An Introduction to C++ 1x

Index

Type Inference
类型推导

Type Inference introduction

Python语言中，声明变量的方法与C++并不相同。

如要声明数字a，字符串b:

```
a = 1
b = "Hello World\n"
```

我们不需要写出变量的类型，但实际上变量是有类型的。这种效果的实现正是依赖于类型推导(Type Inference)。

Type Inference auto

C++98中也有auto关键字，但由于几乎不被使用，因此在C++11中被赋予了新的作用。它可以自动推导变量的类型。

```
auto a = 1;
auto b = "Hello World\n";
```

此时编译器编译时可以自动推导出a的类型为int，
推导出b的类型为char*

Type Inference auto

auto的其中一个明显的用处就是简化代码，例如我们需要遍历STL容器时：

```
std::map<std::string, std::string> sample_map;
for (std::map<std::string, std::string>::iterator iter = sample_map.begin();
      iter != sample_map.end(); ++iter) {
   // do something
}
```

如果使用auto：

```
std::map<std::string, std::string> sample_map;
for (auto iter = sample_map.begin(); iter != sample_map.end(); ++iter) {
   // do something
}
```

Type Inference decltype

decltype是另一个使用类型推导的关键字，基本语法为 decltype(value_name) val;

```
int a;
std::string b;
decltype(a) i;
decltype(b) s;
```

如上面的例子，i会推导为int类型，而s会被推导为std::string类型。

Type Inference decltype

这个语法可以应用于复用匿名的struct或union

```
struct {  // 匿名
    int x, y;
} temp_position;

union {  // 匿名
    int *foo;
} temp_union;

decltype(temp_position) pos;
decltype(temp_union) uni;
```

Type Inference auto & decltype

需要注意，auto和decltype都是编译时推导变量的类型，而不能在运行时推导变量类型，而且其推导有一定的限制。例如

```
auto i;  // Compile error
int foo(auto a, auto b);  // Compile error, but it is valid in C++14
int bar(auto a = 1, auto b = 1);  // 虽然我们可能觉得编译器可以推导
                                  // 但这也是不合法的表达
```

详细的限制可参考相关书籍

Type Inference

Return Value Tracking(追踪返回类型)

追踪返回类型的函数和普通函数的声明最大区别在于返回类型的后置。如

```
int func(char* a, int b);
auto func(char* a, int b) -> int;
```

而追踪返回类型还可以让我们在写返回类型时，不写明作用域

```
class Foo {
    struct InnerType { int i; }
    InnerType GetInner();
    InnerType it;
}
auto OuterType::GetInner() -> InnerType { return it; }
```

Type Inference
Return Value Tracking(追踪返回类型)

上面体现的效果并不是特别的明显，那么下面的例子呢？

```
// 有的时候，你会发现这是面试题
//          ——《Understanding C++11 Analysis
//                and Application of New Features》
int (*(*pf()) ()) () { return nullptr; }

auto pf1() -> auto (*) () -> int (*) () { return nullptr; }
```

上述两个函数的作用是完全一样的，但是可读性却完全不同
这其实是一个返回函数指针的函数，而这个指针指向着返回int*的函数

nullptr是C++11的关键字，表示空指针

Type Inference

Extra

- auto的使用细则

  auto*, auto&, const auto, volatile auto...

- decltype的推导四规则

  推导为T还是T&?

Rvalue Reference
右値引用

Review: Rvalue and Lvalue

通常，在C++中，我们称可以使用取地址运算符(&)的值为左值，而不是左值的值就是右值

```
#include <iostream>
int main() {
    int a = 1, b = 3, c = 0;
    c = a + b;
    std::cout << &c << std::endl;  // Valid, c is an Lvalue
    // std::cout << &(a + b) << std::endl;  // Invalid, (a + b) is an Rvalue
    std::cout << &a << " " << &b << std::endl; // Valid, a and b are Lvalue
    return 0;
}
```

Rvalue Reference

右值引用，就是使右值也拥有一个名字，如果它是一个临时变量（如函数的返回值），右值引用还能给它“续命”。

基本语法 value_type && val = temp_val;

```
int foo() {
    return 1;
}

int main() {
    int && a = foo();  // foo()的返回值不会立即销毁
    return 0;
}
```

Rvalue Reference

Move Syntax

```
class MyPool {
    char* pool;
  public:
    MyPool() {
        std::cout << "new" << std::endl;
        pool = new char[1000];
    }
    ~MyPool() { if (pool) delete[] pool; }
    MyPool(const MyPool& oth) {  // copy
        std::cout << "copy" << std::endl;
        std::cout << "new" << std::endl;
        pool = new char[1000];
        memcpy(pool, oth.pool, sizeof(pool));
    }
};
```

Rvalue Reference

Move Syntax

上面的拷贝构造函数就是我们习以为常的深拷贝。

那么下面的代码中，拷贝会进行多少次呢？

(不考虑编译器优化) (g++ -fno-elide-constructors)

```
MyPool foo() { return MyPool(); }

int main() {
    MyPool m_pool(foo());
    return 0;
}
```

