auto自动类型推导

1. auto用于从初始化表达式中推断出变量的数据类型
2. auto的使用有以下两点必须注意：
   1)auto声明的变量必须要初始化，否则编译器不能判断变量的类型。
   2)auto不能被声明为返回值，auto不能作为形参，auto不能被修饰为模板参数
3. 关于效率: auto实际上实在编译时对变量进行了类型推导，所以不会对程序的运行效率造成不良影响。另外，auto并不会影响编译速度，因为编译时本来也要右侧推导然后判断与左侧是否匹配。

lambda表达式

lambda表达式是匿名函数，可以认为是一个可执行体functor

```
[函数对象参数] (操作符重载函数参数) mutable或exception声明 -> 返回值类型 {函数体}
```

这是一般的函数的写法：

```
// 这里要求n>=0，同时n的取值不能太大，会溢出
// 为了方便，这里并没有处理上面说到的问题
int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; ++ i) fact *= i;
    return fact;
}
```

Lambda表达式的写法：

```
auto factorial = [](int n) {
    int fact = 1;
    for (int i = 1; i <= n; ++ i) fact *= i;
    return fact;
};
```

几个Lambda函数的例子:

```
[](int x, int y) { return x + y; } // 隐式返回类型
[](int& x) { ++x; }   // 没有return语句 -> lambda 函数的返回类型是'void'
[]() { ++global_x; }  // 没有参数,仅访问某个全局变量
[]{ ++global_x; }     // 与上一个相同,省略了()
```

这样显示指定返回类型:

```
[](int x, int y) -> int { int z = x + y; return z; }
```

Lambda闭包

Lambda 函数可以引用在它之外声明的变量. 这些变量的集合叫做一个闭包. 闭包被定义在 Lambda 表达式声明中的方括号 [] 内。这个机制允许这些变量被按值或按引用捕获。如下图的例子：

1. []不捕获任何变量。
2. [&]捕获外部作用域中所有变量，并作为引用在函数体中使用（按引用捕获）。
3. [=]捕获外部作用域中所有变量，并作为副本在函数体中使用(按值捕获)。
4. [=,&foo]按值捕获外部作用域中所有变量，并按引用捕获foo变量。
5. [bar]按值捕获bar变量，同时不捕获其他变量。
6. [this]捕获当前类中的this指针，让lambda表达式拥有和当前类成员函数同样的访问权限。如果已经使用了&或者=，就默认添加此选项。捕获this的目的是可以在lamda中使用当前类的成员函数和成员变量。

右值引用与移动语义

在C++11中所有的值必属于左值、将亡值、纯右值三者之一。比如，非引用返回的临时变量、运算表达式产生的临时变量、原始字面量和lambda表达式等都是纯右值。而将亡值是C++11新增的、与右值引用相关的表达式，比如，将要被移动的对象、T&&函数返回值、std::move返回值和转换为T&&的类型的转换函数的返回值等。

```
T&& k = getVar();
```

这就是右值引用，我们知道左值引用是对左值的引用，那么，对应的，对右值的引用就是右值引用，而且右值是匿名变量，我们也只能通过引用的方式来获取右值。这里，getVar()产生的临时值不会在表达式结束之后就销毁了，而是会被“续命”，他的生命周期将会通过右值引用得以延续，和变量k的声明周期一样长。

右值引用的第一个特点：避免临时对象的拷贝构造和析构

通过右值引用的声明，右值又“重获新生”，其生命周期与右值引用类型变量的生命周期一样长，只要该变量还活着，该右值临时量将会一直存活下去。让我们通过一个简单的例子来看看右值的生命周期。

```
#include <iostream>
using namespace std;
int g_constructCount=0;
int g_copyConstructCount=0;
int g_destructCount=0;
struct A
{
    A(){
        cout<<"construct: "<<++g_constructCount<<endl;
    }
    A(const A& a)
    {
        cout<<"copy construct: "<<++g_copyConstructCount <<endl;
    }
    ~A()
    {
        cout<<"destruct: "<<++g_destructCount<<endl;
    }
};
A GetA()
{
    return A();
}
int main() {
    A a = GetA();
    return 0;
}
```

