements" << std::*endl*;

*clock\_t* timeStart = *clock*();

for (long i = 0; i < value; ++i) {

*snprintf*(buf, 10, "%d", *rand*());

auto ite = c1.*end*();

c1.*insert*(ite, V1type(buf));

}

std::*cout* << "construction, milli-seconds:" << (*clock*() - timeStart) << std::*endl*;

std::*cout* << "size()=" << c1.*size*() << std::*endl*;

output\_static\_data(\*(c1.*begin*()));

timeStart = *clock*();

M c11(c1);

std::*cout* << "copy, milli-seconds: " << (*clock*() - timeStart) << std::*endl*;

timeStart = *clock*();

M c12(std::*move*(c1));

std::*cout* << "move copy, milli-seconds: " << (*clock*() - timeStart) << std::*endl*;

c11.*swap*(c12);

std::*cout* << "swap, milli-seconds: " << (*clock*() - timeStart) << std::*endl*;

std::*cout* << "test, with non-moveable elements" << std::*endl*;

}

int *main*()

{

long value = 3000000L;

test\_moveable(std::*vector*<MyString>(), value);

//test\_moveable(vector<MyString>(), vector<MyStrNoMove>, value);

// MyStrNoMove是MyString去掉移动赋值和移动构造

int a = 0;

process(a);

process(1);

process(std::*move*(a));

forward(2);

forward(std::*move*(a));

std::*string* s1("Hello");

std::*string* s2("World");

s1 + s2 = s2;

std::*cout* << "s1: " << s1 << *endl*;

std::*cout* << "s2: " << s2 << *endl*;

string() = "World";

return 0;

}