答案是两次。return时生成一次临时变量，m_pool构造时再copy一次。

如此的话，new char[1000]就被调用了3次。

Rvalue Reference

Move Syntax

那么下面就是右值引用派上用场的时候了，使用它，可以实现移动语义(Move Syntax)

```
// 在class MyPool中加入另一种版本的构造函数
MyPool(MyPool && oth) {
    std::cout << "move" << std::endl;
    pool = oth.pool;
    oth.pool = nullptr;  // why set it to nullptr?
}
```

这里，没有一次拷贝，且只有一次new。显然其效率提高了不少。

Rvalue Reference

Move Syntax

所谓移动语义，就是从将要销毁的临时变量那里把资源移动过来。这样就节省了分配空间和复制值的开销。这就是移动语义的核心。

而实现了移动语义的构造函数：

MyPool(MyPool && oth);

就称为移动构造函数(Move constructor)

Rvalue Reference
折叠规则

typedef const int T;
typedef T& TR;
TR& v = 1; // 在C++98中导致编译错误

在C++11中，这样的声明不会出错，而会发生折叠。
假定typedef int T;
typedef T& TR;

TR的类型定义	声明变量v的类型	v的实际类型
T&	TR	int&
T&	TR&	int&
T&	TR&&	int&
T&&	TR	int&&
T&&	TR&	int&
T&&	TR&&	int&&

Rvalue Reference
Perfect Forwarding(完美转发)

有了折叠规则，右值引用便有了有趣的效果。

```
template <typename T>
void Forward(T && t) {
   // do something
}
```

这样的写法，使T变成了其原本的类型，完美的转发了参数。即Forward可以接受下列多种参数，而不用多次重载

const int&

int&

int&&

```
// 如果没有完美转发
template <typename T>
void Forward(T& t) { /* do something */ }
template <typename T>
void Forward(const T&t) { /* do something */ }
```

Rvalue Reference
Perfect Forwarding(完美转发)

完美转发依赖于折叠规则。下面给出解释:

```
// perfect forwarding
template <typename T>
void Forward(T && t) {/* do something */}
```

```
// 当T是const int&
void Forward(const int& && t);
// 折叠成
void Forward(const int& t);

// 当T是int& 同上

// 当T是int&&
void Forward(int && &&t);
// 折叠成
void Forward(int &&t);
```

Rvalue Reference
Extra

- std::move() //强制装换为右值
- std::forward() // 转发函数

他们的区别是？

Lambda
匿名函数

Lambda
Functional Programming

```
const double PI = [] (int n) -> double {
    double pi = 4.0 , decimal = 1.0;
    while( n > 2 ) {
       decimal -= ( 1.0 / ( 2.0 * n + 1 ) );
       --n;
       decimal += ( 1.0 / ( 2.0 * n + 1 ) );
       --n;
    }
    if( n > 0 )
       decimal -= ( 1.0 / ( 2.0 * n + 1 ) );
    return( pi * decimla );
 } (100);
```

LambdaLambda Expression

基本语法：

[capture] (parameters) mutable -> return_type { statement }

[capture]: 捕捉列表。能够捕捉上下文的变量供lambda函数使用。

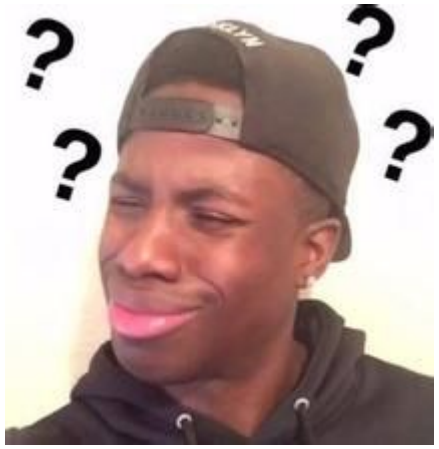
(parameters): 参数列表。

mutable: 默认情况下，lambda函数总是一个const函数，mutable可以取消其常量性。

->return_type: 返回类型。

statement: 函数体。可以使用捕获的变量。

What is Functional Programming?



Lambda Expression

```
// example
int main ( ) {
    int a = 2 ;
    // catch a
    auto pow = [ a ] ( int times ) - > int {
        int ret = 1 ;
        while ( times -- ) ret * = a ; // use a
        return ret ;
    } ;
    std :: cout << pow ( 3 ) << std :: endl ; // 8
    return 0 ;
}
```

More Elegant Code

```
std::list<int> li = { 1,2,3,4,5,6,7,8 };
auto printList = [](std::list<int>& li) {
  for(auto it = li.begin(); it != li.end(); it++) {
    std::cout << *it << " ";
 }
  std::cout << std::endl;
};
printList(li);
// remove all even numbers from the list
li.remove_if([&](int x) -> bool {
    if(x % 2 == 0) {
      return true;
    }
    return false;
});
printList(li);
```

Standard Library With Lambda

```
std :: array < int , 10 > s = { 5 , 7 , 4 , 2 , 8 , 6 , 1 , 9 , 0 , 3 } ;
// sort using a lambda expression
std :: sort ( s. begin ( ) , s. end ( ) , [ ] ( int a, int b ) {
        return b < a ;
} ) ;
```

```
std::thread t([](){
   std::cout << "thread function\n";
});
```

Do you know what this mean?

```
 public:
      std::list<User>
      queryUser
      (std::function<bool(const User&)> filter);
private:
    std::list<User> userList_;
```

What's the output?

```
int x = 4 ;
auto y = [ & r = x, x = x + 1 ] ( ) - > int {
        r + = 2 ;
        return x + 2 ;
} ( ) ;

std :: cout << y << std :: endl ;
```

Closure 函数闭包

What is a closure ?