输出结果：

```
construct: 1
copy construct: 1
destruct: 1
copy construct: 2
destruct: 2
destruct: 3
```

从上面的例子中可以看到，在没有返回值优化的情况下，拷贝构造函数调用了两次，一次是GetA()函数内部创建的对象返回出来构造一个临时对象产生的，另一次是在main函数中构造a对象产生的。第二次的destruct是因为临时对象在构造a对象之后就销毁了。
通过右值引用来绑定函数返回值的话，结果又会是什么样的呢？

```
int main() {
    A&& a = GetA();
    return 0;
}
输出结果：
construct: 1
copy construct: 1
destruct: 1
destruct: 2
```

通过右值引用，比之前少了一次拷贝构造和一次析构，原因在于右值引用绑定了右值，让临时右值的生命周期延长了。我们可以利用这个特点做一些性能优化，即**避免临时对象的拷贝构造和析构**。

右值引用的第二个特点

右值引用独立于左值和右值。意思是右值引用类型的变量可能是左值也可能是右值。

```
int&& var1 = 1;
```

var1类型为右值引用，但var1本身是左值，因为具名变量都是左值。
T&&是什么，一定是右值吗？

```
template<typename T>
void f(T&& t){}

f(10); //t是右值

int x = 10;
f(x); //t是左值
```

从上面的代码中可以看到，T&&表示的值类型不确定，可能是左值又可能是右值。

右值引用的第三个特点

T&& t在发生自动类型推断的时候，它是未定的引用类型（universal references），如果被一个左值初始化，它就是一个左值；如果它被一个右值初始化，它就是一个右值，它是左值还是右值取决于它的初始化。

上面的代码，对于函数templatevoid f(T&& t)，当参数为右值10的时候，根据universal references的特点，t被一个右值初始化，那么t就是右值；当参数为左值x时，t被一个左值引用初始化，那么t就是一个左值。需要注意的是，仅仅是当发生自动类型推导（如函数模板的类型自动推导，或auto关键字）的时候，T&&才是universal references。

```
template<typename T>
void f(T&& param);

template<typename T>
class Test {
    Test(Test&& rhs);
};
```

上面的例子中，param是universal reference，rhs是Test&&右值引用，因为模版函数f发生了类型推断，而Test&&并没有发生类型推导，因为Test&&是确定的类型了。

正是因为右值引用可能是左值也可能是右值，依赖于初始化，并不是一下子就确定的特点，我们可以利用这一点做很多文章，比如后面要介绍的移动语义和完美转发。

这里再提一下引用折叠，正是因为引入了右值引用，所以可能存在左值引用与右值引用和右值引用与右值引用的折叠，C++11确定了引用折叠的规则，规则是这样的：

- 所有的右值引用叠加到右值引用上仍然还是一个右值引用；
- 所有的其他引用类型之间的叠加都将变成左值引用。

```
T(T&& a) : m_val(val){ a.m_val=nullptr; }
```

这行代码实际上来自于一个类的构造函数，构造函数的一个参数是一个右值引用，为什么将右值引用作为构造函数的参数呢？

```
class A
{
public:
    A():m_ptr(new int(0)){cout << "construct" << endl;}
    A(const A& a):m_ptr(new int(*a.m_ptr)) //深拷贝的拷贝构造函数
    {
        cout << "copy construct" << endl;
    }
    ~A(){ delete m_ptr;}
private:
    int* m_ptr;
};
int main() {
    A a = GetA();
    return 0;
}
    输出：
construct
copy construct
copy construct
```

