粒子物理与核物理实验中的数据分析

张黎明

清华大学

第二讲： C++

本讲摘要

- 面对对象的程序设计
- C++的类，实例，指针
- 类的继承，重用
- 朋友类
- 类的多态
- 编译并执行C++程序
- Makefile简介

后继课程的准备要求

- 有能够使用的Linux，自行安装或可以登录其他服务器
- 可以编译链接使用C++
- 安装ROOT
- 学会安装一些支撑软件（各个系统不一样）
- 稍后我们要安装Geant4

大家可以提前着手准备

本节课的例子

登录到一个Linux

- Windows to Linux

使用ssh客户端程序(XManager,SecureCRT, putty...)

- Linux to Linux
- Virtual Box

安装新的工具包(Ubuntu, CentOS不一样yum?)

- sudo apt-get install g++
- sudo apt-get install emacs23 (或vi)

C++速成：HelloWorld

首先用emacs/vi，编写包含以下内容的文件 HelloWorld.cc

```
// A C++ program
#include <iostream>
using namespace std;
int main(){
  cout << "Hello World!" << endl;
  return 0;
}
```

```
emacs -nw HelloWorld.cc
   editing …
save and exit:
   ctrl+x, ctrl+c
save:
   ctrl+x, ctrl+w
```

然后对文件进行编译形成机器可读的代码：

```
> g++ -o HelloWorld HelloWorld.cc
```

调用编译器 (c++)　　输出的文件名　　源代码

最后执行程序

```
>./HelloWorld
Hello World!
```

← 用户键入(注意：>为系统提示符)

← 计算机显示结果

面对对象的程序设计

- C++
  - 对象：英语Object
    应该翻译成“客体或物体”
    面对客体的程序设计，
    基于物理特征的编程。
  - 直接描述一个物体，描述它的属性，功能。
  - 思路更直接，顺畅
    （不用再解应用题）
  - C++编译器，可以翻译，解释我们的逻辑给电脑

面对对象设计
C++

狗
- 颜色
- 名称
- 品种
- 当前行为

粒子
- 类型
- 寿命
- 质量
- 位置
- 动量

C++速成，例子2. VolCuboid

类的定义和使用的重要关键点

0. 关键字 class
1. 构造函数，
2. 析构函数
3. 成员变量
4. 成员函数
5. 实现成员函数的功能
6. 类要生成实例才能使用

C++类（例子）

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

class Line
{
   public:
     void setLength( double len );
     double getLength( void );
     Line(double len); // 这是构造函数
     ~Line(); // 这是析构函数声明
   private:
     double length;
};
```

C++类（构造函数&析构函数）

类的构造函数会在每次创建类的新对象时执行

- 构造函数的名称与类的名称是完全相同的
- 默认的构造函数没有任何参数，如需要，也可以带有参数
- 不会返回任何类型
- 构造函数可用于为某些成员变量(*private*)设置初始值

```
Line::Line( double len): length(len)
{
    cout << "Object is being created, length = " << len << endl;
}
```

上面的语法等同于如下语法：

```
Line::Line( double len)
{
    length = len;
    cout << "Object is being created, length = " << len << endl;
}
```

C++类（构造函数&析构函数）

类的析构函数会在每次删除所创建的对象时执行

- 析构函数的名称与类的名称是完全相同的，只是在前面加了个波浪号（~）作为前缀
- 它不会返回任何值，不能带有任何参数
- 析构函数有助于在跳出程序（比如关闭文件、释放内存等）前释放资源

```
Line::~Line(void)
{
    cout << "Object is being deleted" << endl;
}
```

C++类（例子）

```
void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// 程序的主函数
int main( )
{
   Line line(10.0);

