C++11常用新特性快速一览

1. nullptr

nullptr 出现的目的是为了替代 NULL。
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。
当需要使用 NULL 时候，养成直接使用 nullptr的习惯。

2. 类型推导

C++11 引入了 auto 和 decltype 这两个关键字实现了类型推导，让编译器来操心变量的类型。

auto
auto 在很早以前就已经进入了 C++，但是他始终作为一个存储类型的指示符存在，与 register 并存。在传统 C++ 中，如果一个变量没有声明为 register 变量，将自动被视为一个 auto 变量。而随着 register 被弃用，对 auto 的语义变更也就非常自然了。
使用 auto 进行类型推导的一个最为常见而且显著的例子就是迭代器。在以前我们需要这样来书写一个迭代器：

```
for(vector<int>::const_iterator itr = vec.cbegin(); itr != vec.cend(); ++itr)
```

而有了 auto 之后可以：

```
// 由于 cbegin() 将返回 vector<int>::const_iterator
// 所以 itr 也应该是 vector<int>::const_iterator 类型
for(auto itr = vec.cbegin(); itr != vec.cend(); ++itr);
```

一些其他的常见用法：

```
auto i = 5;              // i 被推导为 int
auto arr = new auto(10) // arr 被推导为 int *
```

注意：auto 不能用于函数传参，因此下面的做法是无法通过编译的（考虑重载的问题，我们应该使用模板）：

```
int add(auto x, auto y);
```

此外，auto 还不能用于推导数组类型：

```
#include <iostream>

int main() {
 auto i = 5;

 int arr[10] = {0};
 auto auto_arr = arr;
 auto auto_arr2[10] = arr;

 return 0;
}
```

decltype

decltype 关键字是为了解决 auto 关键字只能对变量进行类型推导的缺陷而出现的。它的用法和 sizeof 很相似：

```
decltype(表达式)
```

在此过程中，编译器分析表达式并得到它的类型，却不实际计算表达式的值。
有时候，我们可能需要计算某个表达式的类型，例如：

```
auto x = 1;
auto y = 2;
decltype(x+y) z;
```

拖尾返回类型、auto 与 decltype 配合 你可能会思考，auto 能不能用于推导函数的返回类型。考虑这样一个例子加法函数的例子，在传统 C++ 中我们必须这么写：

```
template<typename R, typename T, typename U>
R add(T x, U y) {
    return x+y
}
```

这样的代码其实变得很丑陋，因为程序员在使用这个模板函数的时候，必须明确指出返回类型。但事实上我们并不知道 add() 这个函数会做什么样的操作，获得一个什么样的返回类型。

在 C++11 中这个问题得到解决。虽然你可能马上回反应出来使用 decltype 推导 x+y 的类型，写出这样的代码：

```
decltype(x+y) add(T x, U y);
```

但事实上这样的写法并不能通过编译。 这是因为在编译器读到 decltype(x+y) 时，x 和 y 尚未被定义。为了解决这个问题，C++11 还引入了一个叫做拖尾返回类型（trailing return type），利用 auto 关键字将返回类型后置：

```
template<typename T, typename U>
auto add(T x, U y) -> decltype(x+y) {
    return x+y;
}
```

从 C++14 开始是可以直接让普通函数具备返回值推导，因此下面的写法变得合法：

```
template<typename T, typename U>
auto add(T x, U y) {
    return x+y;
}
```

3. 区间迭代

基于范围的 for 循环 C++11 引入了基于范围的迭代写法，我们拥有了能够写出像 Python 一样简洁的循环语句。 最常用的 std::vector 遍历将从原来的样子：

```
std::vector<int> arr(5, 100);
for(std::vector<int>::iterator i = arr.begin(); i != arr.end(); ++i) {
    std::cout << *i << std::endl;
}
```

变得非常的简单：

```
// & 启用了引用
for(auto &i : arr) {
    std::cout << i << std::endl;
}
```

4. 初始化列表

C++11 提供了统一的语法来初始化任意的对象，例如：

```
struct A {
    int a;
    float b;
};
struct B {
    B(int _a, float _b): a(_a), b(_b) {}
private:
    int a;
    float b;
};
```

```
A a {1, 1.1};    // 统一的初始化语法
B b {2, 2.2};
```