这个例子很简单，一个带有堆内存的类，必须提供一个深拷贝拷贝构造函数，因为默认的拷贝构造函数是浅拷贝，会发生“指针悬挂”的问题。如果不提供深拷贝的拷贝构造函数，上面的测试代码将会发生错误（编译选项-fno-elide-constructors），内部的m_ptr将会被删除两次，一次是临时右值析构的时候删除一次，第二次外面构造的a对象释放时删除一次，而这两个对象的m_ptr是同一个指针，这就是所谓的指针悬挂问题。提供深拷贝的拷贝构造函数虽然可以保证正确，但是在有些时候会造成额外的性能损耗，因为有时候这种深拷贝是不必要的。比如下面的代码：

上面代码中的GetA函数会返回临时变量，然后通过这个临时变量拷贝构造了一个新的对象a，临时变量在拷贝构造完成之后就销毁了，如果堆内存很大的话，那么，这个拷贝构造的代价会很大，带来了额外的性能损失。每次都会产生临时变量并造成额外的性能损失，有没有办法避免临时变量造成的性能损失呢？

```
class A
{
public:
    A() :m_ptr(new int(0)){}
    A(const A& a):m_ptr(new int(*a.m_ptr)) //深拷贝的拷贝构造函数
    {
        cout << "copy construct" << endl;
    }
    A(A&& a) :m_ptr(a.m_ptr)
    {
        a.m_ptr = nullptr;
        cout << "move construct" << endl;
    }
    ~A(){ delete m_ptr;}
private:
    int* m_ptr;
};
int main(){
    A a = Get(false);
}
输出：
construct
move construct
move construct
```

输出结果表明，并没有调用拷贝构造函数，只调用了move construct函数。

```
A(A&& a) :m_ptr(a.m_ptr)
{
    a.m_ptr = nullptr;
    cout << "move construct" << endl;
}
```

这个构造函数并没有做深拷贝，仅仅是将指针的所有者转移到了另外一个对象，同时，将参数对象a的指针置为空，这里仅仅是做了浅拷贝，因此，这个构造函数避免了临时变量的深拷贝问题。
上面这个函数其实就是移动构造函数，他的参数是一个右值引用类型，这里的A&&表示右值，为什么？前面已经提到，这里没有发生类型推断，是确定的右值引用类型。为什么会匹配到这个构造函数？因为这个构造函数只能接受右值参数，而函数返回值是右值，所以就会匹配到这个构造函数。这里的A&&可以看作是临时值的标识，对于临时值我们仅仅需要做浅拷贝即可，无需再做深拷贝，从而解决了前面提到的临时变量拷贝构造产生的性能损失的问题。这就是所谓的**移动语义**，右值引用的一个重要作用是用来支持移动语义的。

右值引用主要目的有两个方面：
1、消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率
2、能够更简洁明确地定义泛型函数

移动语义

转移语义可以将资源(堆、系统对象等)从一个对象转移到另一个对象，这样可以减少不必要的临时对象的创建、拷贝及销毁。移动语义与拷贝语义是相对的，可以类比文件的剪切和拷贝。

标准库函数std::move --- 将左值变成一个右值
编译器只对右值引用才能调用移动构造函数，那么如果已知一个命名对象不再被使用，此时仍然想调用它的移动构造函数，也就是把一个左值当做右值引用来使用，该怎么做呢？std::move，它唯一的功能是将一个左值强制转换为一个右值引用。如果是一些基本类型比如int和char[10]定长数组等类型，使用move的话仍然会发生拷贝（因为没有对应的移动构造函数）。所以，move对于含资源（堆内存或句柄）的对象来说更有意义。