   // 获取默认设置的长度
   cout << "Length of line : " << line.getLength() <<endl;
   // 再次设置长度
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
Object is being deleted
```

C++类（数据封装）

- 数据封装是面向对象编程的一个重要特点，它防止函数直接访问类类型的内部成员。
- 类成员的访问限制是通过在类主体内部对各个区域标记 public、private、protected （访问修饰符）来指定的。

- public：公有成员在程序中类的外部是可访问的。
- private：（默认，封装最严密）私有成员在类的外部是不可访问的。只有类和友元函数可以访问私有成员。
- protected：保护成员与私有成员十分相似，但有一点不同，保护成员在派生类中是可访问的。

多选题 1分

设置

派生类可以访问基类的哪些信息

A 私有成员

B 公有成员

C 保护成员

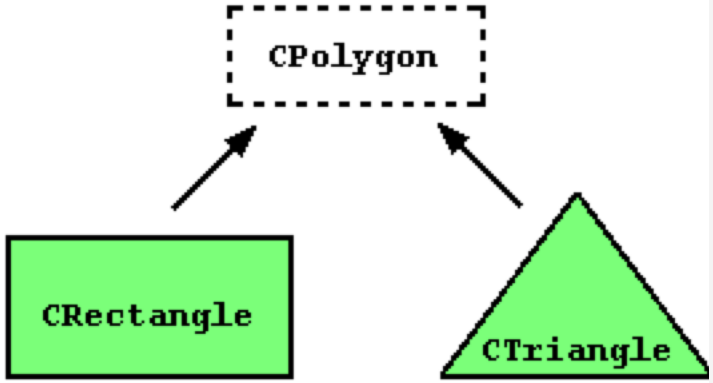
D 友元类

提交

类的重要概念和应用

继承 Inheritance

- ***Inheritance*** 是面向对象编程最重要的特性之一
- 类可以被扩展，即可以创建一个类使其保持“基类”的所有属性 ➔ ***inheritance***
- 关于 ***“基类” base class*** 和 ***“派生类” derived class***: 派生类继承基类的成员，以此为基础还可以添加新的成员。




```
class CPolygon {/*…*/};

class CRectangle: public CPolygon
{/*……*/};

class CTriangle: public CPolygon
{/*……*/};
```

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

继承示例

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {    //base class
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};
class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};
class Triangle: public Polygon {
  public:   int area ()
      { return width * height / 2; }
};
```

//derived class

//derived class

```
int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

- *Polygon* 是基类
- *Rectangle* 和 *Triangle* 是基类 *Polygon* 的派生类
- *Rectangle* 和 *Triangle* 可直接使用 *Polygon* 的**public**和**protected**成员

- **注：派生类不可访问基类的 private 成员**

继承的要点

基类的哪些属性被派生类继承了？

- 原则上，派生类会继承基类的所有成员，除了基类的:
  - **constructors** 和 **destructor**
  - assignment operator members (operator=)
  - friends
  - **private members**

多重继承

- 派生类可以继承自多个基类，不同基类之间用逗号分隔。

```
class Rectangle: public Polygon, public Output;
class Triangle: public Polygon, public Output;
```

Polymorphism (多态性)

- 多态按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联时，就会用到多态。
- C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。

指向基类的指针

```
base class:     Mother
derived class: Daughter

Daughter myD;
Mother *myM = &myD;
```

指向基类的指针

```
// pointers to base class
#include <iostream>
using namespace std;
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};
class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};
class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

- **ppoly1** and **ppoly2** are pointers of **Polygon** class
- They are assigned the addresses of **rect** and **trgl**, objects of type **Rectangle** and **Triangle**
- **They can only access members in base class!**

ppoly1->area(); 

如何访问函数area()

- If **area()** is defined in the base class **Polygon**…
- But the implementations of **area()** in **Rectangle** and **Polygon** are different

- **Virtual member** as a solution:
  - A member function that can be redefined in a derived class, while preserving its calling properties through references
  - The syntax for a function to become virtual is to precede its declaration with the **virtual** keyword

虚成员 virtual members 示例

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};
```

```
class Triangle: public Polygon {
  public:
    int area () { return (width * height / 2); }
};
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

想想我们模拟粒子时候是不是很方便！

question: what are the output?

去掉virtual

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};
```

```
class Triangle: public Polygon {
  public:
    int area () { return (width * height / 2); }
};
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

question: what are the output?

抽象基类

- **Abstract base classes** are classes that can only be used as base classes
- They are allowed to have virtual member functions without definition (known as **pure virtual functions**)
- The syntax is to replace their definition by =0

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

- Abstract base classes cannot be used to instantiate objects

  Polygon mypolygon; 

  Polygon *ppolygon; 

友元函数示例

```
// friend functions
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle duplicate (const Rectangle&);
};

Rectangle duplicate (const Rectangle& param){
  Rectangle res;
  res.width = param.width*2;
  res.height = param.height*2;
  return res;
}
```

Default is Private

```
int main () {
  Rectangle foo;
  Rectangle bar (2,3);
  foo = duplicate (bar);
  cout << foo.area() << '\n';
  return 0;
}
```

