略说C++11

本章导读

学习目标：

1. 初步了解C++11；
2. 初步了解unique_ptr、shared_ptr和weak_ptr；
3. 了解Lambda函数；
4. 了解新增类库；

目录

C++11标准，原名C++0x，是C++的最新正式标准(ISO/IEC 14882:2011)，它将取代第二版标准C++98以及C++03(两者差别极小)。C++11包含了核心机能的升级，STL的拓展，并且加入了大部分C++ TR1程序库(除数学上的特殊函数)。C++11是自1998年C++98以来第一次重大升级。根据计划，ISO将在2014、2017年发布C++的后续版本。目前，各家编译器对标准的成都各有不同。

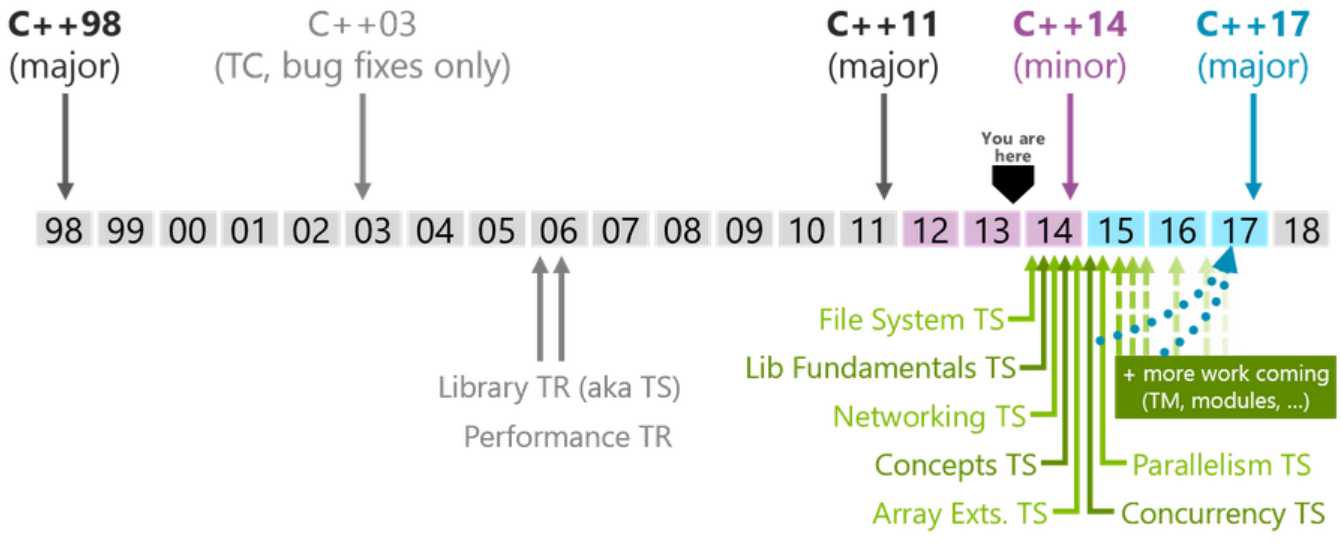

图1　C++标准相关演进

在C++标准升级过程中，标准委员会尽力遵循以下原则：

- 维持与C++98，可能的话还有与C之间的稳定性与兼容性；
- 尽量不通过核心语言的扩展，而是通过标准程序库来引进新的特色；
- 以能够演进编程技术的变更优先；
- 改进 C++以帮助系统以及库设计，而不是引进只针对特别应用的新特色；
- 增进类型安全，提供对现行不安全技术更安全的替代方案；
- 增加直接对硬件的能力与表现；
- 提供现实世界中问题的适当解决方案；
- 尽力实现“zero-overhead(零开销)”原则;
- 使C++更易于教授与学习。

C++11的主要升级可以分为两个方面，即核心语言升级以及STL升级，核心语言升级是STL升级的基础。

第一节　主要升级

C++标准委员会的主要焦点是在语言核心。C++11语言核心升级领域包括：多线程支持、泛型编程强化、统一的初始化、以及性能提升。这些升级大致可以分为： 运行期强化、建构其强化、可用性强化，以及其他一些强化。

1、新数据类型

C++新增类型long long和unsigned long long，以支持64位整型；新增char16_t和char32_t，以支持16位和32位的字符表示。另外，还新增了“原始(Raw)”字符串常量。

例程1

```
第1行  #include <iostream>
第2行  #include <string>
第3行
第4行  using namespace std;
第5行
第6行  int main()
第7行  {
第8行      string normal_str="First line.\nSecond line.\nEnd of message.\n";
第9行      //原样显示
第10行     string raw_str=R"(First line.\nSecond line.\nEnd of message.\n)";
第11行     cout<<normal_str<<endl;
第12行     cout<<raw_str<<endl;
第13行     return(0);
第14行 }
```

2、统一初始化

C++11扩大了用花括号({})括起初始化列表的适用范围，不仅可以适用于数组，还可以适用于基本数据类型以及自定义的结构和类。当使用初始化列表时，可以有等号(=)，也可以省略。如例程2所示。

例程2

```
第1行  int x{5};
第2行  double y{2.75};
第3行  int dataA[5]={1,3,5,1,2};
第4行  int dataB[5]{1,3,5,1,2};
第5行
第6行  //适用于new运算符
第7行  int *p=new int[5]{2,4,6,8,10};
第8行
```

```
第9行  class udC{
第10行 public:
第11行     udC(int a,double b):numA(a),numB(b){}
第12行 private:
第13行     int numA;
第14行     double numB;
第15行 }
第16行
第17行 udC myUDcA(10,99.99);//传统方式
第18行 udC myUDcB{11,9.9};
第19行 udC myUDcC={12,12.34};
第20行
第21行 struct udS{
第22行     int numA;
第23行     double numB;
第24行 }
第25行 udS myUDs{5, 3.2f};//传统C模式
```

3、申明变化

⑴ auto与decltype

在以往的版本中，auto是一种存储类型说明符，C++11将其调整为自动类型推断，此时要求进行显式初始化，以让编译器能够根据初始化数值推断变量的类型。decltype关键字则是让变量申明为表达式指定的类型，在定义模板时特别有用。如例程3所示。