完美转发 Perfect Forwarding

完美转发使用这样的场景：需要将一组参数原封不动地传递给另一个函数。原封不动不仅仅是参数的值不变，在C++中还有以下的两组属性：

- 左值/右值
- const / non-const
  完美转发就是在参数传递过程中，所有这些属性和参数值都不能改变。在泛型函数中，这样的需求十分普遍。
  为了保证这些属性，泛型函数需要重载各种版本，左值右值不同版本，还要分别对应不同的const关系，但是如果只定义一个右值引用参数的函数版本，这个问题就迎刃而解了，原因在于：C++11对T&&的类型推导： 右值实参为右值引用，左值实参仍然为左值。

在函数模板中，完全依照模板的参数的类型（即保持参数的左值、右值特征），将参数传递给函数模板中调用的另外一个函数。C++11中的std::forward正是做这个事情的，他会按照参数的实际类型进行转发。看下面的例子：

```
void processValue(int& a){ cout << "lvalue" << endl; }
void processValue(int&& a){ cout << "rvalue" << endl; }
template <typename T>
void forwardValue(T&& val)
{
    processValue(std::forward<T>(val)); //照参数本来的类型进行转发。
}
void Testdelcl()
{
    int i = 0;
    forwardValue(i); //传入左值
    forwardValue(0);//传入右值
}
输出:
lvaue
rvalue
```

右值引用T&&是一个universal references，可以接受左值或者右值，正是这个特性让他适合作为一个参数的路由，然后再通过std::forward按照参数的实际类型去匹配对应的重载函数，最终实现完美转发。

智能指针

作用

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域时，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

auto_ptr

auto_ptr基于所有权转移的语义，即将一个就的auto_ptr赋值给另外一个新的auto_ptr时，旧的那一个就不再拥有该指针的控制权（内部指针被赋值为null),那么这就会带来一些根本性的破绽：

- 函数参数传递时，会有隐式的赋值，那么原来的auto_ptr自动失去了控制权
- 自我赋值时，会将自己内部指针赋值为null，造成bug

shared_ptr

基于引用计数的智能指针，会统计当前有多少个对象同时拥有该内部指针；当引用计数降为0时，自动释放。

shared_ptr的实现

难点一：引用计数，使用静态map表的方式来实现：

```
static map<T*, int> _map;   //静态数据成员需要在类外进行初始化
```

map表建立了原生指针T* 和次数一个映射。 如图所示，如果有四个shared_ptr（自主实现）类型的变量同时指向一块堆内存，map表中就会建立原生指针_ptr和4之间的一个映射。如果有更多的变量指向该块堆内存或者A、B、C、D其中有任何一个变量析构了，都会引起引用计数的变化。

难点二：为什么成员运算符（俗称箭头）的重载返回类型是原生指针的类型？shared_ptr名为指针，实际上是类。对一个类采用成员运算符重载，返回值很自然的就是类中的成员了。

```
template<typename T>
T* shared_ptr<T>::operator->()     //成员运算符重载
{
    return _ptr;
}
```

难点三：引用计数是如何实现按需变化的？如下代码所示：if语句一定会进入，是否执行还得两说！if语句一经进入，引用计数就自减1了，在决定释放内存之前，万万牢记：不要对NULL指针进行操作，这就是if语句后半部分存在的意义。这小段代码在析构函数和赋值运算符重载中都出现了。值得注意一下。

```
if (--_map[_ptr] <= 0 && NULL != _ptr)
{
    delete _ptr;
    _ptr = NULL;
    _map.erase(_ptr);
}
```