C++11 还把初始化列表的概念绑定到了类型上，并将其称之为 std::initializer_list，允许构造函数或其他函数像参数一样使用初始化列表，这就为类对象的初始化与普通数组和 POD 的初始化方法提供了统一的桥梁，例如：

```
#include <initializer_list>

class Magic {
public:
    Magic(std::initializer_list<int> list) {}
};

Magic magic = {1,2,3,4,5};
std::vector<int> v = {1, 2, 3, 4};
```

5. 模板增强

外部模板

传统 C++ 中，模板只有在使用时才会被编译器实例化。只要在每个编译单元（文件）中编译的代码中遇到了被完整定义的模板，都会实例化。这就产生了重复实例化而导致的编译时间的增加。并且，我们没有办法通知编译器不要触发模板实例化。

C++11 引入了外部模板，扩充了原来的强制编译器在特定位置实例化模板的语法，使得能够显式的告诉编译器何时进行模板的实例化：

```
template class std::vector<bool>;             // 强行实例化
extern template class std::vector<double>;  // 不在该编译文件中实例化模板
```

尖括号“>”

在传统 C++ 的编译器中，>>一律被当做右移运算符来进行处理。但实际上我们很容易就写出了嵌套模板的代码：

```
std::vector<std::vector<int>> wow;
```

这在传统C++编译器下是不能够被编译的，而 C++11 开始，连续的右尖括号将变得合法，并且能够顺利通过编译。

类型别名模板

在传统 C++中，typedef 可以为类型定义一个新的名称，但是却没有办法为模板定义一个新的名称。因为，模板不是类型。例如：

```
template< typename T, typename U, int value>
class SuckType {
public:
    T a;
    U b;
    SuckType():a(value),b(value){}
};
template< typename U>
typedef SuckType<std::vector<int>, U, 1> NewType; // 不合法
```

C++11 使用 using 引入了下面这种形式的写法，并且同时支持对传统 typedef 相同的功效：

```
template <typename T>
using NewType = SuckType<int, T, 1>;    // 合法
```

默认模板参数 我们可能定义了一个加法函数：

```
template<typename T, typename U>
auto add(T x, U y) -> decltype(x+y) {
    return x+y
}
```

但在使用时发现，要使用 add，就必须每次都指定其模板参数的类型。 在 C++11 中提供了一种便利，可以指定模板的默认参数：

```
template<typename T = int, typename U = int>
auto add(T x, U y) -> decltype(x+y) {
    return x+y;
}
```

6. 构造函数

委托构造

C++11 引入了委托构造的概念，这使得构造函数可以在同一个类中一个构造函数调用另一个构造函数，从而达到简化代码的目的：

```
class Base {
public:
    int value1;
    int value2;
    Base() {
        value1 = 1;
    }
    Base(int value) : Base() {  // 委托 Base() 构造函数
        value2 = 2;
    }
};
```

继承构造

在继承体系中，如果派生类想要使用基类的构造函数，需要在构造函数中显式声明。 假若基类拥有为数众多的不同版本的构造函数，这样，在派生类中得写很多对应的“透传”构造函数。如下：

```
struct A
{
  A(int i) {}
  A(double d,int i){}
  A(float f,int i,const char* c){}
  //...等等系列的构造函数版本
};
struct B:A
{
  B(int i):A(i){}
  B(double d,int i):A(d,i){}
  B(folat f,int i,const char* c):A(f,i,e){}
  //......等等好多个和基类构造函数对应的构造函数
};
```

C++11的继承构造：

```
struct A
{
  A(int i) {}
  A(double d,int i){}
  A(float f,int i,const char* c){}
  //...等等系列的构造函数版本
};
struct B:A
```

```
{
  using A::A;
  //关于基类各构造函数的继承一句话搞定
  //......
};
```

如果一个继承构造函数不被相关的代码使用，编译器不会为之产生真正的函数代码，这样比透传基类各种构造函数更加节省目标代码空间。

7. Lambda 表达式

Lambda 表达式，实际上就是提供了一个类似匿名函数的特性，而匿名函数则是在需要一个函数，但是又不想费力去命名一个函数的情况下去使用的。

Lambda 表达式的基本语法如下：

```
[ caputrue ] ( params ) opt -> ret { body; };
```

1. capture是捕获列表；
2. params是参数表；(选填)
3. opt是函数选项；可以填mutable,exception,attribute（选填）