Closure With Lambda Expression

```
fromto = [ ] ( auto start, auto finish ) {
    return [ = ] ( ) mutable {
      if ( start < finish )
        return start ++ ;
      else
        throw std :: runtime_error ( "Complete" ) ;
    } ;
  } ;

  auto range = fromto ( 0 , 10 ) ;

  std :: cout << range ( ) << std :: endl ;
  std :: cout << range ( ) << std :: endl ;
  std :: cout << range ( ) << std :: endl ;

   std :: cout << range ( ) << std :: endl ;
```

output: 0, 1, 2, 3 (why?)

Closure List

```
auto List = [](auto ...xs) {
    return [=](auto access) { return access(xs...); };
};

auto head = [](auto xs) {
    return xs([](auto first, auto .. .rest) { return first; });
};

auto length = [](auto xs) {
    return xs([](auto .. .z) { return sizeof...(z); });
};

int len = length(List(1, '2', "3", false, "abcde"));  // 5
int head_element = head(List(1, '2', "3"));  // 1
```

Functor(仿函数) V.S. Lambda

What is a functor?

What's the difference between functor and lambda?

Functor

```
class func1 {
  public :
    func1 & operator ( ) ( int x ) {
      std :: cout << x << std :: endl ;
      return * this ;
    }
  } f1 ;

    f1 ( 11 ) ( 12 ) ( 13 ) ;
```

This a functor with casacade calling exapmle

Can you ?

```
Sum(4)(5)   // 9
```

Write this function using lambda expression?

```
Sum(3)(4)(5)   // 12
```

What about this one? Can one lambda function solve the two problem ?

Can this functor work in C++03 ?

```
void func2(std::vector<int>& v) {
  struct {
    void operator()(int) {
       // do something
    }
  } f;
  std::for_each(v.begin(), v.end(), f);
}
```

Functor(仿函数) V.S. Lambda

A lambda is a functor

- just defined with a shorter syntax.

The problem is that lambda syntax is limited.

Initializer
初始化

Initializer

Introduction

C++98中对内置类型如int，使用花括号进行初始化是常见的

```
int a[] = { 1, 2, 3 };
int b[5] = { 0 };
```

然而对于自定义的类型，则无法如此的便捷。

而C++11中则可以使用花括号的列表对变量进行初始化，这种初始化方法就称为初始化列表(Initializer list)

初始化列表一词，还出现在构造函数中，如

```
struct A { int a; float b; A() : a(0), b(0.0f) {} };
```

Initializer

Initializer List

```
#include <vector>
#include <map>
using std::vector;
using std::map;

int a[] = { 1, 3, 5 };        // C++98编译通过，C++11编译通过
int b[] { 1, 3, 5};       // C++98编译失败，C++11编译通过
vector<int> c{1, 3, 5}; // C++98编译失败，C++11编译通过
map<int, float> d =
     { { 1, 1.0f }, { 3, 3.0f }, { 5, 5.0f } };  // C++98失败，C++11通过
```

Initializer

initializer_list

在上面的代码中我们看到了初始化列表的便捷，而这个也并不是标准库类型的“特权”，我们也可以自定义使用初始化列表的类型。

```
enum Gender { boy, girl };
class People {
  public:
    People(initializer_list<pair<string, Gender>> il) {
        for (auto i = il.begin(); i != il.end(); ++i) {
            data[i->first] = i->second;
        }
    }
  private:
    map<string, Gender> data;
}

People class_3 = { { "Big god", boy }, { "Huge god", girl } };
```

Have a look at struct

```
struct s {
  int x;
  std::string std;
  std::vector<int> vec;
};

s func (){
  return {1, "abcde", {2,4,6,8,10}};
}

s s1{1, "abcde", {2,4,6,8,10}};
```

Initializer

应用:防止类型收窄

类型收窄是指一些可以使得数据变化或者精度丢失的隐式类型转换。如浮点型->整型，整型->无法精确表示原整型的浮点型(整型数较大)，double->float，int->char

下面是一些类型收窄的例子

Initializer

应用:防止类型收窄

```
const int x = 1024;

char a = x;       // 收窄，但通过编译
char* b = new char(1024);    // 收窄，但通过编译

char c = {x};          // 收窄，不能通过编译
char d = { 10 };           // 通过编译
unsigned char e { -1 }   // 收窄，不能通过编译

float f { 7 };         // 通过编译，列表中的 7 没有收窄
float* g = new float { 1e48 };   // 收窄，不能通过编译，1e48过大
```

Imediate Initialization

```
class c {
private:
  int x = 1;
  float y = 2.3;
  int z[4] = {1, 2, 3, 4};
  std::vector<int> vec {1,2,3,4,5};
};
```

Can we initialize z with:

int z[] = {1, 2, 3, 4};

Smart Pointer
智能指针

Smart Pointer
内存管理

- 一般的内存问题有 :