完整代码：

```cpp
#include<iostream>
#include<map>
using namespace std;
template<typename T>
class mshared_ptr
{
public:
    mshared_ptr(T *ptr = NULL);     //构造方法
    ~mshared_ptr();     //析构方法
    mshared_ptr(mshared_ptr<T> &src);       //拷贝构造
    mshared_ptr& operator = (mshared_ptr<T> &src);      //赋值运算符重载
    T& operator*();     //解引用运算符重载
    T* operator->();    //成员运算符重载
private:
    T *_ptr;
    static map<T*, int> _map;       //静态数据成员需要在类外进行初始化
};

template<typename T>
map<T*, int> mshared_ptr<T>::_map;

template<typename T>
mshared_ptr<T>::mshared_ptr(T *ptr)     //构造方法
{
    cout << "mshared_ptr的构造方法正被调用！" << endl;
    _ptr = ptr;
    _map.insert(make_pair(_ptr, 1));
}

template<typename T>
mshared_ptr<T>::~mshared_ptr()      //析构方法
{
    cout << "mshared_ptr的析构方法正被调用！" << endl;
    if (--_map[_ptr] <= 0 && NULL != _ptr)
    {
        delete _ptr;
        _ptr = NULL;
        _map.erase(_ptr);
    }
}

template<typename T>
mshared_ptr<T>::mshared_ptr(mshared_ptr<T> &src)    //拷贝构造
{
    _ptr = src._ptr;
    _map[_ptr]++;
}

template<typename T>
mshared_ptr<T>& mshared_ptr<T>::operator=(mshared_ptr<T> &src)      //赋值运算符重载
{
```

```
    if (_ptr == src._ptr)
    {
        return *this;
    }

    if (--_map[_ptr] <= 0 && NULL != _ptr)
    {
        delete _ptr;
        _ptr = NULL;
        _map.erase(_ptr);
    }

    _ptr = src._ptr;
    _map[_ptr]++;
    return *this;
}

template<typename T>
T& mshared_ptr<T>::operator*()      //解引用运算符重载
{
    return *_ptr;
}

template<typename T>
T* mshared_ptr<T>::operator->()     //成员运算符重载
{
    return _ptr;
}
```

循环引用

Share_ptr存在**循环引用**的问题。
两个对象互相使用一个shared_ptr成员变量指向对方会造成循环引用。导致引用计数失效。

```
class CB;
class CA
{
public:
    CA() { cout << "CA() called! " << endl; }
    ~CA() { cout << "~CA() called! " << endl; }
    void set_ptr(shared_ptr<CB>& ptr) { m_ptr_b = ptr; }
    void b_use_count() { cout << "b use count : " << m_ptr_b.use_count() << endl; }
    void show() { cout << "this is class CA!" << endl; }
private:
    shared_ptr<CB> m_ptr_b;
};

class CB
{
public:
    CB() { cout << "CB() called! " << endl; }
    ~CB() { cout << "~CB() called! " << endl; }
    void set_ptr(shared_ptr<CA>& ptr) { m_ptr_a = ptr; }
    void a_use_count() { cout << "a use count : " << m_ptr_a.use_count() << endl; }
    void show() { cout << "this is class CB!" << endl; }
private:
    shared_ptr<CA> m_ptr_a;
};

void test_refer_to_each_other()
{
    shared_ptr<CA> ptr_a(new CA());
    shared_ptr<CB> ptr_b(new CB());

    cout << "a use count : " << ptr_a.use_count() << endl;
    cout << "b use count : " << ptr_b.use_count() << endl;

    ptr_a->set_ptr(ptr_b);
    ptr_b->set_ptr(ptr_a);

    cout << "a use count : " << ptr_a.use_count() << endl;
    cout << "b use count : " << ptr_b.use_count() << endl;
}
```

结果如下：

```
CA() called!
CB() called!
a use count : 1
b use count : 1
a use count : 2
b use count : 2
```

通过结果可以看到，最后CA和CB的对象并没有被析构，其中的引用效果如下图所示，起初定义完ptr_a和ptr_b时，只有①③两条引用，然后调用函数set_ptr后又增加了②④两条引用，当test_refer_to_each_other这个函数返回时，对象ptr_a和ptr_b被销毁，也就是①③两条引用会被断开，但是②④两条引用依然存在，每一个的引用计数都不为0，结果就导致其指向的内部对象无法释放