例程3

```
第1行  auto data=112;//data的类型推断为int
第2行  auto pData=&data;//pData的类型推断为int *
第3行  int plus(int a,int b);//函数原型
第4行  auto pFun=plus;//pFun的类型被推断为int (*)(int,int)
第5行
第6行  //iter无需申明为vector<int>::iterator iter，用auto更加简洁
第7行  for(auto iter=myVec.begin();iter!=myVec.end();++iter);//myVec为vector
第8行
第9行  double a;
第10行 int b;
第11行
第12行 decltype (b*2) c;//c的类型为B*2，为int型
第13行 decltype (a*b) d;//d的类型为a*b的类型，为double型
第14行
第15行 //有时，在定义模板时特别有用
第16行
第17行 template<typename Ta,typename Tb>
第18行 void someFun(Ta a,Tb b){
第19行     decltype(a*b) tmp;//tmp的类型有a*b确定
第20行 }
```

⑵ 返回类型后置

C++11新增一种函数申明语法：在函数名和参数表后面指定返回类型，而不是传统模式在前面指定类型，如例程4所示。

例程4

```
第1行  double plus(double a,double b);//传统模式，返回double类型
第2行  template<typename Ta,typename Tb>
第3行  auto someFun(Ta a,Tb b)->decltype(T*U);//C++11
```

返回类型后置，有时在迭代器中显得非常有用。

例程5

```
第1行  #include<iostream>
第2行  #include<vector>
第3行
第4行  using namespace std;
第5行
第6行  //注意：注意观察函数申明和实现
第7行  template<typename T>
第8行  auto iterMulti(T a, T b)->decltype(*a * *b){
第9行      decltype(*a * *b) tmp;
第10行     tmp = *a + *b;
第11行     return tmp;
第12行 }
第13行 int main(){

第14行     vector<int> myVec;
第15行     myVec.push_back(1);

第16行     myVec.push_back(2);
第17行     myVec.push_back(1);
第18行     myVec.push_back(2);
第19行
第20行     cout << iterMulti(myVec.begin(), (myVec.begin() + 1)) << endl;
第21行
第22行     system("pause");
第23行     return 0;
第24行 }
```

一般情况下，不建议采用这种方式。但在定义模板函数时，可以用decltype指定函数的返回类型。

⑶ 模板别名

对于冗长或复杂的类型，使用别名将很方便，在C++11前的版本，可以使用typedef，如：

```
typedef std::vector<std::string>::iterator iterType;
```

C++11还提供了另外一种创建的别名的语法，如：

```
using iterType=std::vector<std::string>::iterator;
```

差别在于，新语法也可以用于模板部分具体化，但typedef则不可以。如：

```
template<typename T>
  using arr12=std::array<T,12>
```

上述语句部分具体化模板array<T,int>，例程：

```
std::array<double,12>dblA;
```

std::array<std::string,12>strB;

上述申明可以替换如下：

arr12<double>dblA;

arr12<std::string>strB;

4、序列for循环

在C++中for循环可以使用类似java、C#的简化for循环，可以用于遍历数组，容器，string以及由begin和end函数定义的序列(即有Iterator)，示例代码如下：

例程6

```
#include<iostream>
#include<map>
#include<string>

using namespace std;

int main(){
    map<string, int> m{ { "a", 1 }, { "b", 2 }, { "c", 3 } };
    for (auto p : m){
        cout<< p.first << " : " << p.second << endl;
    }
}
```

5、deleted和defaulted函数

被称为一个defaulted函数，“=default;”告诉编译器为函数生成默认的实现。Defaulted函数有两个好处：比手工实现更高效，让程序员摆脱了手工定义这些函数的苦差事。

例程7

```
struct A{
    A() = default; //C++11
    virtual ~A() = default; //C++11
};
```

与defaulted函数相反的是deleted函数。Deleted函数对防止对象复制很有用，回想一下C++自动为类声明一个副本构造函数和一个赋值操作符，要禁用复制，声明这两个特殊的成员函数=delete即可。

例程8

```
#include<iostream>
using namespace std;
struct NoCopy{
    NoCopy & operator =(const NoCopy &) = delete;
    NoCopy(const NoCopy &) = delete;
};

int main(){
    NoCopy a;//Error!不存在默认的构造函数
    NoCopy b(a); //Error!
}
```

6、委托构造函数

在C++11中，构造函数可以调用相同类中的其它构造函数。

例程9

```
class M{ //C++11 delegating constructors
    int x, y;
    char *p;
public:
    M(int v) : x(v), y(0), p(new char[1000]) {} //#1 target
    M() : M(0) {
        cout << "delegating ctor" << endl;
    }
};
```

7、空指针

空指针是指不会指向有效数据的指针。在以前的C++版本中，使用0表示空指针，但0既可以表示常量指针，也可以表示整型常量，这带来了潜在的安全问题。在C++11中增加了关键字nullptr，用于表示空指针，它纯是指针类型。为了向后兼容，C++仍允许使用0表示空指针，因此nullptr==0的值为true，但使用nullptr提高了类型安全。

8、智能指针

如果在程序中使用new动态分配内存，应在不需要时使用delete将内存空间释放。C++11以前，可以用auto_ptr以帮助自动完成这个过程。在C++11版本中，使用了更加精致的机制，新增三种智能指针unique_ptr、shared_ptr和weak_ptr，并放弃auto_ptr。新增智能指针与STL容器和move语义协同工作。在C+11中，建议不再使用auto_ptr，而是改用unique_ptr等代替。

(1) unique_ptr

unique_ptr是一种定义在<memory>中的智能指针(smart pointer)。它持有对对象的独有权即两个unique_ptr不能指向一个对象，不能进行复制操作只能进行移动操作。unique_ptr在超出作用域，即以下情况时它指向的对象会被摧毁：unique_ptr指向的对象被破坏；对象通过operator=()或reset()被指定到另一个指针。unique_ptr还可能没有对象，这种情况被称为empty。

例程10

```
#include<iostream>
#include<memory>
using namespace std;
void Test(){
    unique_ptr<int> pA(new int(5));
    cout << *pA << endl;

    //如下使用方法错误，不能让pB指向pA，想想explicit
    //unique_ptr<int>pB = pA;

    unique_ptr<int>pC = move(pA);
    // 转移所有权，现在那块内存归pC所有，pA成为无效的针.

    //注意：*pA错误，pA已经成为无效指针
    //cout << *pA << endl;
    cout << *pC << endl;//输出5

    pA.reset();//,实际上什么都没做，已经是无效指针
    pC.reset();//释放内存

