粒子物理与核物理实验中的数据分析

王喆 杨振伟

清华大学

第二讲： Linux，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
- 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）
- 下半学期稍后我们要安装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

文件解压缩

- tar cvfz Lec2.tgz Lec2 压缩
- tar xvfz Lec2.tgz 解压

本节课的例子在Lec2 Attachment
下载后需要解压

安装新的工具包

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

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 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).

设计带给我们变革

- 例子：
  - 小学时我们要解应用题，思路要清晰，好多加减乘除相当复杂
    - 难倒成年人
  - 初中时我们学会了列方程，这样想法就简单多了
    - 备注：问题是我们要会解
  - 后来我们又学了微积分，思路就更开阔了
    - 备注：我们还是要能够求解

程序语言设计

- 计算机编程语言：
  - 汇编语言：生涩难懂，想要开发高级或复杂的功能等于不可能
- 有了C语言，终于轻松的写a+b了
  - C语言编译器会帮我们把a+b写成机器语言。而且编译器可以深度优化，是实现效率大幅提高
  - 擅长写函数

汇编：

```
RAND  PROC
      PUSH AX
      STI
      MOV AH,0
```

C语言：

```
c=a+b;
```

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
    吠叫:
    毛皮颜色
    品种

C++速成，例子**1.** HelloWorld

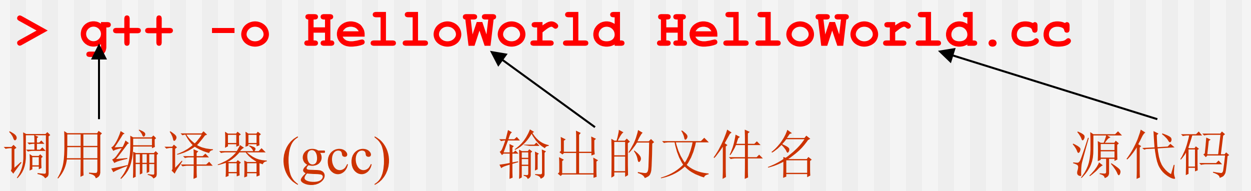
C++速成：HelloWorld

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 (gcc)