- mutable说明lambda表达式体内的代码可以修改被捕获的变量，并且可以访问被捕获的对象的non-const方法。
- exception说明lambda表达式是否抛出异常以及何种异常。
- attribute用来声明属性。

4. ret是返回值类型（拖尾返回类型）。(选填)
5. body是函数体。

捕获列表：lambda表达式的捕获列表精细控制了lambda表达式能够访问的外部变量，以及如何访问这些变量。

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)。注意值捕获的前提是变量可以拷贝，且被捕获的变量在 lambda 表达式被创建时拷贝，而非调用时才拷贝。如果希望lambda表达式在调用时能即时访问外部变量，我们应当使用引用方式捕获。

```
int a = 0;
auto f = [=] { return a; };
a+=1;
cout << f() << endl;        //输出0
int a = 0;
auto f = [&a] { return a; };
a+=1;
cout << f() <<endl;        //输出1
```

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。

```
class A
{
 public:
     int i_ = 0;

     void func(int x,int y){
         auto x1 = [] { return i_; };                  //error,没有捕获外部变量
         auto x2 = [=] { return i_ + x + y; };         //OK
         auto x3 = [&] { return i_ + x + y; };        //OK
         auto x4 = [this] { return i_; };              //OK
         auto x5 = [this] { return i_ + x + y; };      //error,没有捕获x,y
         auto x6 = [this, x, y] { return i_ + x + y; };     //OK
         auto x7 = [this] { return i_++; };            //OK
};

int a=0 , b=1;
auto f1 = [] { return a; };                          //error,没有捕获外部变量
auto f2 = [&] { return a++ };                       //OK
auto f3 = [=] { return a; };                         //OK
auto f4 = [=] {return a++; };                        //error,a是以复制方式捕获的，无法修改
auto f5 = [a] { return a+b; };                       //error,没有捕获变量b
auto f6 = [a, &b] { return a + (b++); };             //OK
auto f7 = [=, &b] { return a + (b++); };             //OK
```

注意f4，虽然按值捕获的变量值均复制一份存储在lambda表达式变量中，修改他们也并不会真正影响到外部，但我们却仍然无法修改它们。如果希望去修改按值捕获的外部变量，需要显示指明lambda表达式为mutable。被mutable修饰的lambda表达式就算没有参数也要写明参数列表。

原因：lambda表达式可以说是就地定义仿函数闭包的"语法糖"。它的捕获列表捕获住的任何外部变量，最终会变为闭包类型的成员变量。按照C++标准，lambda表达式的operator()默认是const的，一个const成员函数是无法修改成员变量的值的。而mutable的作用，就在于取消operator()的const。

```
int a = 0;
auto f1 = [=] { return a++; };                 //error
auto f2 = [=] () mutable { return a++; };       //OK
```

lambda表达式的大致原理：

1. 每当你定义一个lambda表达式后，编译器会自动生成一个匿名类（这个类重载了()运算符），我们称为闭包类型（closure type）。
2. 那么在运行时，这个lambda表达式就会返回一个匿名的闭包实例，是一个右值。

3. 所以，我们上面的lambda表达式的结果就是一个个闭包。对于复制传值捕捉方式，类中会相应添加对应类型的非静态数据成员。
4. 在运行时，会用复制的值初始化这些成员变量，从而生成闭包。对于引用捕获方式，无论是否标记mutable，都可以在lambda表达式中修改捕获的值。
5. 至于闭包类中是否有对应成员，C++标准中给出的答案是：不清楚的，与具体实现有关。

lambda表达式是不能被赋值的：

```
auto a = [] { cout << "A" << endl; };
auto b = [] { cout << "B" << endl; };

a = b;   // 非法，lambda无法赋值
auto c = a;   // 合法，生成一个副本
```

闭包类型禁用了赋值操作符，但是没有禁用复制构造函数，所以你仍然可以用一个lambda表达式去初始化另外一个lambda表达式而产生副本。

在多种捕获方式中，最好不要使用[=]和[&]默认捕获所有变量。

默认引用捕获所有变量，你有很大可能会出现悬挂引用（Dangling references），因为引用捕获不会延长引用的变量的生命周期：

std::function<int(int)> add_x(int x) { return [&](int a) { return x + a; }; } 1 2 3 4 上面函数返回了一个lambda表达式，参数x仅是一个临时变量，函数add_x调用后就被销毁了，但是返回的lambda表达式却引用了该变量，当调用这个表达式时，引用的是一个垃圾值，会产生没有意义的结果。上面这种情况，使用默认传值方式可以避免悬挂引用问题。