- *duplicate* is a friend function of class Rectangle
- It returns an object of class *Rectangle*
- It can access private data member of class *Rectangle* (*width, height*)

友元 Friendships

- 原则上，不能从声明它们的同一类之外访问类的私有 ***private***成员和保护 ***protected***成员。但是，此规则不适用于“友元，friends”。
- 友元可以是一个函数(友元函数)；也可以是一个类(友元类)。
- 友元需要在类定义中声明，并使用关键字***friend***
- 类的友元定义在类外部，但有权访问类的所有私有成员和保护成员。
- 友元函数并不是类成员函数。

友元类示例

```
// friend class
#include <iostream>
using namespace std;
class Square;

class Rectangle {
    int width, height;
  public:
    int area ()
      {return (width * height);}
    void convert (Square a);
};

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};
```

← what?

```
void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

- *Rectangle* is a friend class of class *Square*, and can access class *Square*'s data member(*sides*)
- Notice:
  1) Direction of friendship
  2) friendship NOT transitive

使用刚才的这个例子：

1. 利用g++来编译，而且写成了脚本

例如 Lec2/VolCuboid/build.sh 中的内容

```
#!/bin/tcsh

g++ -o bin/try -Iinclude/ src/*.cc
```

2. 这样脚本还不够智能和快捷，而且功能太差，我们要使用Makefile

Lec2/VolCuboid/Makefile
try：
> make
> bin/VolCub （或者 ./bin/VolCub)

编译，链接

在Lec2/VolCuboid/中展示了多种编译链接的方式

build.sh:

```
g++ -o bin/try -Iinclude/ src/*.cc
```

compile.sh:

```
#!/bin/bash
#### compile cpp programs

g++ -c -I./include/ src/*.cc
g++ -o bin/try *.o
rm -f *.o
```

一个简单的Makefile

Makefile.easy

```
default: hello

hello:
        g++ -o bin/hello -Iinclude/ src/*.cc
clean:
        rm -f obj/*.o bin/*
```

在make命令后可以选择哪一个Makefile

> make -f Makefile.easy

还能选择哪个make目标

> make clean -f Makefile.easy

> make hello -f Makefile.easy

一个复杂一些Makefile

Lec2/VolCuboid/Makefile

语法很复杂，但需要改动的地方很少

```
# # setup control #
TOP := $(shell pwd)/
OBJ := $(TOP)obj/
BIN := $(TOP)bin/
SRC := $(TOP)src/
INCLUDE := $(TOP)include/
#CPPLIBS =
#INCLUDE+=

# # set up compilers #
CPP = g++
CPPFLAGS = -O -Wall -fPIC -I$(INCLUDE)

######## Make Executables #####
all: VolCub
VolCub : $(patsubst $(SRC)%.cc,$(OBJ)%.o,$(wildcard $(SRC)*.cc))
        $(CPP) $^ $(CPPLIBS) -o $(BIN)$(notdir $@)
@echo
#########################
$(OBJ)%.o : $(SRC)%.cc
        $(CPP) $(CPPFLAGS) -c $(SRC)$(notdir $<) -o $(OBJ)$(notdir $@)
@echo
.PHONY:clean
        clean: rm -f $(OBJ)*.o rm -f $(BIN)*
```

头文件或者库文件目录

g++命令的参数

可执行文件

C++后缀,如所有.cc改为.o

A makefile with external libraries

```
# An example of makefile
TOP      := $(shell pwd)/
OBJ      := $(TOP)obj/
BIN      := $(TOP)bin/
SRC      := $(TOP)src/
INCLUDE   := $(TOP)include/

CPP      = g++
LD      = $(CPP)
CPPFLAGS = -O -Wall -fPIC -I$(INCLUDE)

ROOTCFLAGS    = $(shell root-config --cflags)
ROOTLIBS      = $(shell root-config --libs)
ROOTGLIBS     = $(shell root-config --glibs)
CPPFLAGS += -I$(ROOTCFLAGS)
CPPLIBS = $(ROOTLIBS) $(ROOTGLIBS)

######### Make Executables ############################
all: main
main    : $(patsubst $(SRC)%.cc,$(OBJ)%.o,$(wildcard $(SRC)*.cc))
        $(LD) $^ $(CPPLIBS) -o $(BIN)$(notdir $@)
        @echo

#######################################################
$(OBJ)%.o :     $(SRC)%.cc
        $(CPP)  $(CPPFLAGS) -c $(SRC)$(notdir $<) -o $(OBJ)$(notdir $@)
        @echo