输出的文件名

源代码

最后执行程序

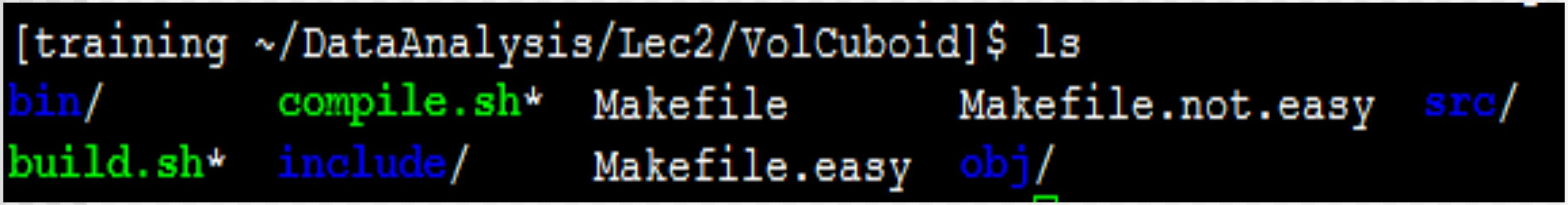
```
>./HelloWorld
Hello World!
```

← 用户键入(注意：>为系统提示符)
← 计算机显示结果

C++速成，例子2. VolCuboid

一个较好的C++程序组织结构

Lec2/VolCuboid/



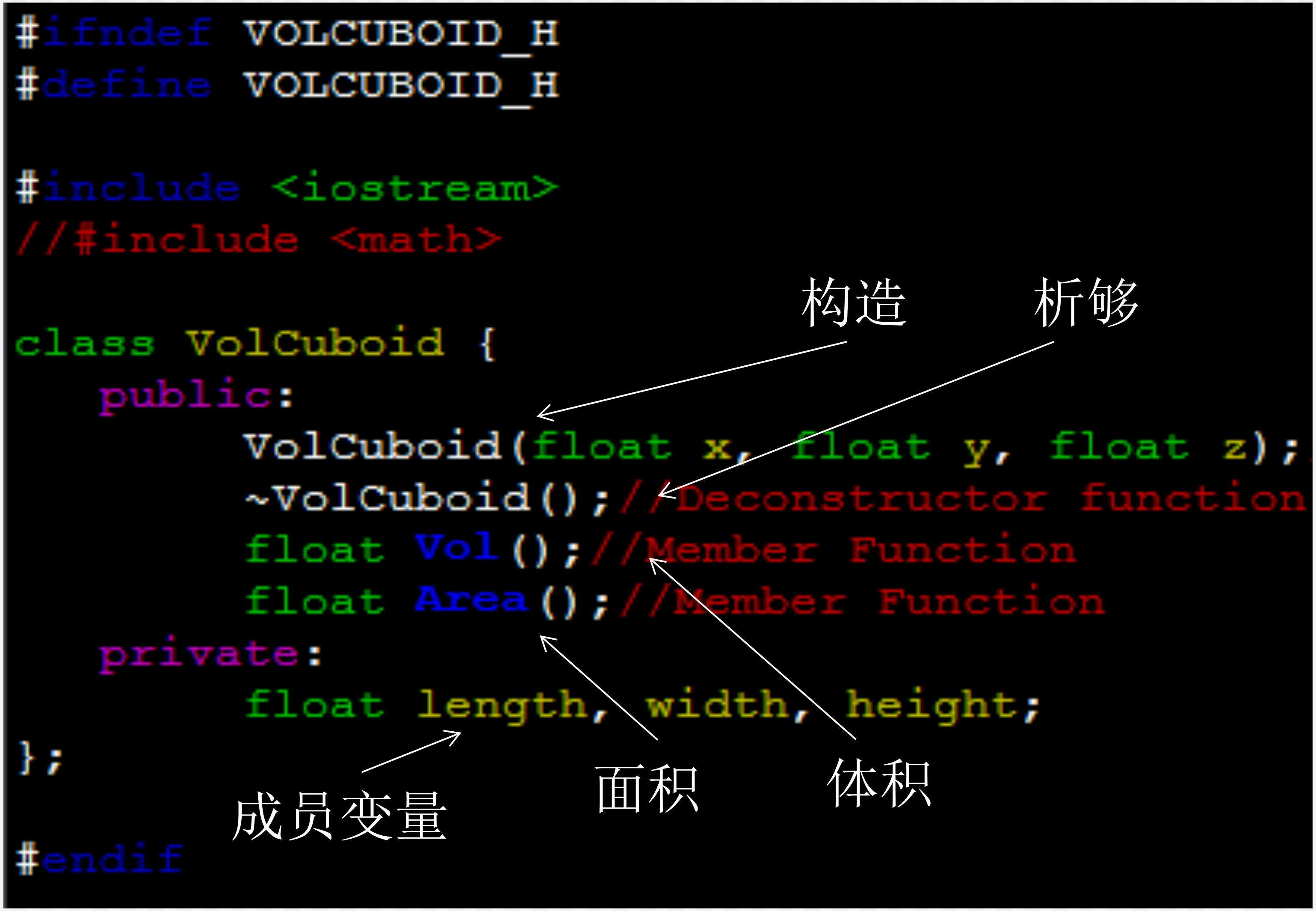
```
[training ~/DataAnalysis/Lec2/VolCuboid]$ ls
bin/        compile.sh*  Makefile        Makefile.not.easy  src/
build.sh*   include/     Makefile.easy   obj/
```

Linux下标准的C++程序项目一般把源文件、头文件、目标文件及可执行文件放在不同目录，便于维护管理。

比如某个程序项目，为该项目建立工作目录(如VolCuboid)，工作目录中一般会有bin, include, obj, src等子目录，分别存放可执行文件、头文件、目标文件和源文件。工作目录中还会有编译文件以及其它辅助文件(如输入参数文件)。