但是采用默认值捕获所有变量仍然有风险，看下面的例子：

class Filter { public: Filter(int divisorVal): divisor{divisorVal} {}

```
std::function<bool(int)> getFilter()
{
    return [=](int value) {return value % divisor == 0; };
}
```

private: int divisor; };

这个类中有一个成员方法，可以返回一个lambda表达式，这个表达式使用了类的数据成员divisor。而且采用默认值方式捕捉所有变量。你可能认为这个lambda表达式也捕捉了divisor的一份副本，但是实际上并没有。因为数据成员divisor对lambda表达式并不可见，你可以用下面的代码验证：

// 类的方法，下面无法编译，因为divisor并不在lambda捕捉的范围

```
std::function<bool(int)> getFilter()
{
```

```
    return [divisor](int value) {return value % divisor == 0; };
}
```

原代码中，lambda表达式实际上捕捉的是this指针的副本，所以原来的代码等价于：

```
std::function<bool(int)> getFilter()
{
    return [this](int value) {return value % this->divisor == 0; };
}
```

尽管还是以值方式捕获，但是捕获的是指针，其实相当于以引用的方式捕获了当前类对象，所以lambda表达式的闭包与一个类对象绑定在一起了，这很危险，因为你仍然有可能在类对象析构后使用这个lambda表达式，那么类似"悬挂引用"的问题也会产生。所以，采用默认值捕捉所有变量仍然是不安全的，主要是由于指针变量的复制，实际上还是按引用传值。

lambda表达式可以赋值给对应类型的函数指针。但是使用函数指针并不是那么方便。所以STL定义在< functional >头文件提供了一个多态的函数对象封装std::function，其类似于函数指针。它可以绑定任何类函数对象，只要参数与返回类型相同。如下面的返回一个bool且接收两个int的函数包装器：

```
std::function<bool(int, int)> wrapper = [](int x, int y) { return x < y; };
```

lambda表达式一个更重要的应用是其可以用于函数的参数，通过这种方式可以实现回调函数。

最常用的是在STL算法中，比如你要统计一个数组中满足特定条件的元素数量，通过lambda表达式给出条件，传递给count_if函数：

```
int value = 3;
vector<int> v {1, 3, 5, 2, 6, 10};
int count = std::count_if(v.beigin(), v.end(), [value](int x) { return x > value;
});
```

再比如你想生成斐波那契数列，然后保存在数组中，此时你可以使用generate函数，并辅助lambda表达式：

```
vector<int> v(10);
int a = 0;
int b = 1;
std::generate(v.begin(), v.end(), [&a, &b] { int value = b; b = b + a; a = value;
return value; });
// 此时v {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

当需要遍历容器并对每个元素进行操作时：

```
std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
int even_count = 0;
for_each(v.begin(), v.end(), [&even_count](int val){
    if(!(val & 1)){
        ++ even_count;
    }
});
std::cout << "The number of even is " << even_count << std::endl;
```

大部分STL算法，可以非常灵活地搭配lambda表达式来实现想要的效果。

8. 新增容器

1. std::array

std::array 保存在**栈内存**中，相比**堆内存**中的 std::vector，我们能够灵活的访问这里面的元素，从而获得更高的性能。

std::array **会在编译时创建一个固定大小的数组**，std::array **不能够被隐式的转换成指针**，使用 std::array只需指定其**类型和大小**即可：

```
std::array<int, 4> arr= {1,2,3,4};

int len = 4;
std::array<int, len> arr = {1,2,3,4}; // 非法，数组大小参数必须是常量表达式
```

当我们开始用上了 std::array 时，难免会遇到要将其兼容 C 风格的接口，这里有三种做法：

```
void foo(int *p, int len) {
    return;
}

std::array<int 4> arr = {1,2,3,4};

// C 风格接口传参
// foo(arr, arr.size());           // 非法，无法隐式转换
foo(&arr[0], arr.size());
foo(arr.data(), arr.size());