.PHONY:clean
clean:
        rm -f $(OBJ)*.o
        rm -f $(BIN)*
```

What if external libraries like ROOT is used?

What you need to do is to let the compiler know:
1) where is the head file?
2) where is the library file

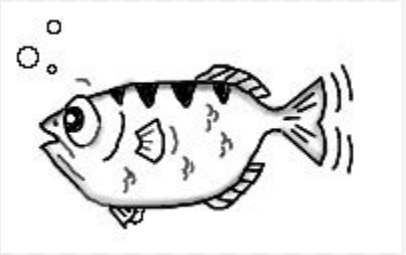
You may do this manually or by the “root-config” command

Try in the command line:
root-config --cflags
root-config --libs
root-config --glibs

GDB: the GNU project debugger

http://www.gnu.org/software/gdb/

调试器（如 GDB）的目的是允许您在它执行时查看"内部"程序的情况，或者另一个程序在崩溃时正在执行的操作。



What GDB can do:

- 启动程序，指定可能影响其行为的任何内容。
- 使程序在指定条件下停止。
- 检查程序停止时发生的情况。
- 更改程序中的事物，以便您可以尝试纠正一个 Bug 的影响，然后继续了解另一个 Bug。

A nice quick-start example:

http://www.cnblogs.com/davidwang456/p/3450532.html

GDB example

- You need compile the program with **"-g"** option

  g++ main.cc -g -Wall -o main

  (-g: produce debug information)

  (-Wall: turns on all optional warnings)
- Start your program with gdb:

  gdb main

  gdb main pid
- Try the following gdb command to see what happens

(gdb) break 11
(gdb) run
(gdb) step
(gdb) list
(gdb) watch n
(gdb) watch result
(gdb) continue
(gdb) backtrace
(gdb) frame numbe

(gdb) b 8 if i==10
(gdb) next
(gdb) info breaks
(gdb) disable
(gdb)
(gdb) print val
(gdb) next
(gdb) continue
(gdb) quit

GDB调试core

https://blog.csdn.net/yockie/article/details/51973740

- 程序执行时, 经常会因为段错误(Segment Fault)而退出, 操作系统会把此程序当前内存信息 dump 到磁盘上, 即生成 core 文件。对 core 进行分析可以很快分析出导致程序 crash 的地方。

- (1) 设置 core 文件大小
  - ulimit –a
  - ulimit -c unlimited
    可以设置 core file size 为无限

```
[zhanglm@hepgpu10 gdb]$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 257378
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
max user processes              (-u) 1024
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

GDB调试core

https://blog.csdn.net/yockie/article/details/51973740

- **(2) 生成稍复杂 core 文件**

```
1  #include <vector>
2  using namespace std;
3
4  int main(int argc, char const *argv[])
5  {
6    vector<int> a;
7    vector<int> b;
8
9    b.push_back(a[0]);
10
11   return 0;
12 }
```

```
[zhanglm@hepgpu10 gdb]$ g++ -g test.C -o test
[zhanglm@hepgpu10 gdb]$ ./test
Segmentation fault (core dumped)
```

```
-rw-------. 1 zhanglm lhcb 476K Mar  9 14:51 core.18983
```

GDB调试core

https://blog.csdn.net/yockie/article/details/51973740

- **(3) gdb test core.18983**

```
[New LWP 18983]
Core was generated by `./test'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000000000040103e in __gnu_cxx::new_allocator<int>::construct (this=0x7ffcf1e34a80,
    __p=0xde8010, __val=@0x0: <error reading variable>)
    at /cvmfs/lhcb.cern.ch/lib/lcg/releases/gcc/4.9.3/x86_64-slc6/include/c++/4.9.3/ext/new_alloca
tor.h:130
130           { ::new((void *)__p) _Tp(__val); }
warning: File "/cvmfs/lhcb.cern.ch/lib/lcg/releases/gcc/4.9.3/x86_64-slc6/lib64/libstdc++.so.6.0.2
0-gdb.py" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/
auto-load".
```

- 简单的程序，gdb给出的#0会直接给出错误的line#
- 复杂的程序（如这里），给出的是C++库函数中的某一行；库函数都是经过千锤百炼，基本不可能出错，所以我们先从自己的程序找起.