- 野指针
- 重复释放
- 内存泄漏

- 使用智能指针可以帮我们有效地减少这些问题

Smart Pointer

auto_ptr(C++98)

auto_ptr以对象的方式管理堆分配的内存，并在适当的时间(比如auto_ptr析构时)，释放获得的堆内存。这种管理方式只要程序员将new返回的指针作为auto_ptr的初始值即可，不需要显式调用delete。比如：

```
auto_ptr(new int);
```

不过auto_ptr有一些缺点，如拷贝时返回一个左值，不能调用delete[]等，在C++11中被废弃。

Smart Pointer

unique_ptr

unique_ptr如其命名，表示它管理一个只属于它的内存，它不能被复制，可以被移动。

```
using namespace std;
unique_ptr<int> up1(new int(11));
unique_ptr<int> up2 = up1;   // 编译错误，复制
unique_ptr<int> up3 = move(up1)  // std::move会使up1清空，
                 // 这句的语义是将up1移动给up3
cout << *up1 << endl;          // 错误，up1已经不拥有该内存
cout << *up3 << endl;          // 11
up3.reset();                 // reset使指针置空并释放内存
up1.reset();                 // 不会导致内存错误
cout << *up3 << endl;          // 错误，内存已释放
// 即使不调用reset，unique_ptr对象在析构时也会释放拥有的内存
```

Smart Pointer

shared_ptr

shared_ptr也如其命名，表示多个智能指针可以拥有同一个堆内存。在实现上使用了引用计数，当一个shared_ptr失效，内存也不会被释放，而是引用计数减小。当引用计数归零的时候，shared_ptr才会真正释放堆内存的空间。

代码例子在讲述完weak_ptr后结合weak_ptr的应用一起放出。

Smart Pointer

weak_ptr

weak_ptr用于指向一个shared_ptr，但它实际上不拥有这个内存。其lock()方法可返回一个shared_ptr，而当所指对象已经失效时，可以返回nullptr。用这个可以验证shared_ptr的有效性。

Smart Pointer
shared_ptr&weak_ptr

```
using namespace std;
shared_ptr<int> sp1(new int(22));
shared_ptr<int> sp2 = sp1;        // 和sp1共享一个堆内存
weak_ptr<int> wp = sp1;       // 指向sp1所指对象
cout << *sp1 << endl;         // 22
cout << *sp2 << endl;         // 22
sp1.reset();                  // 引用计数-1，但不释放内存
if (wp.lock() != nullptr) cout <<  "valid" << endl;   // valid

shared_ptr<int> sp3 = wp.lock();      // sp3指向与sp1相同的对象
sp2.reset();
sp3.reset();                  // 所指对象被释放
if (wp.lock() == nullptr) cout <<  "invalid" << endl;     // invalid
```

Plain Old data (POD)

Plain Old Data

Introduction

Plain意为POD是普通的类型，而不像一些存在着虚函数虚继承的类型那么特别。Old体现了其与C的兼容性，比如可以使用memcpy()进行复制，使用memset()进行初始化。

想知道某个类型是否是POD时，可以使用标准库中的
`template <typename T> struct std::is_pod;`

来判断，如：

```
cout << is_pod<int>::value << endl;   // 1
```

Plain Old Data

Introduction

这种从名字上看，普通、老式的数据由什么好处呢？

1. **字节赋值**。我们可以安全地使用memset和memcpy对POD进行初始化或拷贝等操作。

2. **提供对C内存布局兼容**。POD类型的数据在C和C++间的操作总是安全的。

3. **保证静态初始化的安全有效**。静态初始化能在很多时候提高程序性能，而POD类型对象初始化往往更加简单。

Plain Old Data

Introduction

具体地，C++11将POD划分为两个基本概念的合集，即：平凡的(trivial)和标准布局的(standard layout)。

下面详细介绍以下这两种概念的规则。

Plain Old Data

Trivial(平凡的)

一个平凡的类或结构体要符合以下定义：

1. 拥有平凡的默认构造函数(trivial constructor)和析构函数(trivial destructor)。

平凡的构造函数就是说构造函数“什么都不干”。通常，不定义类的构造函数，编译器会为我们生成一个平凡的默认构造函数。而一旦定义了构造函数，即使是如下的

```
struct NoTrivial { NoTrivial(); };
```

那么该构造函数也不再是平凡的。这个对析构函数也类似。

可以使用=default关键字显式地声明平凡构造函数。

Plain Old Data

Trivial(平凡的)

2. 拥有平凡的拷贝构造函数(trivial copy constructor)和移动构造函数(trivial move constructor)。

平凡的拷贝构造函数基本上等同于memcpy。同样的，不声明拷贝构造函数的话，编译器会自动地生成平凡的拷贝构造函数，或者程序员显式地使用=default声明。

移动构造函数类似，只是用于移动语义。

Plain Old Data

Trivial(平凡的)

3. 拥有平凡的拷贝赋值运算符(trivial assignment operator)和移动赋值运算符(trivial move operator)。

　　类似于上页的平凡拷贝构造和平凡的移动构造。

4. 不能包含虚函数以及虚基类。

Plain Old Data

Trivial(平凡的)

以上四点虽然不算太复杂，不过在需要的时候，也可以使用C++11中辅助的类模板来帮我们判断。

```
template <typename T> struct std::is_trivial;
```

Plain Old Data
Standard Layout(标准布局)

