1.请问C++11有哪些新特性?

参考网址

1、新增基于范围的for循环

2、自动类型推断 auto

3、匿名函数 Lambda

4、后置返回类型（tailng-return-type）

5、显示重写(覆盖)override和final

6、空指针常量 nullptr

7、long long int类型

8、模板的别名

9、允许sizeof运算符可以再类型数据成员上使用，无需明确对象。

10、线程支持

11、元组类型

简介

C++11,之前被称作C++0x，即ISO/IEC 14882:2011，是目前的C++编程语言的正式标准。它取代第二版标准ISO/IEC 14882:2003(第一版ISO/IEC 14882:1998发布于1998年，第二版于2003年发布，分别通称C++98以及C++03，两者差异很小)。新的标准包含了几个核心语言增加的新特性，而且扩展C++标准程序库，并入了大部分的C++ Technical Report 1程序库(数学的特殊函数除外)。最新的消息被公布在 ISO C++ 委员会网站(英文)。 ISO／IEC JTC1/SC22/WG21 C++ 标准委员会计划在2010年8月之前完成对最终委员会草案的投票，以及于2011年3月召开的标准会议完成国际标准的最终草案。然而，WG21预期ISO将要花费六个月到一年的时间才能正式发布新的C++标准。为了能够如期完成，委员会决定致力于直至2006年为止的提案，忽略新的提案。最终,于2011年8月12日公布，并于2011年9月出版。2012年2月28日的国际标准草案(N3376)是最接近于现行标准的草案，差异仅有编辑上的修正。

本文主要罗列C++11相比较旧版本有重大改变并且我们今后可能会频繁接触的一些功能。

1、新增基于范围的for循环 类似Java中foreach语句，为遍历数组提供了很大方便。

```
int nArr[5] = {1,2,3,4,5};
for(int &x : nArr)
{
    x *=2;   //数组中每个元素倍乘
}
```

2、自动类型推断 auto 它的作用就是当编译器在一个变量声明的时候，能够根据变量赋的值推断该变量的数据类型。这样就有些逼近Python中定义变量的功能，无需提前声明定义的变量的数据类型。例如：

```
auto i = 1;    //编译器自动推断i为int类型
```

这个功能在各种标准模板库容器中使用，作用更突出。如下：

```
vector<int> vec(6,10);
vector<int>::iterator iter = vec.iterator();
auto iterAuto = vec.iterator(); //相比较上一句方便很多
```

3、匿名函数 Lambda　　如果代码里面存在大量的小函数，而这些函数一般只被一两处调用，那么不妨将它们重构成Lambda表达式，也就是匿名函数。作用就是当你想用一个函数，但是又不想费神去命名一个函数。

该功能函数实际上在其他面向对象语言中早就存在，例如Java，Python都定义了该功能。C++中Lambda表达式格式如下：

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

① [capture]指定在可见域范围内lambda表达式代码内可见的参数。例如：

- [a, &b]，前文定义的a以值方式被表达式捕获，b则是以引用的方式；
- [this] 以值的方式捕获 this 指针。
- [&] 以引用的方式捕获所有的外部自动变量。
- [=] 以值的方式捕获所有的外部自动变量。
- [] 不捕获外部的任何变量。
  【注】：const 类型的 lambda 表达式，该类型的表达式不能改捕获(“capture”)列表中的值。
  ② (params)指定lambda表达式内部变量定义。
  ③ ->ret是返回类型，如果 lambda 代码块中包含了 return 语句，则该 lambda 表达式的返回类型由 return 语句的返回类型确定。如果没有 return 语句，则类似 void f(...) 函数。
  ④ {body}是Lambda表达式主题结构。
  例句：

```
auto func = [](int i){ return i+4};// 可以体会auto的好处了
cout<< func(10) << endl;           //输出为14
```

4、后置返回类型（tailng-return-type）

```
 template Ret adding_func(const Lhs &lhs, const Rhs &rhs) {return lhs + rhs;}
```

这不是合法的C++，因为lhs和rhs还没定义；解析器解析完函数原型的剩余部分之前,它们还不是有效的标识符。
　　为此, C++11引入了一种新的函数声明语法，叫做后置返回类型(trailing-return-type)。

```
template auto adding_func(const Lhs &lhs, const Rhs &rhs) -> decltype(lhs+rhs) {return lhs +
rhs;}
```

这种语法可以用到更普通的函数声明和定义上：

```
struct SomeStruct  {
    auto func_name(int x, int y) -> int;
};
auto SomeStruct::func_name(int x, int y) -> int {
    return x + y;
}
```

