2018-2019年度第二学期 00106501

计算机图形学

童 伟 华　管理科研楼1205室

E-mail: tongwh@ustc.edu.cn

中国科学技术大学　数学科学学院

http://math.ustc.edu.cn/

附讲四　C/C++编程（三）

C++特点

- C++是面对对象（object oriented）编程语言
  - 纯面向对象语言：指不管什么东西，都应该存在于对象之中，例如JAVA和Small Talk语言
- 三大特性：
  - 封装（encapsulation）
  - 继承（inheritance）
  - 多态（polymorphism）
- 其它特性：
  - 动态生成（dynamic creation）
  - 异常处理（exception handling）
  - 泛型编程（template）

泛型编程（generic programming）

- Wiki: In the simplest definition, generic programming is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters. This approach, pioneered by Ada in 1983, permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication. Software entities are known as templates in C++.

泛型编程（generic programming）

- Wiki: The term generic programming was originally coined by David Musser and Alexander Stepanovin a more specific sense than the above, to describe an approach to software decomposition whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalised as concepts, analogously to the abstraction of algebraic theories in abstract algebra. The best known example is the Standard Template Library (STL) in which is developed a theory of iterators which is used to decouple sequence data structures and the algorithms operating on them.

模板（template）

- 重用：继承与组合提供了一种重用对象代码的方法，而模板提供了一种重用源代码的方法
- 模板定义了一族基于参数（模板参数，template parameter）的类或函数，是类或函数的推广，类模板的实例化（instantiated）生成类，函数模板的实例化生成函数，类模板常被用于构建容器（containers）
- C++中模板
  - 类模板（支持继承）
  - 函数模板
  - STL
- 特征：
  - 类型安全（type-safe）
  - 编译时展开（expanded at compile-time）

类模板

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

template<class T>
class Array {
   enum { size = 100 };
   T A[size];
public:
   T& operator[](int index) {
       if (index < 0 || index >= size)
              cout << "Index out of range" << endl;
       return A[index];
   }
};
```

类模板

```
int main() {
   Array<int> ia;
   Array<float> fa;
   for(int i = 0; i < 20; i++) {
       ia[i] = i * i;
       fa[i] = float(i) * 1.414;
   }
   for(int j = 0; j < 20; j++)
       cout << j << ": " << ia[j]
       << ", " << fa[j] << endl;

   getchar();
}
```

函数模板

```
#ifndef STRINGCONV_H
#define STRINGCONV_H
#include <string>
#include <sstream>
template<typename T>
T fromString(const std::string& s) {
   std::istringstream is(s);
   T t;
   is >> t;
   return t;
}
template<typename T>
std::string toString(const T& t) {
   std::ostringstream s;
   s << t;
   return s.str();
}
#endif // STRINGCONV_H
```

函数模板

```
#include "StringConv.h"
#include <iostream>
#include <complex>
using namespace std;
int main() {
   int i = 1234;
   cout << "i == \"" << toString(i) << "\"\n";
   float x = 567.89f;
   cout << "x == \"" << toString(x) << "\"\n";
   complex<float> c(1.0, 2.0);
   cout << "c == \"" << toString(c) << "\"\n"; cout << endl;
   i = fromString<int>(string("1234"));
   cout << "i == " << i << endl;
   x = fromString<float>(string("567.89"));
   cout << "x == " << x << endl;
   c = fromString< complex<float> >(string("(1.0,2.0)"));
   cout << "c == " << c << endl;
   return(0);
} ///:~
```

模板（template）

- 优点：
  - 支持源代码级的重用（source code reuse）
  - 类型安全，编译时展开（没有run-time overhead）
- 缺点：
  - 编译器支持（compilers support）
  - 错误消息（poor error messages）
  - 代码膨胀（code bloat）
  - 难于调试（调试时难于定位代码）
  - 不能隐藏实现（模板定义与实现都必须提供源代码）
  - 编译时间长

模板与多文件工程

- 模板这种类似宏（macro-like）的功能，对多文件工程有一定的限制：函数或类模板的实现（定义）必须与原型声明在同一个文件中。也就是说我们不能再将接口（interface）存储在单独的头文件中，而必须将接口和实现放在使用模板的同一个文件中
- 在一个工程中多次包含同时具有声明和实现的模板文件并不会产生链接错误（linkage errors），因为它们只有在需要时才被编译，而兼容模板的编译器应该已经考虑到这种情况，不会生成重复的代码

C++标准库

- C++标准库
  - Language support library
  - Diagnostics library
  - General utilities library
  - Strings library
  - Localization library
  - Containers library
  - Iterators library
  - Algorithms library
  - Numerics library
  - Input/output library
  - Regular expressions library
  - Atomic operations library
  - Thread support library

标准模板库（STL）

- STL之父：Alexander Stepanov，1994年加入C++标准库
- 主要组成部分：
  - 算法（algorithms）
  - 容器（containers）
  - 算子（functors）
  - 迭代器（iterators）

标准模板库（STL）

- **标准容器**
  - 序列化容器（Sequence containers）：array，deque，forward_list，list，vector
  - 关联容器（Associative containers）：map，set
  - 无序关联容器（Unordered associative containers）：unordered_map，unordered_set
  - 容器适配器（Container adaptors）：queue，stack（原来容器的一个受限界面，不提供迭代器）

标准模板库（STL）

- 标准模板库里的算法覆盖了在容器上的各种最具有普遍性的操作，譬如遍历、排序、检索、插入或者删除等
- 分类：
  - 非修改性的序列操作：用于从序列中获取信息或者找出某些元素在序列中的位置
  - 修改性的序列操作：其它操作

标准模板库（STL）

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

int main(int argc, char *argv[])
{
   vector<string> SS;
   SS.push_back("The number is 10");
   SS.push_back("The number is 20");
   SS.push_back("The number is 30");

   cout << "Loop by index:" << endl;
   for (int i=0; i < SS.size(); i++)
   {
       cout << SS[i] << endl;
   }
   cout << endl;
```