解决这种状况的办法就是将两个类中的一个成员变量改为weak_ptr对象，因为weak_ptr不会增加引用计数，使得引用形不成环，最后就可以正常的释放内部的对象，不会造成内存泄漏，比如将CB中的成员变量改为weak_ptr对象，代码如下：

```
class CB
{
public:
    CB() { cout << "CB() called! " << endl; }
    ~CB() { cout << "~CB() called! " << endl; }
    void set_ptr(shared_ptr<CA>& ptr) { m_ptr_a = ptr; }
    void a_use_count() { cout << "a use count : " << m_ptr_a.use_count() << endl; }
    void show() { cout << "this is class CB!" << endl; }
private:
    weak_ptr<CA> m_ptr_a;
};
```

测试结果如下：

```
CA() called!
CB() called!
a use count : 1
b use count : 1
a use count : 1
b use count : 2
~CA() called!
~CB() called!
```

通过这次结果可以看到，CA和CB的对象都被正常的析构了，引用关系如下图所示，流程与上一例子相似，但是不同的是④这条引用是通过weak_ptr建立的，并不会增加引用计数，也就是说CA的对象只有一个引用计数，而CB的对象只有2个引用计数，当test_refer_to_each_other这个函数返回时，对象ptr_a和ptr_b被销毁，也就是①③两条引用会被断开，此时CA对象的引用计数会减为0，对象被销毁，其内部的m_ptr_b成员变量也会被析构，导致CB对象的引用计数会减为0，对象被销毁，进而解决了引用成环的问题。

为了解决循环引用，可以使用weak_ptr来解决。
weak_ptr是为了配合shared_ptr而引入的一种智能指针，更像是shared_ptr的一个助手而不是智能指针（不具备普通指针的行为operator*和operator->），最大的作用在于协助shared_ptr工作，像旁观者那

样观测资源的使用情况！weak_ptr是与shared_ptr协同工作的：获取资源的观测权，并没有共享资源！而构造weak_ptr不会引起引用计数的变化。

unique_ptr

unique_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。
采用所有权模式.

```
unique_ptr<string> p3 (new string ("auto"));   //#4
unique_ptr<string> p4;                         //#5
p4 = p3;//此时会报错！！
```

编译器认为p4=p3非法，避免了p3不再指向有效数据的问题。因此，unique_ptr比auto_ptr更安全。

新的for语句

基于范围的for循环

nullptr空指针

在某种意义上来说，传统 C++ 会把 NULL、0 视为同一种东西，这取决于编译器如何定义 NULL，有些编译器会将 NULL 定义为 ((void*)0)，有些则会直接将其定义为 0。
C++ 不允许直接将 void * 隐式转换到其他类型，但如果 NULL 被定义为 ((void*)0)，那么当编译char *ch = NULL;时，NULL 只好被定义为 0。
而这依然会产生问题，将导致了 C++ 中重载特性会发生混乱，考虑：

```
void foo(char *);
void foo(int);
```

对于这两个函数来说，如果 NULL 又被定义为了 0 那么 foo(NULL); 这个语句将会去调用 foo(int)，从而导致代码违反直观。
为了解决这个问题，C++11 引入了 nullptr 关键字，专门用来区分空指针、0。
nullptr 的类型为 nullptr_t，能够隐式的转换为任何指针或成员指针的类型，也能和他们进行相等或者不等的比较。