关键字auto的这种用法与在自动类型推导中有所不同。

5、显示重写(覆盖)override和final

在C++03中，很容易让你在本想重写基类某个函数的时候却意外地创建了另一个虚函数。例如：

```
struct Base {
    virtual void some_func(float);
};
struct Derived : Base {
    virtual void some_func(int);
};
```

本来Derived::some_func函数是想替代Base中那个函数的。但是因它的接口不同，又创建了一个虚函数。这是个常见的问题，特别是当用户想要修改基类的时候。　　C++11引入了新的语法来解决这个问题：

```
struct Base {
    virtual void some_func(float);
};
struct Derived : Base {
    virtual void some_func(int) override; // 病态的,不会重写基类的方法
};
```

override 这个特殊的标识符意味编译器将去检查基类中有没有一个具有相同签名的虚函数，如果没有，编译器就会报错！　　C++11还增加了防止基类被继承和防止子类重写函数的能力。这是由特殊的标识符final来完成的，例如：

```
struct Base1 final { };

struct Derived1 : Base1 { }; // 病态的, 因为类Base1被标记为final了

struct Base2 {
    virtual void f() final;
};

struct Derived2 : Base2 {
    void f(); // 病态的, 因为虚函数Base2::f 被标记为final了.
};
```

在这个例子中, virtual void f() final；语句声明了一个虚函数却也阻止了子类重写这个函数。它还有一个作用，就是防止了子类将那个特殊的函数名与新的参数组合在一起。

需要注意的是，override和final都不是C++语言的关键字。他们是技术上的标识符，只有在它们被用在上面这些特定的上下文在才有特殊意义。用在其它地方他们仍然是有效标识符。

6、空指针常量 nullptr　　NULL通常在C语言中预处理宏定义为(void*)0或者0，这样0就有int型常量和空指针的双重身份。但是C++03中只允许0宏定义为空指针常量，这就会造成如下的错误：

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

上面声明了两个重载函数，当我调用foo(NULL)，编译器将会调用foo(int)函数，而实际上我是想调用foo(char*)函数的。为了避免这个歧义，C++11重新定义了一个新关键字nullptr，充当单独空指针常量。

7、long long int类型　　C++03中，最大的整数类型是long int。它保证使用的位数至少与int一样。 这导致long int在一些实现是64位的，而在另一些实现上却是32位的。C++11增加了一个新的整数类型long long int来弥补这个缺陷。它保证至少与long int一样大,并且不少于64位。这个类型早在C99就引入到了标准C中， 而且大多数C++编译器都以扩展的形式支持这种类型了。　　8、模板的别名　　在进入这个主题前,先弄清楚“模板”和“类型”的区别。类型，是具体的数据类型，可以直接用来定义变量。 模板，是类型的模板，根据这个模板可以产生具体的类型；模板是不能直接定义变量的；当指定了所有的模板参数后，就产生了一个具体的类型，就可以用来定义变量了。

在C++03中，只能为类型（包括完全特化的模板，也是一种类型）定义别名，而不能为模板定义别名：

```
template <typename First, typename Second, int Third>
class SomeType;
template <typename Second>
typedef SomeType<OtherType, Second, 5> TypedefName; // 在C++03中, 这是非法的.
```

C++11增加为模板定义别名的能力,用下面这样的语法：

```
template <typename First, typename Second, int Third>
class SomeType;
template <typename Second>
using TypedefName = SomeType<OtherType, Second, 5>;
//这种using语法也可以用来定义类型的别名:
typedef void (*FunctionType)(double);  // 老式语法
using FunctionType = void (*)(double); // 新式语法
```

9、允许sizeof运算符可以再类型数据成员上使用，无需明确对象。

```
struct p {otherClass member;};
sizeof(p::member);
```

10、线程支持　　C++11虽然从语言上提供了支持线程的内存模型,但主要的支持还是来自标准库。　　新的标准库提供了一个线程类(std::thread)来运行一个新线程，它带有一个函数对象参数和一系列可选的传递给函数对象的参数。通过std::thread::join()支持的线程连接操作可以让一个线程直到另一个线程执行完毕才停止。std:thread::native_handle()成员函数提供了对底层本地线程对象的可能且合理的平台相关的操作。　　为支持线程同步,标准库增加了互斥体(std::mutex, std::recursive_mutex等)和条件变量(std::condition_variable 和 std::condition_variable_any)。这些都是通过RAII锁和加锁算法就可以简单使用的。　　有时为了高性能或底层工