1. 所有非静态成员有相同的访问权限。
2. 在类或者结构体继承时，满足以下情况之一

    1) 派生类中有非静态成员，且只有一个仅包含静态成员的基类。

    2) 基类有非静态成员，而派生类没有非静态成员
3. 类中第一个非静态成员的类型与其基类不同。
4. 没有虚函数和虚基类
5. 所有非静态数据成员均符合标准布局，其基类也符合标准布局

Plain Old Data
Standard Layout(标准布局)

```
struct NoStandard {
        public:
                int a;
        private:
                int b;
};
struct Standard {    // 所有非静态成员有相同的访问权限。
        public:
                int a;
                int b;
}
```

Plain Old Data
Standard Layout(标准布局)

```
/*
    2.  在类或者结构体继承时，满足以下情况之一
    1)  派生类中有非静态成员，且只有一个仅包含静态成员的基类。
    2)  基类有非静态成员，而派生类没有非静态成员
*/
struct B1 { static int a; };
struct D1 : B1 { int d; };

struct B2 { int a; };
struct D2 : B2 { static int d; };

struct D3 : B2, B1 { static int d; };
struct D4 : B2 { int d; };
struct D5 : B2, D1 { };
```

Plain Old Data
Standard Layout(标准布局)

```
// 3. 类中第一个非静态成员的类型与其基类不同。
struct B { };
struct D1 : B {
        B b;                // 第一个非静态员量与基类相同，不是标准布局
        int i;
};

struct D2 : B {  // 第一个非静态变量与基类不同，是标准布局
        int i;
        B b;
};
```

Plain Old Data
Standard Layout(标准布局)

```
/*
4.    没有虚函数和虚基类
5.    所有非静态数据成员均符合标准布局，其基类也符合标准布局
*/

class B {                    // 虚基类
      virtual int foo() = 0;
};

class D : B { };          // 不符合标准布局
```

Object construct enhancement

Delegate Contructor
委托构造函数

在C++98中，如果你想让两个构造函数完成相似的事情，可以写两个大段代码相同的构造函数，或者是另外定义一个init()函数，让两个构造函数都调用这个init()函数。

这样的实现方式重复罗嗦，并且容易出错。并且，这两种方式的可维护性都很差。所以，在C++0x中，我们可以在定义一个构造函数时调用另外一个构造函数，称为委托构造函数。

Delegate Contructor (without)

```
class class_c {
public:
    int max; int min; int middle;
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
```

Delegate Contructor (with)

```
class class_c {
public:
    int max; int min;int middle;
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) : class_c(my_max) {
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle)
        : class_c (my_max, my_min){
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
```

Delegate Constructor (without)

```
class X {
    int a;
    // 实现一个初始化函数
    validate(int x) {
        if (0<x && x<=max) a=x; else throw bad_X(x);
    }
public:
    // 三个构造函数都调用validate(), 完成初始化工作
    X(int x) { validate(x); }
    X() { validate(42); }
    X(string s) {
        int x = lexical_cast<int>(s); validate(x);
    }
    // …
};
```

Delegate Contructor (with)

```
class X {
    int a;
public:
    X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
    // 构造函数X()调用构造函数X(int x)
    X() :X{42} { }
    // 构造函数X(string s)调用构造函数X(int x)
    X(string s) :X{lexical_cast<int>(s)} { }
    // …
};
```

Inherit Constructor

We will meet this question in C++98….

```
struct B {
    void f(double);
};
struct D : B {
    void f(int);
};
B b;    b.f(4.5);     // OK
// 调用的到底是B::f(doube)还是D::f(int)呢？
// 实际情况往往会让人感到意外：调用的f(int)函数实参为4
D d;    d.f(4.5);
```

Solution in C++98

```
struct B {
    void f(double);
};
struct D : B {
    using B::f;       // 将类B中的f()函数引入到类D的作用域内
    void f(int);      // 增加一个新的f()函数
};
B b;   b.f(4.5);      // OK
// 可行：调用类D中的f(double)函数
// 也即类B中的f(double)函数
D d;   d.f(4.5);
```

But how can you solve constructors ??

Inherit Constructor

```
class Derived : public Base {
public:
    // 提升Base类的f函数到Derived类的作用范围内
    // 这一特性已存在于C++98标准内
    using Base::f;
    void f(char);      // 提供一个新的f函数
    void f(int);  // 与Base类的f(int)函数相比更常用到这个f函数
    // 提升Base类的构造函数到Derived的作用范围内
    // 这一特性只存在于C++11标准内
    using Base::Base;
    Derived(char);     // 提供一个新的构造函数
    // 与Base类的构造函数Base(int)相比
    // 更常用到这个构造函数
    Derived(int);
    // …
};
```

Inherit Constructor(Confuse...)

```
struct B1 {
    B1(int) { }
};
struct D1 : B1 {
    using B1::B1; // 隐式声明构造函数D1(int)
    int x;
};
void test()
{
    D1 d(6);   // 糟糕：调用的是基类的构造函数，d.x没有初始化
    D1 e;      // 错误：类D1没有默认的构造函数
}
```

Remember Imediate Initialize?

```
struct D1 : B1 {
    using B1::B1;     // 隐式声明构造函数D1(int)
    // x变量初始化
    int x{0};
};
void test()
{
    D1 d(6);     // d.x的值是0
}
```