```

```
第21行  }//注意：没有使用delete，Why?
第22行  int main(){
第23行      Test();
第24行      system("pause");
第25行      return 0;
第26行  }
```

unique_ptr的功能：

1. 不管是正常退出还是异常退出，均可通过保证删除为处理拥有动态寿命的类和函数提供额外的保护；
2. 将独有的持有动态寿命对象传递给函数；
3. 从函数获取持有动态寿命对象的所有权
4. 所有auto_ptr应该已经具有的功能

(2) shared_ptr

shared_ptr是一个引用计数智能指针，用于共享对象的所有权。它可以从一个裸指针、另一个shared_ptr、一个auto_ptr、或者一个weak_ptr构造。还可以传递第二个参数给shared_ptr的构造函数，它被称为删除器（deleter）。删除器用于处理共享资源的释放，这对于管理那些不是用new分配也不是用delete释放的资源时非常有用。shared_ptr被创建后，就可以像普通指针一样使用了，除了一点，它不能被显式地删除。

例程11

```
第1行   #include<iostream>
第2行   #include<memory>
第3行   using namespace std;
第4行   shared_ptr<int> pOuter;
第5行   void Test(){
第6行       shared_ptr<int> pA;
第7行       cout << pA.use_count() << endl;
第8行       shared_ptr<int> pB(new int(100));
第9行       cout << pB.use_count() << endl;
第10行      pOuter = pB;
第11行      cout << pB.use_count() << endl;
第12行  }
第13行  int main(){
第14行      Test();
第15行      cout << pOuter.use_count() << endl;
第16行      cout << *pOuter << endl;//输出100
第17行      system("pause");
第18行      return 0;
第19行  }
```

(3) weak_ptr

有时对象必须存储一种方法，用来在不引起引用计数增加的情况下访问 shared_ptr 的基础对象。 通常，当您在 shared_ptr 实例之间循环引用时，就会出现此情况。

最佳的设计能够尽可能地避免指针具有共享所有权。 但是，如果您必须具有共享的 shared_ptr 实例所有权，请避免在实例之间进行循环引用。 如果循环引用不可避免，甚至由于某种原因而更为可取，请使用 weak_ptr 为一个或多个所有者提供对其他 shared_ptr 的弱引用。 使用 weak_ptr，您可以创建连接到现有相关实例组的 shared_ptr，但仅当基础内存资源有效时才行。 weak_ptr 本身并不参与引用计数，因此，它无法阻止引用计数转到为零。 但是，您可以使用 weak_ptr 来尝试获取 shared_ptr 的新副本，通过使用该副本进行初始化。 如果内存已被删除，则会引发 bad_weak_ptr 异常。 如果内存仍有效，则新的共享指针会递增引用计数，并确保只要 shared_ptr 变量保持在范围内，内存就有效。

```
第1行  #include <iostream>
第2行  #include <memory>
第3行  #include <string>
第4行  #include <vector>
第5行  #include <algorithm>
第6行
第7行  using namespace std;
第8行
第9行  class Controller
第10行 {
第11行 public:
第12行     int Num;
第13行     wstring Status;
第14行     vector<weak_ptr<Controller>> others;
第15行     explicit Controller(int i) : Num(i), Status(L"On")
第16行     {
第17行         wcout << L"Creating Controller" << Num << endl;
第18行     }
第19行
第20行     ~Controller()
第21行     {
第22行         wcout << L"Destroying Controller" << Num << endl;
第23行     }
第24行
第25行     // Demonstrates how to test whether the
第26行     // pointed-to memory still exists or not.
第27行     void CheckStatuses() const
第28行     {
第29行         for_each(others.begin(), others.end(), [](weak_ptr<Controller> wp)
第30行         {
第31行             try
第32行             {
第33行                 auto p = wp.lock();
第34行                 wcout << L"Status of " << p->Num << " = " << p->Status << endl;
第35行             }
第36行
第37行             catch (bad_weak_ptr b)
第38行             {
第39行                 wcout << L"Null object" << endl;
第40行             }
第41行         });
第42行     }
```

```
第43行  };
第44行
第45行  void RunTest()
第46行  {
第47行      vector<shared_ptr<Controller>> v;
第48行
第49行      v.push_back(shared_ptr<Controller>(new Controller(0)));
第50行      v.push_back(shared_ptr<Controller>(new Controller(1)));
第51行      v.push_back(shared_ptr<Controller>(new Controller(2)));
第52行      v.push_back(shared_ptr<Controller>(new Controller(3)));
第53行      v.push_back(shared_ptr<Controller>(new Controller(4)));
第54行
第55行      // Each controller depends on all others not being deleted.
第56行      // Give each controller a pointer to all the others.
第57行      for (unsigned int i = 0; i < v.size(); ++i)
第58行      {
第59行          for_each(v.begin(), v.end(), [v, i](shared_ptr<Controller> p)
第60行          {
第61行              if (p->Num != i)
第62行              {
第63行                  v[i]->others.push_back(weak_ptr<Controller>(p));
第64行                  wcout << L"push_back to v[" << i << "]: " << p->Num << endl;
第65行              }

第66行          });
第67行      }
第68行
第69行      for_each(v.begin(), v.end(), [](shared_ptr<Controller>& p)
第70行      {
第71行          wcout << L"use_count = " << p.use_count() << endl;
第72行          p->CheckStatuses();
第73行      });
第74行  }
第75行
第76行  int main()
第77行  {
第78行      RunTest();
第79行
第80行      system("pause");

第81行      return 0;
第82行  }
```

9、异常规约升级

在C++11的异常规约中，增加了noexcept关键字，如：

void print()noexcept;//不会触发任何异常

10、右值引用与move语义

如例程例程13第14行所示，move(str)的含义是将str变量的内容移动，注意：不是拷贝，当移动后，str内容为空。例程13能更好地看出move语义的使用。

例程13

```
第1行  #include <iostream>
第2行  #include <vector>
第3行  #include <string>
第4行  using namespace std;
第5行
第6行  int main()
第7行  {
第8行      string str = "Hello";
第9行      vector<string> v;
第10行
第11行     v.push_back(str);
第12行     cout << "After copy, str is \"" << str << "\"\n";
第13行
第14行     v.push_back(move(str));//!!!!!!
第15行     cout << "After move, str is \"" << str << "\"\n";
第16行
第17行     cout << "The contents of the vector are " << v[0]<< " "<< v[1] << endl;
第18行
第19行     system("pause");
第20行     return 0;
第21行 }
```