C++类（定义，头文件）

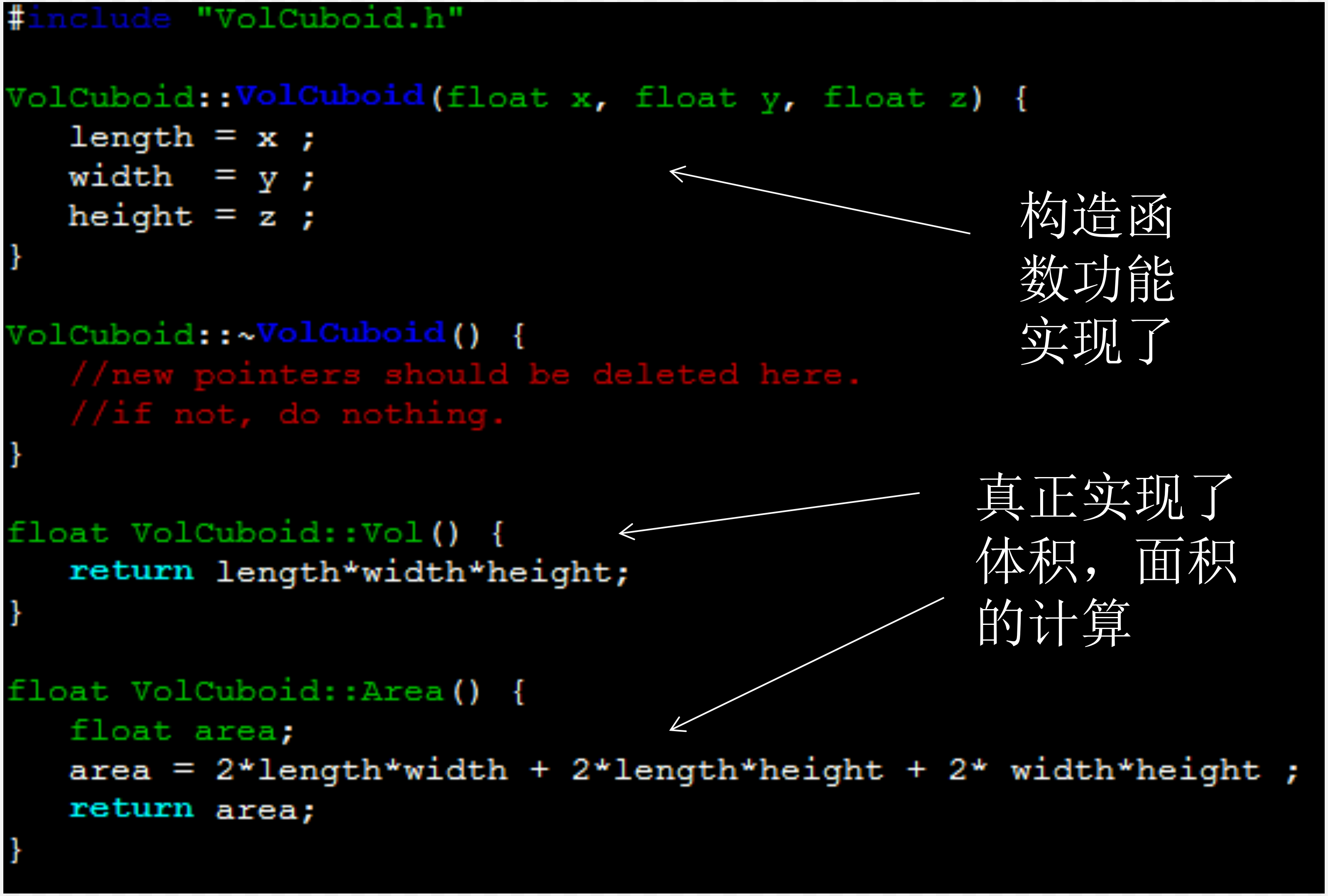
Lec2/VolCuboid/include/VolCuboid.h



```
#ifndef VOLCUBOID_H
#define VOLCUBOID_H

#include <iostream>
//#include <math>

class VolCuboid {
   public:
         VolCuboid(float x, float y, float z);
         ~VolCuboid();//Deconstructor function
         float Vol();//Member Function
         float Area();//Member Function
   private:
         float length, width, height;
};

#endif
```