// 使用 `std::sort`
std::sort(arr.begin(), arr.end());
```

2. std::forward_list

std::forward_list 是一个列表容器，使用方法和 std::list 基本类似。 和 std::list 的双向链表的实现不同，std::forward_list 使用单向链表进行实现，提供了 O(1) 复杂度的元素插入，不支持快速随机访问（这也是链表

的特点），也是标准库容器中唯一一个不提供 size() 方法的容器。当不需要双向迭代时，具有比 std::list 更高的空间利用率。

3. 无序容器

C++11 引入了两组无序容器： std::unordered_map/std::unordered_multimap 和 std::unordered_set/std::unordered_multiset。

无序容器中的元素是不进行排序的，内部通过 Hash 表实现，插入和搜索元素的平均复杂度为 O(constant)。

4. 元组 std::tuple

元组的使用有三个核心的函数：

std::make_tuple: 构造元组 std::get: 获得元组某个位置的值 std::tie: 元组拆包

```
#include <tuple>
#include <iostream>

auto get_student(int id)
{
    // 返回类型被推断为 std::tuple<double, char, std::string>
    if (id == 0)
        return std::make_tuple(3.8, 'A', "张三");
    if (id == 1)
        return std::make_tuple(2.9, 'C', "李四");
    if (id == 2)
        return std::make_tuple(1.7, 'D', "王五");
    return std::make_tuple(0.0, 'D', "null");
    // 如果只写 0 会出现推断错误，编译失败
}

int main()
{
    auto student = get_student(0);
    std::cout << "ID: 0, "
    << "GPA: " << std::get<0>(student) << ", "
    << "成绩: " << std::get<1>(student) << ", "
    << "姓名: " << std::get<2>(student) << '\n';

    double gpa;
    char grade;
    std::string name;

    // 元组进行拆包
    std::tie(gpa, grade, name) = get_student(1);
    std::cout << "ID: 1, "
    << "GPA: " << gpa << ", "
    << "成绩: " << grade << ", "
    << "姓名: " << name << '\n';
```

合并两个元组，可以通过 std::tuple_cat 来实现。

```
auto new_tuple = std::tuple_cat(get_student(1), std::move(t));
```

9. 正则表达式

正则表达式描述了一种字符串匹配的模式。一般使用正则表达式主要是实现下面三个需求：

1. 检查一个串是否包含某种形式的子串；
2. 将匹配的子串替换；
3. 从某个串中取出符合条件的子串。

C++11 提供的正则表达式库操作 std::string 对象，对模式 std::regex (本质是 std::basic_regex)进行初始化，通过 std::regex_match 进行匹配，从而产生 std::smatch （本质是 std::match_results 对象）。

我们通过一个简单的例子来简单介绍这个库的使用。考虑下面的正则表达式：

[a-z]+.txt: 在这个正则表达式中, [a-z] 表示匹配一个小写字母, + 可以使前面的表达式匹配多次，因此 [a-z]+ 能够匹配一个及以上小写字母组成的字符串。在正则表达式中一个 . 表示匹配任意字符，而 . 转义后则表示匹配字符 . ，最后的 txt 表示严格匹配 txt 这三个字母。因此这个正则表达式的所要匹配的内容就是文件名为纯小写字母的文本文件。 std::regex_match 用于匹配字符串和正则表达式，有很多不同的重载形式。最简单的一个形式就是传入std::string 以及一个 std::regex 进行匹配，当匹配成功时，会返回 true，否则返回 false。例如：

```
#include <iostream>
#include <string>
#include <regex>

int main() {
    std::string fnames[] = {"foo.txt", "bar.txt", "test", "a0.txt", "AAA.txt"};
    // 在 C++ 中 `\` 会被作为字符串内的转义符，为使 `\.` 作为正则表达式传递进去生效，需
要对 `\` 进行二次转义，从而有 `\\.`
    std::regex txt_regex("[a-z]+\\.txt");
    for (const auto &fname: fnames)
        std::cout << fname << ": " << std::regex_match(fname, txt_regex) <<
std::endl;
}
```

另一种常用的形式就是依次传入 std::string/std::smatch/std::regex 三个参数，其中 std::smatch 的本质其实是 std::match_results，在标准库中， std::smatch 被定义为了 std::match_results，也就是一个子串迭代器类型的 match_results。使用 std::smatch 可以方便的对匹配的结果进行获取，例如：