作，要求线程间的通信没有开销巨大的互斥锁。原子操作可以达到这个目的,这可以随意地为一个操作指定最小的内存可见度。显式的内存屏障也可以用于这个目的。　　C++11线程库还包含了futures和promises，用于在线程间传递异步结果。并且提供了std::packaged_task来封装可以产生这种异步结果的函数调用。　　更高级的线程支持，如线程池，已经决定留待在未来的 Technical Report 加入此类支持。更高级的线程支持不会是 C++11 的一部份，但是其最终实现将建立在目前已有的线程支持之上。std::async 提供了一个简便方法来运行线程，并将线程绑定在 std::future上。用户可以选择一个工作是要在多个线程上异步的运行，还是在一个线程上运行并等待其所需要的数据。默认的情况，实现可以根据底层硬件选择前面两个选项的其中之一。另外在较简单的使用场景下，实现也可以利用线程池提供支持。　　11、元组类型　　元组（tuple）由预先确定数量的多种对象组成，元组可以看作是struct数据成员的泛化，在Python中是一个基本数据结构。TR1 tuple类型的C++11版本获益于像可变参数模板这样的C++11语言特性。TR1版本的元组需要一个由实现定义的包含的类型的最大数目，而且需要大量的宏技巧来实现。相比之下，C++11版本的不需要显式的实现定义的最大类型数目。尽管编译器有一个内部的模板实例化的最大递归深度，但C++11版的元组不会把它暴露给用户。

用可变参数模板,元组类的定义看上去像下面这样：

```
template <class ...Types> class tuple;
//下面是定义和使用元组的一个例子：
typedef std::tuple <int, double, long &, const char *> test_tuple;
long lengthy = 12;
test_tuple proof (18, 6.5, lengthy, "Ciao!");
lengthy = std::get<0>(proof);  // 把'lengthy' 赋值为18.
std::get<3>(proof) = " Beautiful!";  // 修改元组的第四个元素
```

其它新增标准程序库

正则化表达式库；字符串类字新增与其他类型互换的方法，如to_string()，stoi()，stol等；STL标准模板库新增unordered_map以及unordered_set，基于hash表的关联容器等等。这些详细使用可自行上网查找，也十分重要的知识点。

2、请你详细介绍一下C++11中的可变参数模板、右值引用和lambda这几个新特性。

可变参数模板

C++11增加了可变参数模板，可以接受可变数目，类型的参数。我们通过开发中常用的输出调试信息的例子介绍可变参数模板的使用方法。首先是声明可变参数模板。

```
template<typename T, typename... Args>
void writeLog(const T&t, const Args... rest);
```


在模板参数定义的部分，通过typename...定义可变模板参数。三个点的含义和C语言中的可变参数定义类似，不同的是后面接着指定了可变参数列表的名称Args。

在参数定义的部分，使用可变模板参数定义的Args以相同的格式定义函数的可变参数列表。

这个参数列表可以在模板函数的实现中直接使用，这样在定义可变参数模板时就解决了可变参数函数的第二个难点。

可变参数模板的实现

可变参数模板的实现通常需要一些小技巧：递归和重载。还是先看代码。

```
template<typename T>
void writeLog(const T& t){
    cout << t;
}

template<typename T, typename... Args>
void writeLog(const T&t, const Args... rest){
    writeLog(t);
    writeLog(rest...);
}
```


代码中定义了两个重载的writeLog函数，一个只接受类型为T的参数t，另一个除了t之外，还接受可变参数rest。当使用一个参数调用writeLog的时候，实际调用上面的函数；当使用多个参数调用writeLog的时候调用下面的writeLog。

下面的writeLog首先使用第一个参数t调用上面的writeLog之后，使用rest递归调用writeLog（严格讲是rest中有多于一个参数的 时候）。从调用者来看，每次处理一个参数之后，使用其余的参数再次调用writeLog，直到最后调用一个参数的writeLog。

下面是使用writeLog的示例代码:

```
int main()
{
    //单个参数输出
    writeLog(5);
    writeLog(",");
    writeLog(6);
    writeLog("\n");

    //任意参数自由组合输出
    int x = 5;
    int y = 6;
    writeLog("x=", x, ",y=",6, ",x+y=[", y + x, "].\n");

    return 0;
}
```