GDB调试core

https://blog.csdn.net/yockie/article/details/51973740

- **(3) gdb test core.18983**

使用 backtrace(或者 where) 命令列出当前调用堆栈, 再使用 frame n 命令列出第 n 层堆栈的信息, 就可以直接看到出错的代码语句了

```
(gdb) bt
#0  0x000000000040103e in __gnu_cxx::new_allocator<int>::construct (this=0x7ffcf1e34a80,
    __p=0xde8010, __val=@0x0: <error reading variable>)
    at /cvmfs/lhcb.cern.ch/lib/lcg/releases/gcc/4.9.3/x86_64-slc6/include/c++/4.9.3/ext/new_alloca
tor.h:130
#1  0x0000000000400c4a in __gnu_cxx::__alloc_traits<std::allocator<int> >::construct<int> (
    __a=..., __p=0xde8010, __arg=@0x0: <error reading variable>)
    at /cvmfs/lhcb.cern.ch/lib/lcg/releases/gcc/4.9.3/x86_64-slc6/include/c++/4.9.3/ext/alloc_trai
ts.h:189
#2  0x0000000000400d91 in std::vector<int, std::allocator<int> >::_M_insert_aux (
    this=0x7ffcf1e34a80, __position=..., __x=@0x0: <error reading variable>)
    at /cvmfs/lhcb.cern.ch/lib/lcg/releases/gcc/4.9.3/x86_64-slc6/include/c++/4.9.3/bits/vector.tc
c:361
#3  0x0000000000400b40 in std::vector<int, std::allocator<int> >::push_back (
    this=0x7ffcf1e34a80, __x=@0x0: <error reading variable>)
    at /cvmfs/lhcb.cern.ch/lib/lcg/releases/gcc/4.9.3/x86_64-slc6/include/c++/4.9.3/bits/stl_vecto
r.h:925
#4  0x00000000004009c3 in main (argc=1, argv=0x7ffcf1e34bb8) at test.C:9
(gdb) f 4
#4  0x00000000004009c3 in main (argc=1, argv=0x7ffcf1e34bb8) at test.C:9
9         b.push_back(a[0]);
```

小结

- C++
- 类，多态
- 友元
- g++编译C++程序
- 用Makefile编译C++程序

作业

- 对刚才的类添加新的功能
  计算它可以容纳的最大的圆的面积，输出。
- 理解课上的类的继承的重要的概念，编译链接通过，并成功运行课上20，22，23，35-37页的例子。
- 为什么我们需要多态？列举几种应用场景。

ROOT安装

1. 下载root的源代码http://root.cern.ch/

找到Download，还有Documentation->Building root

2. 如上提示的方法解压，安装

3. 你的系统可能会缺少一些支持软件

http://root.cern.ch/drupal/content/build-prerequisites

4. 在Ubuntu环境中可以利用apt-get命令安装缺少的内容。一般缺少的都是开发包，例如libglew1.5-dev，即头文件，库文件，链接库等。

演示ROOT

- > root

我们看到类在这里有着充分的应用！

演示1：

- root [0] TH1F h1("h1","myhistorgram", 100, -5., 5.)
- root [1] h1.FillRandom("gaus",5000)
- root [2] h1.Draw()

演示2：

- root [3] TF1 poi("poi","5**int(x)*exp(-5)/ TMath::Factorial(int(x))",0,20)
- root [4] poi.Draw()

演示3：

- 其他

一些基本概念的测试

下面这些表达式是什么意思？

Statement:
`int A::b(int c) { }`
`a->b`
`class A: public B {};`

模板 Template

- In C++, two different functions can have the same name if their parameters are different
  - either because they have a different number of parameters
  - or because any of their parameters are of a different type
- They are called overloaded functions
- Template of functions is convenient

```
// function template
#include <iostream>
using namespace std;
template <class T>
T sum (T a, T b){
  T result;
  result = a + b;
  return result;
}
int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
  cout << k << '\n';
  cout << h << '\n';
  return 0;
}
```

C++ 的历史

- C 语言大约在 1970 年诞生于Bell Labs
  UNIX 的一部分是用C语言写的
- Bjarne Stroustrup (本贾尼·斯特劳斯特卢普) 在80年代基于C语言开发了 C++ “C with classes”, 也就是说允许面向对象编程的用户自定义的数据类型”

- C++ 是作为C的增量改进版而开发的
- C 可以看做 C++ 的子集，所以，大部分C程序可以直接用C++编译器编译
- 从开发以来有4个重要 C++ 标准： C++98 (1998), C++03 (2003) and C++11 (2011) and C++14 (2014).