```
std::regex base_regex("([a-z]+)\\.txt");
std::smatch base_match;
for(const auto &fname: fnames) {
    if (std::regex_match(fname, base_match, base_regex)) {
        // sub_match 的第一个元素匹配整个字符串
        // sub_match 的第二个元素匹配了第一个括号表达式
```

```
            if (base_match.size() == 2) {
                std::string base = base_match[1].str();
                std::cout << "sub-match[0]: " << base_match[0].str() << std::endl;
                std::cout << fname << " sub-match[1]: " << base << std::endl;
            }
        }
    }
```

以上两个代码段的输出结果为：

```
foo.txt: 1
bar.txt: 1
test: 0
a0.txt: 0
AAA.txt: 0
sub-match[0]: foo.txt
foo.txt sub-match[1]: foo
sub-match[0]: bar.txt
bar.txt sub-match[1]: bar
```

10. 语言级线程支持

std::thread std::mutex/std::unique_lock std::future/std::packaged_task std::condition_variable

代码编译需要使用 -pthread 选项

11. 右值引用和move语义

右值引用

先看一个简单的例子直观感受下：

```
string a(x);                                    // line 1
string b(x + y);                                    // line 2
string c(some_function_returning_a_string());      // line 3
```

如果使用以下拷贝构造函数：

```
string(const string& that)
{
    size_t size = strlen(that.data) + 1;
    data = new char[size];
    memcpy(data, that.data, size);
}
```

以上3行中，只有第一行(line 1)的x深度拷贝是有必要的，因为我们可能会在后边用到x，x是一个左值(lvalues)。

第二行和第三行的参数则是右值，因为表达式产生的string对象是匿名对象，之后没有办法再使用了。

C++ 11引入了一种新的机制叫做“**右值引用**”，以便我们通过重载直接使用右值参数。我们所要做的就是写一个以右值引用为参数的构造函数：

```
string(string&& that)   // string&& is an rvalue reference to a string
{
data = that.data;
that.data = 0;
}
```

我们没有深度拷贝堆内存中的数据，而是仅仅复制了指针，并把源对象的指针置空。事实上，**我们“偷取”了属于源对象的内存数据。**由于源对象是一个右值，不会再被使用，因此客户并不会觉察到源对象被改变了。在这里，**我们并没有真正的复制**，所以我们把这个构造函数叫做“**转移构造函数**”（move constructor），**他的工作就是把资源从一个对象转移到另一个对象，而不是复制他们。**

有了右值引用，再来看看赋值操作符：

```
string& operator=(string that)
{
std::swap(data, that.data);
return *this;
}
```

注意到我们是直接对参数that传值，所以that会像其他任何对象一样被初始化，那么确切的说，that是怎样被初始化的呢？对于C++ 98，答案是复制构造函数，**但是对于C++ 11，编译器会依据参数是左值还是右值在复制构造函数和转移构造函数间进行选择。**

如果是a=b，这样就会调用复制构造函数来初始化that（因为b是左值），赋值操作符会与新创建的对象交换数据，深度拷贝。这就是copy and swap 惯用法的定义：构造一个副本，与副本交换数据，并让副本在作用域内自动销毁。这里也一样。

如果是a = x + y，这样就会调用转移构造函数来初始化that（因为x+y是右值），所以这里没有深度拷贝，只有高效的数据转移。相对于参数，that依然是一个独立的对象，但是他的构造函数是无用的（trivial），因此堆中的数据没有必要复制，而仅仅是转移。没有必要复制他，因为x+y是右值，再次，从右值指向的对象中转移是没有问题的。

总结一下：复制构造函数执行的是深度拷贝，因为源对象本身必须不能被改变。而转移构造函数却可以复制指针，把源对象的指针置空，这种形式下，这是安全的，因为用户不可能再使用这个对象了。

下面我们进一步讨论右值引用和move语义。

C++98标准库中提供了一种唯一拥有性的智能指针std::auto_ptr，该类型在C++11中已被废弃，因为其“复制”行为是危险的。

```
auto_ptr<Shape> a(new Triangle);
auto_ptr<Shape> b(a);
```

注意b是怎样使用a进行初始化的，它不复制triangle，而是把triangle的所有权从a传递给了b，也可以说成“a 被转移进了b”或者“triangle被从a转移到了b”。

auto_ptr 的复制构造函数可能看起来像这样（简化）：