标准模板库（STL）

```
cout << "Constant Iterator:" << endl;
vector<string>::const_iterator ci;
for (ci=SS.begin(); ci!=SS.end(); ci++)
{
    cout << *ci << endl;
}
cout << endl;

cout << "Reverse Iterator:" << endl;
vector<string>::reverse_iterator ri;
for (ri=SS.rbegin(); ri!=SS.rend(); ++ri)
{
    cout << *ri << endl;
}
cout << endl;
```

标准模板库（STL）

```
    cout << "Sample Output:" << endl;
  cout << SS.size() << endl;
  cout << SS[2] << endl;
  swap(SS[0], SS[2]);
  cout << SS[2] << endl;

  return(0);
}
```

异常处理

- 错误处理：一个库的作者可以检查出运行时错误，但一般却根本不知道怎样去处理它们；库的用户知道怎样对付这些错误，但却无法去检查它们。提供异常的概念就是为了有助于处理这类异常，希望它的（直接或间接）调用者能够处理这个问题，能明确地把错误处理代码从“正常”代码中分离出来
- Exception handling provides a way of transferring control and information from a point in the execution of a thread to an exception handler associated with a point previously passed by the execution

异常处理

- 一个异常：某个用于表示异常发生的类的一个对象
- 异常抛出：当检查到一个异常时，代码段throw语句一个对象，即异常
- 异常捕获：一个代码段用catch子句表明它要处理的某个异常
- 处理程序：异常抛出后获得执行权的称为处理程序（handler）
- 一个throw的作用就是导致堆栈的一系列回退，直到找到某个适当的catch（在某个直接或间接调用了抛出异常的那个函数的函数里）

异常处理

- 基本语法：

```
try
{
// code that may generate exceptions
}
catch(type1 id1)
{
// handle exceptions of type1
}
catch(type2 id2)
{
// handle exceptions of type2
}
// etc...
```

异常处理

```
#include <iostream>
using namespace std;
class Matherr{
   //...
public:
   virtual void debug_print( ) const {
        cerr << "Matherror";
   }
};

class Int_overflow: public Matherr{
   const char* op;
   int a1, a2;
public:
   Int_overflow(const char*p, int a, int b) {op=p; a1=a; a2=b;}
   virtual void debug_print() const {
        cerr << op << '(' << a1 << ',' << a2 << ')';
   }
   //...
};
```

异常处理

```
int add(int x,int y)
{
   if ((x>0 && y>0 && x>INT_MAX-y) || (x<0 && y<0 && x<INT_MIN-y))
        throw Int_overflow("+", x, y);
   return x+y;  //x+y will not overflow
}

int main(int argc, char *argv[])
{
   try {
        int i1 = add(1,2);
        int i2 = add(INT_MAX,2);
        int i3 = add(INT_MAX,2);  //here we go!
   }
   catch (Matherr&m){
        //...
        m.debug_print();
   }

   return(0);
}
```

C++语言回顾

- From International Standard Programming Language C++ 2011（C++11）：C++ is a general purpose programming language based on the C programming language as specified in ISO/IEC 9899:1999. In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities

C++简史

- 发展初期：
  - C with Classes-1979年，Dr. BjarneStroustrup博士在C语言中加入类似Simula语言的类机制，
  - 1983年，在AT&T实验室内部投入使用，正式命名：C++
  - 1985分，商业版本正式发布
  - 上世纪八十年代后期，模板和异常处理功能加入
  - 上世纪九十年度－至今：与C语言一起成为被广泛使用的编程语言
- C++国际标准：
  - 1998年发布C++98(ISO/IEC 1988-1998)
  - 2003年发布C++03(ISO/IEC 14882)
  - 2011年发布C++11(ISO/IEC 14882:2011)
- For more information：

  http://www.stroustrup.com/

C++与其它主流语言的比较

- 比较标准：高效性、灵活性、抽象性（面向对象、泛型编程等）、生成能力（自动化服务、工具）
- C++11核心：性能与抽象的平衡

Where the Focus Is

	Efficiency BOT – 1999, 2009 – EOT	Flexibility	Abstraction (OO, Generics)	Productivity (Automatic Services, Tools)
C = efficient high-level portable code. Structs, functions.	●	●	*non-goal*	*non-goal*
C++ = C + efficient abstraction. Classes, templates.	●	●	●	*non-goal*
Java, C# = productivity. Mandatory metadata & GC, JIT compilation.	*at the expense of*	*at the expense of*	●	●

Thanks for your attention!