例程14

```
第1行  #include <iostream>
第2行  #include<string>
第3行  using namespace std;
第4行  void stringSwap(string &a, string &b){
第5行      string tmp = move(a);
第6行      a = move(b);
第7行      b = move(tmp);
第8行  }
第9行  int main()
第10行 {
第11行     string a = "ABC";
第12行     string b = "XYZ";
第13行     stringSwap(a, b);
第14行     cout << a <<endl<< b;
第15行     system("pause");
第16行     return 0;
```

```
第17行  }
```

move的实现与右值引用密切相关，如例程15所示。

例程15

```
第1行  class CMyString
第2行  {
第3行  public:
第4行      // 构造函数
第5行   CMyString(const char *pszSrc = NULL)
第6行   {
第7行    cout << "CMyString(const char *pszSrc = NULL)" << endl;
第8行    if (pszSrc == NULL)
第9行    {
第10行    m_pData = new char[1];
第11行    *m_pData = '\0';
第12行   }
第13行   else
第14行   {
第15行    m_pData = new char[strlen(pszSrc)+1];
第16行    strcpy(m_pData, pszSrc);
第17行   }
第18行  }
第19行
第20行     // 拷贝构造函数
第21行  CMyString(const CMyString &s)
第22行  {
第23行   cout << "CMyString(const CMyString &s)" << endl;
第24行   m_pData = new char[strlen(s.m_pData)+1];
第25行   strcpy(m_pData, s.m_pData);
第26行  }
第27行
第28行     // move构造函数
第29行  CMyString(CMyString &&s)
第30行  {
第31行   cout << "CMyString(CMyString &&s)" << endl;
第32行   m_pData = s.m_pData;
第33行   s.m_pData = NULL;
第34行  }
第35行
第36行     // 析构函数
第37行  ~CMyString()
第38行  {
第39行   cout << "~CMyString()" << endl;
第40行   delete [] m_pData;
```

```
第41行     m_pData = NULL;
第42行    }
第43行
第44行      // 拷贝赋值函数
第45行    CMyString &operator =(const CMyString &s)
第46行    {
第47行     cout << "CMyString &operator =(const CMyString &s)" << endl;
第48行     if (this != &s)
第49行     {
第50行      delete [] m_pData;
第51行      m_pData = new char[strlen(s.m_pData)+1];
第52行      strcpy(m_pData, s.m_pData);
第53行     }
第54行     return *this;
第55行    }
第56行
第57行      // move赋值函数
第58行    CMyString &operator =(CMyString &&s)
第59行    {
第60行     cout << "CMyString &operator =(CMyString &&s)" << endl;
第61行     if (this != &s)
第62行     {
第63行      delete [] m_pData;
第64行      m_pData = s.m_pData;
第65行      s.m_pData = NULL;
第66行     }
第67行     return *this;
第68行    }
第69行
第70行   private:
第71行    char *m_pData;
第72行   };
```

第二节　Lambda函数对象

许多编程语言支持匿名函数，这些函数有函数体，但是没有函数名。C++11中的Lambda是与匿名函数相关的编程技术。Lambda隐式定义函数对象类(class)并构造函数对象(Function Object)。

1、初识Lambda函数对象

例程16是Lambda的是一个应用示例，注意比对第26行、第27行，会发现createNum和[]{return rand() % 10; }地位相同，其中createNum是函数指针，指向已定义的createNum()函数，[]{return rand() % 10; }即是Lambda函数。如果进一步观察二者的代码更会发现，其功能也相同。和传统函数相比，Lambda函数的实现generate()函数中，直接阅读代码，即可知道其功能，而函数或者函数对象模式，则将功能定义在其外，难以直接就近知道其实现代码及其功能。从这个意义上讲，应用Lambda函数更加简捷，Lambda函数如同一个匿名函数。Lambda函数也可以有参数，如第31行所示。第33-35行代码是用STL函数对象、自定义函数的指针以及自定义函数对象实现，与其相比，Lambda函数更加直观。

例程16

```
第1行   #include<iostream>
```

```
第2行  #include<vector>
第3行  #include<algorithm>

第4行  #include<functional>
第5行  #include<iterator>
第6行  using namespace std;
第7行  int createNum(){
第8行      return rand() % 10;
第9行  }
第10行 bool isPrime(int N){//判断是否质数，代码未优化
第11行     if (N == 0 || N == 1)return false;
第12行     for(int iLoop = 2; iLoop < N; ++iLoop)if (N%iLoop == 0)return false;
第13行     return true;
第14行 }
第15行
第16行 template<typename T>
第17行 class greaterThan{//大于，自定义类
第18行 public:
第19行     greaterThan(T x) :num(x){};
第20行     bool operator()(T x){ return x>num; }
第21行 private:
第22行     T num;
第23行 };
第24行 int main(){
第25行     vector<int> myVec(10);
第26行     generate(myVec.begin(), myVec.end(),createNum);//应用Lambda函数
第27行     generate(myVec.begin(), myVec.end(), []{return rand() % 10; });//应用Lambda函数
第28行     copy(myVec.begin(), myVec.end(), ostream_iterator<int>(cout, " ")); cout << "\n\n";
第29行

第30行     //统计被3整除数的数量，应用Lambda函数
第31行     int countA=count_if(myVec.begin(), myVec.end(), [](int x){return x %2 == 0; });
第32行
第33行     int countB = count_if(myVec.begin(), myVec.end(), bind2nd(less<int>(), 5));//STL定义的函数对象
第34行     int countC = count_if(myVec.begin(), myVec.end(), isPrime);//自定义的处理函数

第35行     int countD = count_if(myVec.begin(), myVec.end(), greaterThan<int>(5));//自定义的类
第36行
第37行     return 0;
第38行 }
```

在编写代码时，可能使用函数指针或函数对象解决问题和执行计算，特别是当使用 STL 算法时。函数指针和函数对象各有优缺点。函数指针具有最低的语法开销，但不能保留状态，而函数对象能够维护状态，但需要额外的类定义语法开销。

Lambda函数对象具有函数指针和函数对象的优点并避免其缺点。类似函数对象，Lambda是灵活且可以维护状态，但是，不同函数对象，其简洁语法不需要类定义。和等效的函数对象相比，Lambda可以写出不太复杂且不容易出错的代码。