可以任意组合参数的类型和个数，而且不需要另外提供任何信息。这样就有效地解决了可变参数函数的第一个难点。

更近一步

本例中一个参数的writeLog非常简单，只是简单的使用cout进行输出，如果有特殊的需求，可以继续重载这个函数。例如log输出中经常需要的时间信息，就可以这样实现：

```
class LogTime
{
public:
    LogTime(){
        time(&t);
    }
    void output(){
        tm* p = localtime(&t);
        cout << p->tm_hour << ":" << p->tm_min << ":" << p->tm_sec;
    }
    time_t t;
};

void writeLog(LogTime t){
    t.output();
}
```


使用方法更加简单：

```
    LogTime lt;
    writeLog(lt, " ", "How are you!", 24, "\n");
```


输出结果

```
5,6
x=5,y=6,x+y=[11].
20:16:7 How are you!24
```


右值引用见　面向对象与泛型编程

lambad

详细参考网址

C++11的一大亮点就是引入了Lambda表达式。利用Lambda表达式，可以方便的定义和创建匿名函数。对于C++这门语言来说来说，“Lambda表达式”或“匿名函数”这些概念听起来好像很深奥，但很多高级语言在很早以前就已经提供了Lambda表达式的功能，如C#，Python等。今天，我们就来简单介绍一下C++中Lambda表达式的简单使用。

声明Lambda表达式

Lambda表达式完整的声明格式如下：

```
[capture list] (params list) mutable exception-> return type { function body }
```

各项具体含义如下

1. capture list：捕获外部变量列表
2. params list：形参列表
3. mutable指示符：用来说用是否可以修改捕获的变量
4. exception：异常设定

5. return type：返回类型
6. function body：函数体

此外，我们还可以省略其中的某些成分来声明“不完整”的Lambda表达式，常见的有以下几种：

序号	格式
1	[capture list] (params list) -> return type {function body}
2	[capture list] (params list) {function body}
3	[capture list] {function body}

其中：

- 格式1声明了const类型的表达式，这种类型的表达式不能修改捕获列表中的值。
- 格式2省略了返回值类型，但编译器可以根据以下规则推断出Lambda表达式的返回类型：（1）：如果function body中存在return语句，则该Lambda表达式的返回类型由return语句的返回类型确定；（2）：如果function body中没有return语句，则返回值为void类型。
- 格式3中省略了参数列表，类似普通函数中的无参函数。

  讲了这么多，我们还没有看到Lambda表达式的庐山真面目，下面我们就举一个实例。

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool cmp(int a, int b)
{
    return  a < b;
}

int main()
{
    vector<int> myvec{ 3, 2, 5, 7, 3, 2 };
    vector<int> lbvec(myvec);

    sort(myvec.begin(), myvec.end(), cmp); // 旧式做法
    cout << "predicate function:" << endl;
    for (int it : myvec)
        cout << it << ' ';
    cout << endl;

    sort(lbvec.begin(), lbvec.end(), [](int a, int b) -> bool { return a < b; });   // Lambda表
达式
    cout << "lambda expression:" << endl;
    for (int it : lbvec)
        cout << it << ' ';
}
```

在C++11之前，我们使用STL的sort函数，需要提供一个谓词函数。如果使用C++11的Lambda表达式，我们只需要传入一个匿名函数即可，方便简洁，而且代码的可读性也比旧式的做法好多了。

下面，我们就重点介绍一下Lambda表达式各项的具体用法。

捕获外部变量

Lambda表达式可以使用其可见范围内的外部变量，但必须明确声明（明确声明哪些外部变量可以被该Lambda表达式使用）。那么，在哪里指定这些外部变量呢？Lambda表达式通过在最前面的方括号[]来明确指明其内部可以访问的外部变量，这一过程也称过Lambda表达式“捕获”了外部变量。

我们通过一个例子来直观地说明一下：

```
#include <iostream>
using namespace std;

int main()
{
    int a = 123;
    auto f = [a] { cout << a << endl; };
    f(); // 输出：123

    //或通过“函数体”后面的‘()’传入参数
    auto x = [](int a){cout << a << endl;}(123);
}
```

上面这个例子先声明了一个整型变量a，然后再创建Lambda表达式，该表达式“捕获”了a变量，这样在Lambda表达式函数体中就可以获得该变量的值。

类似参数传递方式（值传递、引入传递、指针传递），在Lambda表达式中，外部变量的捕获方式也有值捕获、引用捕获、隐式捕获。

1、值捕获

值捕获和参数传递中的值传递类似，被捕获的变量的值在Lambda表达式创建时通过值拷贝的方式传入，因此随后对该变量的修改不会影响影响Lambda表达式中的值。

示例如下：