Const Expression
常量表达式

Const Expression

运行时常量性与编译时常量性

以前的学习中我们遇到过const关键字，它可以修饰变量、函数等等，表示某些值不可修改，是“常量”。而这些“常量”都是“运行时常量”，即是在运行时保证其不被修改。

但是有时候需要用到“编译时常量”时，const的约束是不够的。

Const Expression
运行时常量性与编译时常量性

```
const int GetConst() { return 1; }   // 仅保证运行时常量性

int main() {
    int arr[GetConst()] = { 0 };     // 无法通过编译
                                     // 数组的长度是编译时需要确定的
    int a = 0;
    cin >> a;

    switch(a) {
        case GetConst():             // 无法通过编译
            break;                   // 同上理
        default:
            break;
    }
}
```

Const Expression

constexpr

实际上，使用宏定义#define也能解决一些问题，但是由于它只是字符串上的替换，所以不便控制。所以C++11提供了新的关键字constexpr，将其替换const即可达到想要的效果。如上页代码中，改为

```
constexpr int GetConst() { return 1; }
```

即可通过编译。

Const Expression

constexpr

虽然我们可以使用constexpr来使一个函数成为常量表达式函数，但其实常量表达式函数要求非常严格。要求有以下几点：

1. 函数体只有单一的return返回语句(C++14放松了要求)
2. 函数必须返回值(不能是void)
3. 在使用前必须定义
4. return语句中不能使用非常量表达式的函数、全局数据，且必须是一个常量表达式。

Const Expression

constexpr

```
// 1. 只有单一的return语句
constexpr int f1() { const int a = 0; return a; }      // 编译错误
//    然而，不产生实际代码的语句是可以使用的
constexpr int f2() {
    typedef int INT;     // 不产生实际代码，类似的还有using
    return 1;
}

// 2. 必须返回值
constexpr void f3() { }    // 这样的函数并没有什么意义
```

Const Expression

constexpr

```
// 3. 常量表达式函数在使用前必须被定义
constexpr int f();                    // 仅声明，未定义
int a = f();                          // 可通过
const int b = f();                    // 可通过

constexpr int c = f();                // 编译不通过，常量表达式未被定义
constexpr int f() { return 1; }  // 真正的定义
constexpr int d = f();                // 编译可通过

/*
或许你疑惑，为什么a和b的编译是可以通过的。其实常量表达式如果被这样调用时，会自动生成一个没有constexpr的版本，这样其使用就可以和普通函数一样(即先声明后定义)。并且显式地声明一种无constexpr的函数反而会报编译错误。
*/
```

Const Expression

constexpr

```
// 4. return语句中不能使用非常量表达式的函数、全局数据，且必须是一个
//    常量表达式。

const int f() { return 1; }                    // 并不是常量表达式
constexpr GetConst1() { return f(); }          // 编译不通过

const int x = 1;
constexpr GetConst2() { return x; }            // 不通过

constexpr GetConst3(int x) { return x = 1; }   // 同时一些危险的操作如赋值
                                               // 在常量表达式中也是不允许
                                               // 的
```

Const Expression
constant-expression value

常量表达式值(constant-expression value)其实就是使用constexpr修饰的变量(其实这个在前面的代码中已经有出现过)。而我们需要区别的是

```
const int x = 1;
constexpr int y = 1;
```

这两者。

Const Expression
constant-expression value

```
const int x = 1;
constexpr int y = 1;
```

对于x，编译器是一定会为其生成数据的；而对于y，除非有代码显式使用了y，否则是不会生成数据的。前文说过，constexpr是编译时生效的，因此y就如有名字而不产生数据的枚举值，还有右值字面常量(1, 2.0f等等)一样，是编译时的常量。

Const Expression
constant-expression value

如果想要用constexpr修饰自定义类型的变量，那就没有那么简单了。如

```
constexpr struct MyType { int i; };

constexpr MyType mt = { 0 };
```

在C++11是无法通过编译的。

想要这样使用，就需要定义常量构造函数。

Const Expression
constant-expression constructor

```
/*
常量表达式构造函数也有约束
1.    函数体必须为空
2.    初始化列表只能由常量表达式来赋值
*/

// 正确示范
struct MyType {
        constexpr MyType(int x) : i(x) { }
        int i;
};
constexpr MyType mt = { 0 };
```

Const Expression

Extra

C++14放松了constexpr函数的要求(https://en.wikipedia.org/wiki/C%2B%2B14#Relaxed_constexpr_restrictions)

String Literals
字符串字面量

String Literals

Introduction

字符串字面量，就是指这个量就如“字面上”一样。
例如我们想让程序输出

Hello

World

我们应该会写

```
std::cout << "Hello\nWorld" << std::endl;
```

String Literals

Introduction

```
std::cout << "Hello\nWorld" << std::endl;
```

这样的写法就不是“所见即所得”，或许我们更希望输入

"Hello

World"

就能得到我们想要的

而字符串字面量就是为此而生的

String Literals

R

```
int main() {
        char s1[] = "Hello\nWorld";
        char s2[] = R"Hello
                                World";                 // 输出时前面的缩进也会被保留

        std::cout << s1 << std::endl;
        std::cout << s2 << std::endl;
        /*      输出
Hello
World
Hello
                                World
        */

        std::cout << R"Hello\nWorld";      // 如“字面上”一样输出，不会对\n转义
        /*      输出
Hello\nWorld
        */
        return 0
}
```