2、Lambda函数对象本质

观察例程17，其中第10行将一个Lambda函数赋值给自动变量whatIs，第11行用typeid(whatIs).name()反馈whatIs的类型名称，实际

上是Lambda函数的类型，从输出可以看出其类型为以lambda开始的一个class。

例程17

```
第1行  #include<iostream>
第2行  #include<typeinfo>
第3行  using namespace std;
第4行  class udC{
第5行  public:
第6行      int udFun(int X){ return X; }
第7行      int operator()(){return rand() % 100; }
第8行  };
第9行  int main(){
第10行     auto whatIs = []{return rand() % 100; };//应用Lambda函数
第11行     cout << typeid(whatIs).name() << endl;
第12行     //输出：class <lambda_9fdee1da7eb56b805fe97de0a142d4ba>
第13行
第14行     auto myUDc = udC();
第15行     cout << typeid(myUDc).name() << endl;//输出：class udC
第16行
第17行     cout << sizeof(whatIs) << endl;//输出：1
第18行     cout << sizeof(udC) << endl;//输出：1
第19行
第20行     cout << whatIs.operator()() << endl;//输出：41(随机，可能因机器不同而不同)
第21行     cout << udC().operator()() << endl;//输出：67
第22行
第23行     cout << whatIs() << endl;//输出：34(随机，可能因机器不同而不同)
第24行     cout << udC()()<< endl;//输出：0(随机，可能因机器不同而不同)
第25行     cout << udC().udFun(10) << endl;//输出：10
第26行
第27行     return 0;
第28行 }
```

whatIs不是一种class类型，而是class类型的实例或者对象，如第20、21行以及第23、24、25行所示，从中可以看出whatIs相当于udC()，而udC()是自定义类的实例化，生成匿名对象。whatIs()相当于调用对象中的重载括号运算符函数，与whatIs().operator()()功能相同。

3、Lambda函数对象更多用法

完整的Lambda函数对象的语法如下所示，其中throw()表示函数的异常规约，->rtnType表示函数的返回值，其他部分将随后讲述。观察例程18有助于更完整理解Lambda函数对象。

```
[Capture_List](Parameter_List) mutable throw()->rtnType{
  //Statement
}
```

例程18

```
第1行  #include<iostream>
第2行  #include<algorithm>
第3行  #include<vector>
```

```
第4行  using namespace std;
第5行  bool isPrime(int N){
第6行      if (N == 0 || N == 1)return false;
第7行      int LastNum = N / 2+1;
第8行      for (int i = 2; i < LastNum; ++i)if (N%i == 0)return false;
第9行      return true;
第10行 }

第11行 int main(){
第12行     vector<int> myVec;
第13行     for (int i = 0; i < 100; ++i)
第14行         myVec.push_back(i);
第15行
第16行     int numEven = 0, numOdd = 0, numPrime = 0;
第17行     for_each(myVec.begin(), myVec.end(),
第18行         [&numEven, &numOdd, &numPrime](int X){
第19行             numEven += (X % 2 == 0);
第20行             numOdd += (X % 2 == 1);
第21行             numPrime += (isPrime(X) == true?1 : 0);
第22行         }
第23行     );
第24行     cout << "偶数=" << numEven << " 奇数=" << numOdd << " 质数=" << numPrime << endl;
第25行
第26行     return 0;
第27行 }
```

Capture_List可以称之为捕获列表，用于捕获Lambda函数对象外的参数。在例程18中，代码[&numEven, &numOdd, &numPrime](int X)的含义是将Lambda函数对象外的numEven、numOdd和numPrime传递到Lambda函数对象内部。当含有&符号时，表示该参数以引用的方式传递到Lambda内部，内部的改变将影响到外部。当Capture_List为空时，表示外部任何变量都不传递到Lambda内部。Capture_List还有其他形式的用法，如例程19所示。

例程19

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  int main(){
第5行      int numA = 100, numB = 200, numC = 300, numSum = 0;
第6行      auto lambdaA = [numA](int x){cout << numA; };//值拷贝方式传递numA
第7行      auto lambdaB = [&numB](int x){return numB += x; };//引用方式传递numB
第8行      cout << lambdaB(10) << endl;//输出：210;
第9行
第10行     auto lambdaC = [=, &numSum]{numSum = numA + numB + numC;};
第11行     lambdaC();
第12行     cout << numSum << endl; //输出：610
第13行
第14行     auto lambdaD = [=]{cout << (numA + numB + numC) << endl; };
```

```
第15行      lambdaD();//输出：610
第16行
第17行      auto lambdaE = [&]{cout << (numA++) << endl; };
第18行      lambdaE();//输出：100
第19行      cout << numA << endl;//输出：101
第20行
第21行      auto lambdaF = [&, numSum](int N)mutable {
第22行          numC += N; numSum += numC; cout << numC <<" "<< numSum << endl;
第23行      };
第24行      lambdaF(100);//输出：400和1010
第25行      cout << numC <<" "<<numSum<< endl;//输出：410 610
第26行
第27行      return 0;
第28行  }
```

从例程19可以看出，&可以用在变量前，也可以单独存在，表示Lambda函数对象外的所有变量都以引用的方式传递到Lambda函数对象内，此时修改该变量的值，则将改变函数对象体外的变量值，如第17-19行所示，代码[&]{cout << (numA++) << endl; }中的&变量Lambda函数对象外的变量都可以以引用的方式传递到函数对象内，numA值被改变，因此第19行输出101。当将&替换为=时，则表示体外所有变量以值拷贝的形式传递到Lambda函数对象内，如第14行所示。第10行代码[=, &numSum]{numSum = numA + numB + numC;}表示除numSum外，其他变量都以只拷贝的形式传递到Lambda函数对象内，而numSum因为前有&因此以引用的方式传递。第21行Lambda函数对象代码[&, numSum](int N)则表示除numSum变量外其他都已引用方式传递，而numSum则以只拷贝的形式传递。

Lambda函数对象还可以用mutable关键字，此时以值拷贝形式传递的变量，也可以在函数内修改其值，但不影响函数外的该变量的值。如第21-23行定义Lambda函数对象，在第24行被执行后，numSum值为1010，但第25行仍然输出为numSum值为610，没有受到该Lambda函数影响。

当Lambda函数对象出现在类类型时，可以将this关键字传递到Lambda函数对象内，如例程19所示，在第11-13行的Lambda函数中，代码[this](int N)中的this传递到Lambda内，因此，可以通过this->num访问类的数据成员或函数成员。

例程20