```
auto_ptr(auto_ptr& source)   // note the missing const
{
p = source.p;
source.p = 0;   // now the source no longer owns the object
}
```

auto_ptr 的危险之处在于看上去应该是复制，但实际上确是转移。调用被转移过的auto_ptr 的成员函数将会导致不可预知的后果。所以你必须非常谨慎的使用auto_ptr ，如果他被转移过。

```
auto_ptr<Shape> make_triangle()
{
    return auto_ptr<Shape>(new Triangle);
}

auto_ptr<Shape> c(make_triangle());      // move temporary into c
double area = make_triangle()->area();   // perfectly safe

auto_ptr<Shape> a(new Triangle);    // create triangle
auto_ptr<Shape> b(a);               // move a into b
double area = a->area();                // undefined behavior
```

显然，在持有auto_ptr 对象的a表达式和持有调用函数返回的auto_ptr值类型的make_triangle()表达式之间一定有一些潜在的区别，每调用一次后者就会创建一个新的auto_ptr对象。这里a 其实就是一个左值（lvalue）的例子，而make_triangle()就是右值（rvalue）的例子。

转移像a这样的左值是非常危险的，因为我们可能调用a的成员函数，这会导致不可预知的行为。另一方面，转移像make_triangle()这样的右值却是非常安全的，因为复制构造函数之后，我们不能再使用这个临时对象了，因为这个转移后的临时对象会在下一行之前销毁掉。

我们现在知道转移左值是十分危险的，但是转移右值却是很安全的。如果C++能从语言级别支持区分左值和右值参数，我就可以完全杜绝对左值转移，或者把转移左值在调用的时候暴露出来，以使我们不会不经意的转移左值。

C++ 11对这个问题的答案是右值引用。右值引用是针对右值的新的引用类型，语法是X&&。以前的老的引用类型X& 现在被称作左值引用。

使用右值引用X&&作为参数的最有用的函数之一就是转移构造函数X::X(X&& source)，它的主要作用是把源对象的本地资源转移给当前对象。

C++ 11中，`std::auto_ptr< T >`已经被`std::unique_ptr< T >`所取代，后者就是利用的右值引用。

其转移构造函数：

```
unique_ptr(unique_ptr&& source)   // note the rvalue reference
{
    ptr = source.ptr;
    source.ptr = nullptr;
}
```

这个转移构造函数跟auto_ptr中复制构造函数做的事情一样，但是它却只能接受右值作为参数。

```
unique_ptr<Shape> a(new Triangle);
unique_ptr<Shape> b(a);                 // error
unique_ptr<Shape> c(make_triangle());       // okay
```

第二行不能编译通过，因为a是左值，但是参数unique_ptr&& source只能接受右值，这正是我们所需要的，杜绝危险的隐式转移。第三行编译没有问题，因为make_triangle()是右值，转移构造函数会将临时对象的所有权转移给对象c，这正是我们需要的。

转移左值&&move语义

有时候，我们可能想转移左值，也就是说，有时候我们想让编译器把左值当作右值对待，以便能使用转移构造函数，即便这有点不安全。出于这个目的，C++ 11在标准库的头文件< utility >中提供了一个模板函数**std::move**。实际上，**std::move仅仅是简单地将左值转换为右值，它本身并没有转移任何东西。它仅仅是让对象可以转移。**

以下是如何正确的转移左值：

```
unique_ptr<Shape> a(new Triangle);
unique_ptr<Shape> b(a);              // still an error
unique_ptr<Shape> c(std::move(a));   // okay
```

请注意，第三行之后，a不再拥有Triangle对象。不过这没有关系，因为通过明确的写出std::move(a)，我们很清楚我们的意图：亲爱的转移构造函数，你可以对a做任何想要做的事情来初始化c；我不再需要a了，对于a，您请自便。

当然，如果你在使用了mova(a)之后，还继续使用a，那无疑是搬起石头砸自己的脚，还是会导致严重的运行错误。

总之，std::move(some_lvalue)将左值转换为右值（可以理解为一种类型转换），使接下来的转移成为可能。

一个例子：