列表初始化

数据类型 变量{初始化列表}

初始化列表可以在花括号“{}”之前使用等号=，其效果与不带等号的初始化相同。如： `int a{10}` 和 `int a = {10}` 。

```
//数组列表初始化
int xx[5]={1,2,3,4,5};
int yy[]={6,7,8,9,0};

//值类型进行初始化
int a{10};
int b={10};
int c={10.123}; // 编译器报错，g++ 5.3.1当列表初始化用于值类型的时候，如果有精度损失，编译器会报错。

//列表初始化还可以用结构体
typedef struct Str{
   int x;
   int y;
}Str;
Str s = {10,20};

//列表初始化类,必须是public成员,如果含有私有成员会失败
class Cls{
public:
   int x;
   int y;
};
Cls c  = {10,20};

//vector不仅可以使用列表初始化，还可以使用列表进行赋值，数组不能用列表赋值
vector<int>v1={1,2,3,4,5,6,7,8,9}; // 初始化
vector<int>v2;
v2={3,4,5,6,7}; //赋值

//map列表初始化
map<string ,int> m = {
      {"x",1},
      {"y",2},
      {"z",3}
};

//用函数返回初始化列表只展示关键代码，相关头文件自行添加
//同理结构体，类，map的返回也可以使用初始化列表返回
vector<int> getVector()
{
  return {1,2,3,4,5};
}

int main()
{
  vector<int> v = getVector();
  cout<<v[0]<<v[1]<<v.size()<<endl;
  return 0 ;
}
```

decltype类型指示符

有时候会有这样的需求，我们需要知道一个表达式的类型，并使用该类型去定义一个变量，例如：

```
int a = 10;
int b = 20;
auto c = a + b; // OK a+b的类型是int，此时c的类型是int，并且c的值是 a+b
```

auto可以解决部分问题，例如我们定义的变量的类型就是表达式 a+b 的类型，但是如果我们仅仅需要定义一个与表达式 a+b 的类型相同的变量，但是我们又不希望将表达式a+b的值赋值给刚刚定义的变量，我们希望赋另外一个值或者是仅仅定义变量而不赋值呢。 这就需要用到C++11 提供的另一个类型说明符 decltype了。decltype作用于一个表达式，并且返回该表达式的类型，在此过程中编译器分析表达式的类型，并不会计算表达式的值。例如：

```
int a = 10;
int b = 20;
decltype(a+b) c = 50; // OK c的类型就是 a+b 的类型int
```

对于引用类型decltype有一些特别的地方：

```
int a = 20 ;
int &b = a;
decltype(b) c ;  // Error c是引用类型必须赋值
decltype(b) d = a; // OK  d是引用类型，指向a
```

可以看到decltype如果作用于一个引用类型，其得到的还是一个引用类型。我们知道一个引用类型在使用的时候一般会当作其关联的那个变量的同义词处理，例如如果使用 cout<<b<<endl; 其中b实际上相当于a，但是decltype作用于引用类型的时候会保留引用性质。
如果一个表达式是一个解指针引用的操作，decltype得到的也是一个引用类型：

```
int a = 20 ;
int *p = &a;
decltype(*p) c = a;  // c的类型是int&
c = 50;
cout<<a<<endl;  // 输出50
```

当decltype作用于一个变量的时候，变量加不加括号是有区别的，例如：

```
int a = 20;
decltype(a) b = 30; //ok b的类型是 int
decltype((a)) c = a ; // ok c的类型是int& 其关联变量 a
```

=default 生成默认构造函数

在C++的类中，如果我们没有定义构造函数，编译器会为我们合成默认的无参构造函数，如果我们定义了构造函数，则编译器就不生成默认构造函数了，但是如果我们定义构造函数同时也希望编译器生成默认构造函数呢? C++11中可以通过在构造函数的声明中直接 =default的方式要求编译器生成构造函数。

```
class ClassName{
    public:
        ClassName(int x);
        ClassName()=default; // 显示要求编译器生成构造函数
};
```

多线程

std::thread 构造，头文件：#include
互斥量：Mutex又称互斥量，C++ 11中与 Mutex 相关的类（包括锁类型）和函数都声明在 头文件中。
<condition_variable>：该头文件主要声明了与条件变量相关的类，包括 std::condition_variable和std::condition_variable_any。
：该头文件主要声明了std::promise, std::package_task两个Provider类，以及std::future和std::shared_future两个Future类，另外还有一些与之相关的类型和函数，std::async()函数就声明在此头文件中。