```
第1行   #include<iostream>
第2行   #include<algorithm>
第3行   #include<vector>
第4行   using namespace std;
第5行
第6行   class udC{
第7行   public:
第8行       udC(int N) :num(N){}
第9行       void printVector(const vector<int> &myVec){
第10行          for_each(myVec.begin(), myVec.end(),
第11行              [this](int N){
第12行              cout << N + this->num << endl;
第13行          }
第14行          );
第15行      }
第16行  private:
第17行      int num;
第18行  };
第19行
```

```
第20行  int main(){
第21行      vector<int> myVec{12,13,14};
第22行      udC myUDc(100);
第23行      myUDc.printVector(myVec);//输出：112,113,114
第24行
第25行      return 0;
第26行  }
```

Lambda函数可以嵌套，如例程21所示。

例程21

```
第1行  #include <iostream>
第2行  using namespace std;

第3行  int main()
第4行  {
第5行      int m = [](int N){
第6行          return [](int M){return M*10;}(N+3);
第7行      }(5);
第8行      cout << m << endl;//输出：80
第9行
第10行     return 0;
第11行 }
```

第三节　function包装器

function包装器与Lambda配合，可以实现高阶函数，高阶函数要求函数能以函数作为参数或能返回函数。如例程22所示。

例程22

```
第1行  #include <iostream>
第2行  #include <functional>
第3行  using namespace std;
第4行  int main()
第5行  {
第6行      auto g = [](int x) -> function<int(int)>{
第7行          return [=](int y) {return x + y; };
第8行      };
第9行
第10行     auto h = [](const function<int(int)>& f, int z){
第11行         return f(z) + 1;
第12行     };
第13行
第14行     cout << g(7)(8) << endl;
第15行     auto a = h(g(7), 8);
第16行
第17行     cout << a << endl;
第18行
```

```
第19行        return 0;
第20行    }
```

第四节　新增库文件

C++11增加了多个库文件，如下表所示。

表1：C++11新增库文件

序号	库文件	描述
1	<random>	用于生成多种分布的随机数；
2	<chrono>	用于处理时间间隔；
3	<tuple>	用于处理多个数据组合，类似pair；
4	<ratio>	用于处理极大数或者极小数；
5	<regex>	用于正则表达式；

1、ratio库

ratio库用于精确地表示在编译时使用的有理数，并且ratio已定义在std名称空间中。有理数的分子和分母数据类型为std:intmax_t，通常为long long类型的别名。由于ratio是编译时特性，因此其使用显得不同寻常，对象的定义与普通对象不同，且不能使用ratio对象的方法，通常使用typedef。有关ratio库的应用示例如例程23所示，第6行是ratio的定义示例，第9行、第10行分别为访问分子和分母(注意格式为myRatio::num，不是myRatio.num)。

例程23

```
第1行    #include<iostream>
第2行    #include<ratio>
第3行    using namespace std;
第4行
第5行    int main(){
第6行        typedef ratio<1,60> myRatio;//注意：这是ratio定义，实为类型别名
第7行
第8行        //num是分子的英文单词numerator的简写；den是denominator的简写
第9行        intmax_t UpNum = myRatio::num;//intmax_t是long long的别名，注意不是myRatio.num
第10行       intmax_t DownNum = myRatio::den;//intmax_t是long long的别名
第11行
第12行       cout << myRatio::num <<"/"<< myRatio::den<<endl;
第13行
第14行       //有理数化简
第15行       typedef ratio<4, 6> myND;
第16行       cout << myND::num << "/" << myND::den << endl;//输出：2/3已经化简
第17行
第18行       //自定义分数
第19行       const intmax_t numU = 1;
第20行       const intmax_t numD = 11;
第21行       typedef ratio<numU, numD> myNumUD;
第22行
第23行       //分数相加
第24行       typedef ratio_add<myRatio, myNumUD>::type Result;
第25行       cout << Result::num << "/" << Result::den << endl;//输出：71/660
```

```
第26行
第27行        //分数关系运算
第28行        typedef ratio_less<myRatio, myNumUD> boolCheck;
第29行        cout << boolalpha << boolCheck::value << endl;//输出：true
第30行
第31行        /*错误用法：
第32行
第33行        myRatio tmp1;
第34行        myNumUD tmp2;
第35行        cout << tmp1 + tmp2 << endl;
第36行        cout<<myRatio+myNumUD<<endl;
第37行
第38行        */
第39行
第40行        /*以下用法错误
第41行            本意是表示1/5，但ratio是编译时常量，即分子分母在编译时确定，
第42行            而U和D都是变量，因此错误。
第43行
第44行            intmax_t U=1;
第45行            intmax_t D=5;
第46行            typedef ratio<U,D> myUD;
第47行
第48行        */
第49行        //ratio提供的SI(国际标准单位)的typedef，如giga定义为：typedef ratio<1000000000,1>giga
第50行        cout << giga::num << "/" << giga::den << endl;
第51行
第52行        return 0;
第53行    }
```

ratio是编译时常量，即分子和分母时在编译时确定，因此第44-46行用法错误，其对应的正确用法是第19-21行。ratio总是简化的，对于一个有理数ratio<n,d>总是计算其最大公约数，如第15-16行所示。

ratio库支持有理数加减乘除运算。由于所有这些操作都是编译时进行，因此不能使用标准的算术运算符，而应该使用特定的模板和typedef结合。ratio的算术模板包括：ratio_add、ratio_substract、ratio_multiply和ratio_divide，这些模板将结果计算为新的ratio类型。如例程23第24-25行所示。

ratio库还定义了一些比较模板：ratio_equal、ratio_not_equal、ratio_less、ratio_less_equal、ratio_greater以及ratio_greater_equal。与ratio的算术模板一样，这些模板也是在编译时求值，其结果为std::integral_constant类型。integral_constant是一个struct模板，保存一个类型和编译时常量值，如integral_constant<bool,true>即保存其值为true的bool型常量，integral_constant<int,100>则保存其值为100的int型常量。通过value数据成员可以访问保存在integral_constant的值，如例程23第28-29行所示。

为方便应用，ratio库还提供了一些SI(国际单位)的typedef，如表2所示，其应用如例程23第50行所示。

表2：ratio库预定义的SI类型

ratio的SI类型	描述
typedef ratio<1000000000000000000000000,1> yotta	尧它，10^{24}
typedef ratio<1000000000000000000000,1> zetta	泽它，10^{21}

typedef ratio<1000000000000000000,1> exa	艾可萨，10^{18}
typedef ratio<1000000000000000,1> peta	排它，10^{15}
typedef ratio<1000000000000,1> tera	太拉，10^{12}
typedef ratio<1000000000,1> giga	吉咖，10^{9}
typedef ratio<1000000,1> mega	兆，10^{6}
typedef ratio<1000,1> hecto	千，10^{3}
typedef ratio<100,1> hecto	百，10^{2}
typedef ratio<10,1> deca	十，10^{1}
typedef ratio<1,10> deci	分，10^{-1}
typedef ratio<1,100> centi	厘，10^{-2}
typedef ratio<1,1000> milli	毫，10^{-3}
typedef ratio<1,1000000> micro	微，10^{-6}
typedef ratio<1,1000000000> nano	纳诺，10^{-9}
typedef ratio<1,1000000000000> pico	皮可，10^{-12}
typedef ratio<1,1000000000000000> femto	费姆托，10^{-15}
typedef ratio<1,1000000000000000000> atto	阿托，10^{-18}
typedef ratio<1,1000000000000000000000> zepto	仄普托，10^{-21}
typedef ratio<1,1000000000000000000000000> yocto	幺科托，10^{-24}

2、chrono库文件

chrono库文件主要用于时间操作，主要包含三个类：duration(持续时间)、clock(时钟)、time_point(时间点)。

duration是一个模板类，模板参数为tick(滴答数)和tick period(滴答周期)，表示两个时间点之间的间隔。滴答周期是指两个滴答之间的间隔秒数，是一个编译时ratio常量。

clock类由time_point和duration组成。在chrono库文件中定义了3中clock，分别是system_clock(系统时钟)、steady_clock(稳定时钟，其time_point绝不递减)和high_resolution_clock(高分时钟，滴答周期达到最小值)。另外，high_resolution_clock可能是steady_clock或system_clock的别名，取决于具体的编译器。每个clock都有一个静态的now()函数用于获得类型为time_point的当前时间。另外，system_clock定义了两个静态的辅助函数用于time_point和time_t(C风格时间表示法)之间的相互转换。to_time_t()将给定的time_point转换为一个time_t；from_time_t()则将time_t转换为time_point。time_t类型在<time.h>中定义。

time_point类表示时间中的一个点，存储为相对于纪元(epoch)的duration，相当于相对于某个基点的偏移量，一般采用1970年1月1日作为基点，其度量单位可以是秒或100纳秒，具体由操作系统和编译器决定。time_point类中有time_since_epoch()成员函数，返回相对于epoch的偏移量。另外，time_point支持有意义的算术运算符，如：+、-、+=、-=以及两个时点的比较运算符。例程24是chrono库的应用示例。

例程24