String Literals
User-defined literals

在以前的编程中，我们遇到过用后缀(suffix)来表示字面量的类型的写法，如

```
11L       // long
1.2f      // float
```

而C++11还支持用户自定义后缀。

String Literals
User-defined literals

基本语法是

```
return_type operator "" _suffix(paraments_list);
```

如

```
long long int operator "" _x(long double);
```

这样，就可以用1.2_x, 123.5_x来调用上面重载的函数了

下面来个例子吧

String Literals
User-defined literals

```
struct Test {
  unsigned long long int x;
};

Test operator "" _X (unsigned long long int t_x) {        // operator "" _suffix
  Test a;
  a.x = t_x;
  return a;
}

int main() {
  std::cout << (123_X).x << std::endl;
  return 0;
}
```

String Literals
User-defined literals

需要注意的是，这种重载运算符，只支持一些固定的参数类型

Only the following parameter lists are allowed on literal operators :

(`const char *`)	(1)
(`unsigned long long int`)	(2)
(`long double`)	(3)
(`char`)	(4)
(`wchar_t`)	(5)
(`char16_t`)	(6)
(`char32_t`)	(7)
(`const char *` , `std::size_t`)	(8)
(`const wchar_t *` , `std::size_t`)	(9)
(`const char16_t *` , `std::size_t`)	(10)
(`const char32_t *` , `std::size_t`)	(11)

参考自cppreference.com

Standard library Updates

Upgrades to standard library components

- Rvalue references and the associated move support
- Support for the UTF-16 encoding unit, and UTF-32 encoding unit Unicode character types
- Variadic templates (coupled with Rvalue references to allow for perfect forwarding)
- Compile-time constant expressions
- decltype
- explicit conversion operators
- default/deleted functions

Tuple types

```
tuple<string,int> t2(“Kylling”,123);


// t的类型被推断为tuple
auto t = make_tuple(string(“Herring”),10, 1.23);
// 获取元组中的分量
string s = get<0>(t);
int x = get<1>(t);
double d = get<2>(t);
```

Hash tables

Type of hash table	Associated values	Equivalent keys
std::unordered_set	No	No
std::unordered_multiset	No	Yes
std::unordered_map	Yes	No
std::unordered_multimap	Yes	Yes

Regular Expression

```
// Simple regular expression matching
std::string fnames[] = {"foo.txt", "bar.txt", "baz.dat",
"zoidberg"};
std::regex txt_regex("[a-z]+\\.txt");
for (const auto &fname : fnames) {
    std::cout << fname << ": ";
    std::cout << std::regex_match(fname, txt_regex);
    std::cout << '\n';
}
```

foo.txt: 1\nbar.txt: 1\nbaz.dat: 0\nzoidberg: 0

General-purpose smart pointers

```
template <class T, class D = default_delete<T>> class
unique_ptr;

template <class T> class shared_ptr;

template <class T> class weak_ptr;
```

Explore other library updates by yourself….

Other C++11 Features

Range Base Loop

```
#include <iostream>
#include <vector>


int main() {
  vector<int> a {1,2,3,4,5,6,7,8};
  for(auto i : a) {
    std::cout << i << " ";
  }
  std::cout << endl;
  return 0;
}
```

Variadic templates

```
template<typename T, typename... Args>     // 注意这里的"..."
void printf(const char* s, T value, Args... args)    // 注意"..."
{
    while (s && *s) {
        //一个格式标记（避免格式控制符）
                    if (*s=='%' && *++s!='%') {
            std::cout << value;
            return printf(++s, args...);// 使用第一个非格式参数
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra args provided to printf");
}
```

Static (Compile Time) Assertion

静态(编译期)断言由一个常量表达式及一个字符串文本构成:

static_assert(expression, string);

Static (Compile Time) Assertion

```
static_assert(sizeof(long) >= 8,
   "64-bit code generation required for
this library.");
struct S { X m1; Y m2; };
static_assert(sizeof(S)==sizeof(X)+sizeof
(Y),
    "unexpected padding in S");
```

Static (Compile Time) Assertion

```
int f(int* p, int n) {
  //错误：表达式“p == 0”,不是一个常量表达式
  // 无法在编译阶段确定p
  static_assert(p == 0,
     “p is not null”);
}
```

Sizeof operator without an explicit object

```
struct SomeType { OtherType member; };


sizeof(SomeType::member); // Does not work
with C++03. Okay with C++11
```

Explicit override

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

struct Derived : Base {
    virtual void some_func(int) override;
// ill-formed - doesn't override a base
class method
};
```

Explicit final

```
struct Base1 final { };
struct Derived1 : Base1 { };
// ill-formed because the class Base1 has been marked
final
```

```
struct Base2 {
    virtual void f() final;
};
struct Derived2 : Base2 {
    void f(); // ill-formed because the virtual
function Base2::f has been marked final
};
```

Explicit default and deleted

```
class Producer {
private:
 /**
  * Disallow copy constructor and assign operator
  */
 Producer(const Producer & t_another) = delete;
 Producer &operator=(const Producer & t_another)=delete;
 Producer() = default;
public:
....
};
```

Explicit conversion operators

In C++ 98:

```
struct S { S(int); };     // "普通构造函数"默认是隐式转换
S s1(1);          // ok, 直接构造
S s2 = 1;     // ok, 隐式拷贝构造
void f(S);
// 能通过编译（但是经常会产生意外结果—如果S是vector类型会怎么样呢？）
// 译注：详见下一用例的解释
f(1);

struct E { explicit E(int); };     // 显式构造函数
E e1(1);          // ok
E e2 = 1;     // 错误（但是常常会让人感到意外—这怎么会错呢？）
void f(E);
// 该处会产生编译错误（而非编译通过），以避免因隐式类型转换而得到莫名其妙的结果。
f(1);
```

Explicit conversion operators

```
struct S { S(int) { } /* … */ };
struct SS {>
    int m;
    SS(int x) :m(x) { }
    // 在struct S无须定义S(SS)—所谓的"非侵入"式做法
    operator S() { return S(m); }
};
SS ss(1);     // ok; 默认构造函数
S s1 = ss;    // ok; 隐式转换为S后调用拷贝构造函数
S s2(ss);     // ok; 隐式转换为S后调用直接构造函数
void f(S);
f(ss);         // ok; 隐式转换为S后传参
```

Explicit conversion operators

C++11:

```
struct S { S(int) { } };
struct SS {
    int m;
    SS(int x) :m(x) { }
    // 因为结构体S中没有定义构造函数S(SS)
    // 无法将SS转换为S，所以只好在SS中定义一个返回S的转换操作符，
    // 将自己转换为S。
    // 转换动作，可以由目标类型S提供，也可以由源类型SS提供。）
     explicit operator S() { return S(m); }
};
SS ss(1);     // ok；默认构造函数
S s1 = ss;     // 错误；拷贝构造函数不能使用显式转换
S s2(ss);     // ok；直接构造函数可以使用显式转换
void f(S);
f(ss);         // 错误；从SS向S的转换必须是显式的.
```

Null pointer constant

```
char *pc = nullptr;      // OK
int  *pi = nullptr;      // OK
bool   b = nullptr;      // OK. b is false.
int    i = nullptr;      // error
foo(nullptr);            // calls foo
(nullptr_t), not foo(int);
```

Why “NULL” is not good?

Right Angle Bracket

```
template<bool Test> class SomeType;
std::vector<SomeType<1>2>> x1;
// Interpreted as a std::vector of SomeType<true>,
// followed by "2 >> x1", which is not legal syntax for
a declarator. 1 is true.
std::vector<SomeType<(1>2)>> x1;
// Interpreted as std::vector of SomeType<false>,
// followed by the declarator "x1", which is legal C++11
syntax. (1>2) is false.
```

Unmentioned Features

1. External templates
2. Variadic templates
3. Control and query object alignment
4. Attributes
5. Unrestricted Union
6. Template aliases
7. Type long long int
8. Strongly type enumerations
9. **Multithreading memory model (very important, but Beyond your scope of knowledge)**

Explore it by yourself....

C++14 Feature Updates

Function return type deduction

```
auto DeduceReturnType();

// Return type to be determined.
```

Alternate type deduction on declaration

C++14 adds the decltype(auto) syntax.

```
int i;
int&& f();
auto x3a = i;                  // x3a的类型是int
decltype(i) x3d = i;           // x3d的类型是int
auto x4a = (i);                // x4a的类型是int
decltype((i)) x4d = (i);       // x4d的类型是int&
auto x5a = f();                // x5a的类型是int
decltype(f()) x5d = f();       // x5d的类型是int&&
```

Digit separators

```
auto integer_literal = 1'000'000;

auto floating_point_literal = 0.000'015'3;

auto binary_literal = 0b0100'1100'0110;

auto silly_example = 1'0'0'000'00;
```

Standard user-defined literals

```
auto str = "hello world"s;

// auto deduces string

auto dur = +1s;

// auto deduces chrono::seconds

auto z   = 1i;

// auto deduces complex<double>
```

C++14 Other Features

Explore it by yourself....

C++17 Future Features

Just Explore!

Expected features [edit]

- Addition of a default text message for `static_assert`[2]
- Removal of trigraphs[3][4]
- Allow `typename` in a template template parameter[5]
- New rules for `auto` deduction from braced-init-list[6][7]
- `std::uncaught_exceptions`, as a replacement of `std::uncaught_exception`[8][9]
- Nested namespace definition[7][10]
- Attributes for namespaces and enumerators[9][11]
- UTF-8 character literals[9][12]
- Constant evaluation for all non-type template arguments[9][13]
- Folding expressions[9][14]
- New insertion functions for `std::map` and `std::unordered_map`[15][16]
- Uniform container access[16][17]
- Definition of "Contiguous Iterators"[16][18]
- Removal of some deprecated types and functions like `std::auto_ptr`, `std::random_shuffle` or old function adaptors[19][7]
- A file system library based on `boost::filesystem`[20]
- Parallel versions of STL algorithms[21]
- Additional mathematical special functions[22]
- Most of Library Fundamentals TS I[23]

GCC已经支持部分特性

g++ SOURCE_CODE -std=c++1z

Thank You!

(If you do not know C++1x, you are out)