C++类（实现，执行，源文件）

Lec2/VolCuboid/src/VolCuboid.cc



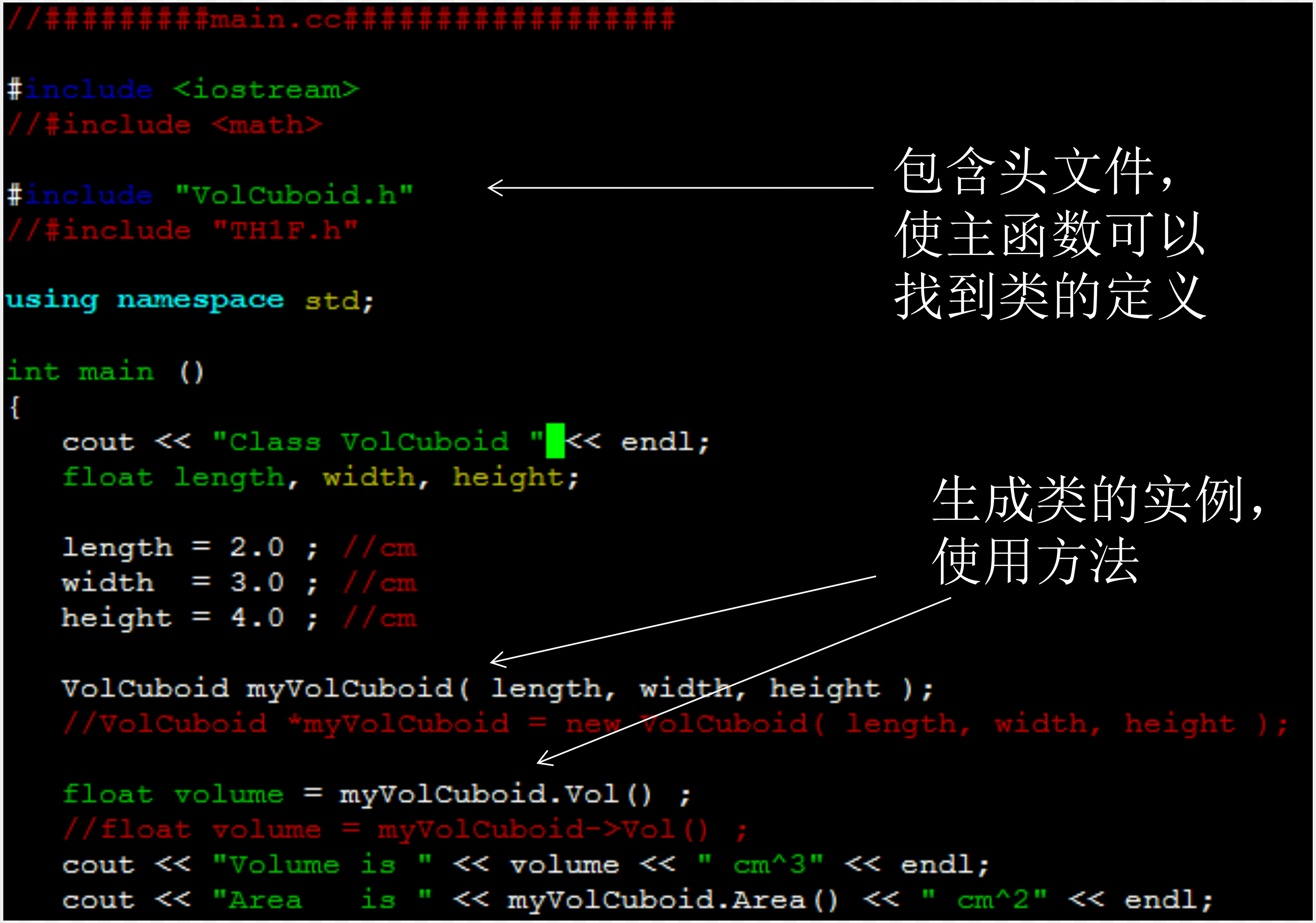
```
#include "VolCuboid.h"

VolCuboid::VolCuboid(float x, float y, float z) {
   length = x ;
   width  = y ;
   height = z ;
}

VolCuboid::~VolCuboid() {
   //new pointers should be deleted here.
   //if not, do nothing.
}

float VolCuboid::Vol() {
   return length*width*height;
}

float VolCuboid::Area() {
   float area;
   area = 2*length*width + 2*length*height + 2* width*height ;
   return area;
}
```