```
第1行  #include<iostream>
第2行  #include<ctime>
第3行  #include<chrono>
第4行  using namespace std;
第5行  using namespace std::chrono;
第6行
第7行  int main(){
```

```
第8行      system_clock::time_point today1 = high_resolution_clock::now();
第9行      system_clock::duration dtn1 = today1.time_since_epoch();
第10行     for (auto i = 0; i < 1000000; ++i)sqrt(sin(i)*cos(i));
第11行     system_clock::time_point today2 = high_resolution_clock::now();
第12行
第13行     cout << (today2 - today1).count() << endl;
第14行
第15行     cout << std::chrono::duration_cast<std::chrono::nanoseconds> (today2 - today1).count() <<"纳秒"<< endl;
第16行     cout << std::chrono::duration_cast<std::chrono::microseconds> (today2 - today1).count() << "微秒"<<endl;
第17行     cout << std::chrono::duration_cast<std::chrono::milliseconds> (today2 - today1).count() <<"毫秒"<< endl;
第18行
第19行     return 0;
第20行   }
```

3、random库

4、regex库

正则表达式(Regular Expression)，是计算机科学的一个概念。正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。在很多文本编辑器里，正则表达式通常被用来检索、替换那些符合某个模式的文本。

元字符	描述
\	将下一个字符标记为一个特殊字符、或一个原义字符、或一个向后引用、或一个八进制转义符。例如，“\\n”匹配\n。“\n”匹配换行符。序列“\\”匹配“\”而“\(”则匹配“(”。
^	匹配输入字符串的开始位置。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。
$	匹配输入字符串的结束位置。如果设置了RegExp对象的Multiline属性，$也匹配“\n”或“\r”之前的位置。
*	匹配前面的子表达式零次或多次(大于等于0次)。例如，zo*能匹配“z”，“zo”以及“zoo”。*等价于{0,}。
+	匹配前面的子表达式一次或多次(大于等于1次）。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。
?	匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“do”或“does”中的“do”。?等价于{0,1}。
{n}	n是一个非负整数。匹配确定的n次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个o。
{n,}	n是一个非负整数。至少匹配n次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“foooood”中的所有o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o*”。
{n,m}	m和n均为非负整数，其中n<=m。最少匹配n次且最多匹配m次。例如，“o{1,3}”将匹配“fooooood”中的前三个o。“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。
?	当该字符紧跟在任何一个其他限制符（*,+,?，{n}，{n,}，{n,m}）后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“oooo”，“o+?”将匹配单个“o”，而“o+”将匹配所有“o”。
.点	匹配除“\r\n”之外的任何单个字符。要匹配包括“\r\n”在内的任何字符，请使用像“[\s\S]”的模式。
(pattern)	匹配pattern并获取这一匹配。所获取的匹配可以从产生的Matches集合得到，在VBScript中使用SubMatches集合，在JScript中则使用$0…$9属性。要匹配圆括号字符，请使用“\(”或“\)”。
(?:pattern)	匹配pattern但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用或字符“(|)”来组合一个模式的各个部分是很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。
(?=pattern)	正向肯定预查，在任何匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?!pattern)	正向否定预查，在任何不匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹

	配“Windows2000”中的“Windows”。
(?<=pattern)	反向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。
(?<!pattern)	反向否定预查，与正向否定预查类似，只是方向相反。例如“(?<!95|98|NT|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。
x|y	匹配x或y。例如，“z|food”能匹配“z”或“food”。“(z|f)ood”则匹配“zood”或“food”。
[xyz]	字符集合。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。
[^xyz]	负值字符集合。匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“plin”。
[a-z]	字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。 注意:只有连字符在字符组内部时,并且出现在两个字符之间时,才能表示字符的范围; 如果出字符组的开头,则只能表示连字符本身.
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”。
\B	匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。
\cx	匹配由x指明的控制字符。例如，\cM匹配一个Control-M或回车符。x的值必须为A-Z或a-z之一。否则，将c视为一个原义的“c”字符。
\d	匹配一个数字字符。等价于[0-9]。
\D	匹配一个非数字字符。等价于[^0-9]。
\f	匹配一个换页符。等价于\x0c和\cL。
\n	匹配一个换行符。等价于\x0a和\cJ。
\r	匹配一个回车符。等价于\x0d和\cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于[\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于[^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于\x09和\cI。
\v	匹配一个垂直制表符。等价于\x0b和\cK。
\w	匹配包括下划线的任何单词字符。等价于“[A-Za-z0-9_]”。
\W	匹配任何非单词字符。等价于“[^A-Za-z0-9_]”。
\xn	匹配n，其中n为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，“\x41”匹配“A”。“\x041”则等价于“\x04&1”。正则表达式中可以使用ASCII编码。
\num	匹配num，其中num是一个正整数。对所获取的匹配的引用。例如，“(.)\1”匹配两个连续的相同字符。
\n	标识一个八进制转义值或一个向后引用。如果\n之前至少n个获取的子表达式，则n为向后引用。否则，如果n为八进制数字（0-7），则n为一个八进制转义值。
\nm	标识一个八进制转义值或一个向后引用。如果\nm之前至少有nm个获得子表达式，则nm为向后引用。如果\nm之前至少有n个获取，则n为一个后跟文字m的向后引用。如果前面的条件都不满足，若n和m均为八进制数字（0-7），则\nm将匹配八进制转义值nm。
\nml	如果n为八进制数字（0-7），且m和l均为八进制数字（0-7），则匹配八进制转义值nml。
\un	匹配n，其中n是一个用四个十六进制数字表示的Unicode字符。例如，\u00A9匹配版权符号（©）。
\< \>	匹配词（word）的开始（\<）和结束（\>）。例如正则表达式\<the\>能够匹配字符串"for the wise"中的"the"，但是不能匹配字符串"otherwise"中的"the"。注意：这个元字符不是所有的软件都支持的。
\(\)	将 \(和 \) 之间的表达式定义为“组”（group），并且将匹配这个表达式的字符保存到一个临时区域（一个正则表达式中最多可以保存9个），它们可以用 \1 到\9 的符号来引用。

|	将两个匹配条件进行逻辑“或”（Or）运算。例如正则表达式(him|her) 匹配"it belongs to him"和"it belongs to her"，但是不能匹配"it belongs to them."。注意：这个元字符不是所有的软件都支持的。
+	匹配1或多个正好在它之前的那个字符。例如正则表达式9+匹配9、99、999等。注意：这个元字符不是所有的软件都支持的。
?	匹配0或1个正好在它之前的那个字符。注意：这个元字符不是所有的软件都支持的。
{i} {i,j}	匹配指定数目的字符，这些字符是在它之前的表达式定义的。例如正则表达式A[0-9]{3} 能够匹配字符"A"后面跟着正好3个数字字符的串，例如A123、A348等，但是不匹配A1234。而正则表达式[0-9]{4,6} 匹配连续的任意4个、5个或者6个数字

例程25