```
class Foo
{
    unique_ptr<Shape> member;

public:
```

```
    Foo(unique_ptr<Shape>&& parameter)
    : member(parameter)   // error
    {}
};
```

上面的parameter，其类型是一个右值引用，只能说明parameter是指向右值的引用，而parameter本身是个左值。 (Things that are declared as rvalue reference can be lvalues or rvalues. The distinguishing criterion is: if it has a name, then it is an lvalue. Otherwise, it is an rvalue.)

因此以上对parameter的转移是不允许的，需要使用std::move来显示转换成右值。 参考资料：C++11常用新特性快速一览_jiange_zh的博客-CSDN博客_c++11新特性

总结：

1. nullptr代替有二义性的NULL代表空指针
2. C++11 引入了 auto (对变量)和 decltype(对表达式) 这两个关键字实现了类型推导，让编译器来操心变量的类型。
3. 引入了区间迭代 `for(auto &i : arr) { std::cout << i << std::endl; }`
4. 初始化列表，C++11 提供了统一的语法来初始化任意的对象，`B(int _a, float _b): a(_a), b(_b) {}`,C++11 还把初始化列表的概念绑定到了类型上,并将其称之为 std::initializer_list，允许构造函数或其他函数像参数一样使用初始化列表，这就为类对象的初始化与普通数组和 POD 的初始化方法提供了统一的桥梁
5. 模板增强，引入了**外部模板**，扩充了原来的强制编译器在特定位置实例化模板的语法，使得能够显式的告诉编译器何时进行模板的实例化，减少编译时间；**多个尖括号合法，类型别名模板**，在传统 C++中，typedef 可以为类型定义一个新的名称，但是却没有办法为模板定义一个新的名称。因为，模板不是类型。新特性中可以实现。
6. C++11引入了委托构造的概念，这使得构造函数可以在同一个类中一个构造函数调用另一个构造函数，从而达到简化代码的目的;还引入了继承构造
7. lambda表达式，类似一个匿名函数;大部分STL算法，可以非常灵活地搭配lambda表达式来实现想要的效果。
8. 新增容器,std::array保存在栈内存中，相比堆内存中的std::vector，访问更灵活，性能更高，std::array 会在编译时创建一个固定大小的数组，std::array 不能够被隐式的转换成指针，使用 std::array只需指定其类型和大小即可;std::forward_list 是一个列表容器，使用方法和 std::list 基本类似。实现使用单向链表，list使用双向链表，提供O(1)的复杂度元素插入，不支持快速随机访问，不提供size()方法，当不需要双向迭代时，具有比std::list更高的空间利用率;无序容器，std::unordered_map/std::unordered_multimap 和 std::unordered_set/std::unordered_multiset。元素不进行排序，内部通过hash表实现，插入和搜索元素的平均复杂度为O(n);元组tuple
9. 正则表达式,检查一个串是否包含某种形式的子串;将匹配的子串替换;从某个串中取出符合条件的子串。
10. 语言级线程支持，代码编译需要使用 -pthread 选项
11. 右值引用和MOVE语义，拷贝构造函数分为复制构造函数和转移构造函数，转移构造函数没有深度拷贝堆内存中的数据，而是仅仅复制了指针，并把源对象的指针置空。由于源对象是一个右值，不会再被使用，因此客户并不会觉察到源对象被改变了。在这里，我们并没有真正的复制，所以我们把这个构造函数叫做"转移构造函数"（move constructor），他的工作就是把资源从一个对象转移到另一个对象，而不是复制他们。C++新特性中，**对于C++ 11，编译器会依据参数是左值还是右值在复制构造函数和转移构造函数间进行选择。**总结来说，**复制构造函数执行的是深度拷贝，因为源对象本身必须不能被改变。而转移构造函数却可以复制指针，把源对象的指针置空，这种形式下，这是安全的，因为用户不可能再使**

用这个对象了。旧的auto_ptr指针存在将左值进行看起来是复制实际是转移的操作，容易引起未知的后果。已知，转移左值是十分危险的，但是转移右值却是很安全的。C++11考虑进行右值引用，auto_ptr也变成了unique_ptr,unique_ptr使用的就是右值引用，它和auto_ptr很像，但是只接受右值作为参数。那要想转移左值，就需要用到函数move(),将左值转化为右值，让左值对象变成可转移的右值对象。之后被move转移的对象不能够再使用。