```
int main()
{
    int a = 123;
    auto f = [a] { cout << a << endl; };
    a = 321;
    f(); // 输出：123
}
```

这里需要注意的是，如果以传值方式捕获外部变量，则在Lambda表达式函数体中不能修改该外部变量的值。

2、引用捕获

使用引用捕获一个外部变量，只需要在捕获列表变量前面加上一个引用说明符&。如下：

```
int main()
{
    int a = 123;
    auto f = [&a] { cout << a << endl; };
    a = 321;
    f(); // 输出: 321
}
```

从示例中可以看出，引用捕获的变量使用的实际上就是该引用所绑定的对象。

3、隐式捕获

上面的值捕获和引用捕获都需要我们在捕获列表中显示列出Lambda表达式中使用的外部变量。除此之外，我们还可以让编译器根据函数体中的代码来推断需要捕获哪些变量，这种方式称之为隐式捕获。隐式捕获有两种方式，分别是[=]和[&]。[=]表示以值捕获的方式捕获外部变量，[&]表示以引用捕获的方式捕获外部变量。

隐式值捕获示例：

```
int main()
{
    int a = 123;
    auto f = [=] { cout << a << endl; };    // 值捕获
    f(); // 输出: 123
}
```

隐式引用捕获示例：

```
int main()
{
    int a = 123;
    auto f = [&] { cout << a << endl; };    // 引用捕获
    a = 321;
    f(); // 输出: 321
}
```

4、混合方式

上面的例子，要么是值捕获，要么是引用捕获，Lambda表达式还支持混合的方式捕获外部变量，这种方式主要是以上几种捕获方式的组合使用。

到这里，我们来总结一下：C++11中的Lambda表达式捕获外部变量主要有以下形式：

捕获形式	说明
[]	不捕获任何外部变量
[变量名, ...]	默认以值得形式捕获指定的多个外部变量（用逗号分隔），如果引用捕获，需要显示声明（使用&说明符）
[this]	以值的形式捕获this指针
[=]	以值的形式捕获所有外部变量
[&]	以引用形式捕获所有外部变量
[=, &x]	变量x以引用形式捕获，其余变量以传值形式捕获
[&, x]	变量x以值的形式捕获，其余变量以引用形式捕获

修改捕获变量

前面我们提到过，在Lambda表达式中，如果以传值方式捕获外部变量，则函数体中不能修改该外部变量，否则会引发编译错误。那么有没有办法可以修改值捕获的外部变量呢？这是就需要使用mutable关键字，该关键字用以说明表达式体内的代码可以修改值捕获的变量，示例：

```
int main()
{
    int a = 123;
    auto f = [a]()mutable { cout << ++a; }; // 不会报错
    cout << a << endl; // 输出: 123
    f(); // 输出: 124
}
```

Lambda表达式的参数

Lambda表达式的参数和普通函数的参数类似，那么这里为什么还要拿出来说一下呢？原因是在Lambda表达式中传递参数还有一些限制，主要有以下几点：

1. 参数列表中不能有默认参数
2. 不支持可变参数
3. 所有参数必须有参数名

常用举例：

```
  {
       int m = [](int x) { return [](int y) { return y * 2; }(x)+6; }(5);
       std::cout << "m:" << m << std::endl;                //输出m:16

       std::cout << "n:" << [](int x, int y) { return x + y; }(5, 4) << std::endl;
//输出n:9

       auto gFunc = [](int x) -> function<int(int)> { return [=](int y) { return x + y; }; };
       auto lFunc = gFunc(4);
       std::cout << lFunc(5) << std::endl;
```

```
        auto hFunc = [](const function<int(int)>& f, int z) { return f(z) + 1; };
        auto a = hFunc(gFunc(7), 8);

        int a = 111, b = 222;
        auto func = [=, &b]()mutable { a = 22; b = 333; std::cout << "a:" << a << " b:" << b <<
std::endl; };

        func();
        std::cout << "a:" << a << " b:" << b << std::endl;

        a = 333;
        auto func2 = [=, &a] { a = 444; std::cout << "a:" << a << " b:" << b << std::endl; };
        func2();

        auto func3 = [](int x) ->function<int(int)> { return [=](int y) { return x + y; }; };
        std::function<void(int x)> f_display_42 = [](int x) { print_num(x); };
    f_display_42(44);
   }
```