```
#include <regex>
#include <iostream>
#include <string>
using namespace std;
int main(){
    const regex pattern("(\\w+day)");
    // the source text
    std::string weekend = "Saturday and Sunday";
    std::smatch result;
    bool match = regex_search(weekend, result, pattern);
    if (match){
        for (size_t i = 1; i < result.size(); ++i){
            cout << result[i] << endl;
        }
    }

    system("pause");
    return 0;
}
```

例程26

```
#include <regex>
#include <iostream>
#include <string>
using namespace std;
int main(){
    // regular expression
    const std::regex pattern("\\w+day");

    // the source text
    std::string weekend = "Saturday and Sunday, but some Fridays also.";
    const std::sregex_token_iterator end; //需要注意一下这里
    for (std::sregex_token_iterator i(weekend.begin(), weekend.end(), pattern); i != end; ++i){
        std::cout << *i << std::endl;
    }

```

```
    system("pause");
    return 0;
}
```

例程27

```
//下面的例子将元音字母打头的单词前面的a替换为an：
#include <regex>
#include <iostream>
#include <string>

int main(){
    // text to transform
    std::string text = "This is a element and this a unique ID.";

    // regular expression with two capture groups

    const std::regex pattern("(\\ba (a|e|i|u|o))+");

    // the pattern for the transformation, using the second
    // capture group
    std::string replace = "an $2";

    std::string newtext = std::regex_replace(text, pattern, replace);
    std::cout << newtext << std::endl;

    system("pause");
    return 0;

}
```

5、tuple库

C++11引入的std::tuple类定义在<tuple>库文件中，与std::pair类似，是std::pair的泛化，允许存储任意类型任意数量的值。和pair一样，tuple的数值及其类型都必须在编译时确定，并且固定不变。例程28tuple的应用示例。

例程28

```
#include<iostream>
#include<string>
#include<tuple>
#include<typeinfo>
using namespace std;

int main(){
    tuple<string, string, int> myTuple("John", "010110", 23);
    cout <<"Name:"<<get<0>(myTuple)
        <<" ID:"<<get<1>(myTuple)
        <<" Year Old:"<<get<2>(myTuple)<<endl;

```

```
第13行     typedef tuple<unsigned char, unsigned char, unsigned char> tupleRGB;
第14行     tupleRGB myRGB(10, 20, 220);
第15行     cout << tuple_size<tupleRGB>::value << endl;
第16行
第17行     auto udTuple=make_tuple(100, 'c', true, 23.45, "xyz");
第18行     cout << get<4>(udTuple) << endl;
第19行
第20行     string myName;
第21行     string myID;
第22行     int myOld;
第23行     tie(myName, myID, myOld) = myTuple;
第24行     cout<<"Name:"<<myName<<" ID:"<<myID<<" Year Old:"<<myOld<<endl;
第25行
第26行     get<2>(myTuple) = 30;
第27行     tie(myName, myID, myOld) = myTuple;
第28行     cout << "Name:" << myName << " ID:" << myID << " Year Old:" << myOld << endl;
第29行
第30行     auto myTupleOther = make_tuple(ref(myName), ref(myID), cref(myOld));
第31行     tie(myName, myID, myOld) = myTupleOther;
第32行     cout << "Name:" << myName << " ID:" << myID << " Year Old:" << myOld << endl;
第33行
第34行
第35行     return 0;
第36行   }
```