构造函数功能实现了

真正实现了体积，面积的计算

C++类（使用，主函数）

Lec2/VolCuboid/src/main.cc



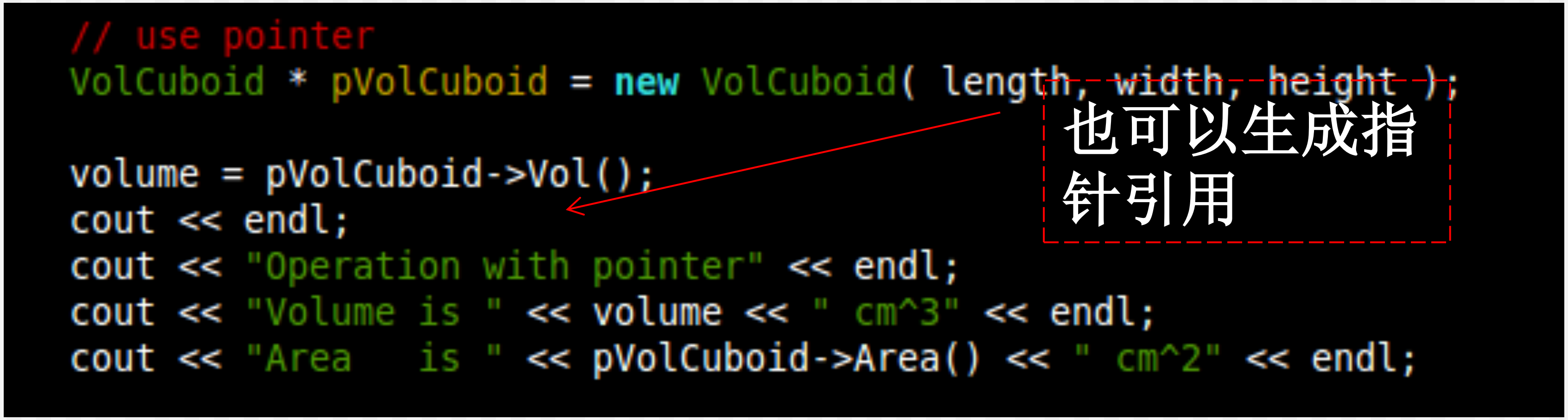
```
//#########main.cc##################

#include <iostream>
//#include <math>

#include "VolCuboid.h"
//#include "TH1F.h"

using namespace std;

int main ()
{
   cout << "Class VolCuboid " << endl;
   float length, width, height;

   length = 2.0 ; //cm
   width  = 3.0 ; //cm
   height = 4.0 ; //cm

   VolCuboid myVolCuboid( length, width, height );
   //VolCuboid *myVolCuboid = new VolCuboid( length, width, height );

   float volume = myVolCuboid.Vol() ;
   //float volume = myVolCuboid->Vol() ;
   cout << "Volume is " << volume << " cm^3" << endl;
   cout << "Area   is " << myVolCuboid.Area() << " cm^2" << endl;
```

包含头文件，
使主函数可以
找到类的定义

生成类的实例，
使用方法

C++类（指针）

Lec2/VolCuboid/src/main.cc



```
// use pointer
VolCuboid * pVolCuboid = new VolCuboid( length, width, height );

volume = pVolCuboid->Vol();
cout << endl;
cout << "Operation with pointer" << endl;
cout << "Volume is " << volume << " cm^3" << endl;
cout << "Area   is " << pVolCuboid->Area() << " cm^2" << endl;
```

也可以生成指针引用

指针里面放的是一个数据在计算机内存中的地址。例如大家知道我的办公室地址804就可以找到我了。

计算机的内存分成多个段（segment），数据存储的，程序代码的，每个程序还有自己的段内存空间，不能跑到别人那去，去了就叫段错误，segment fault，根本原因是某个指针被赋予了一个非法的值。

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

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)

Makefile简介

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

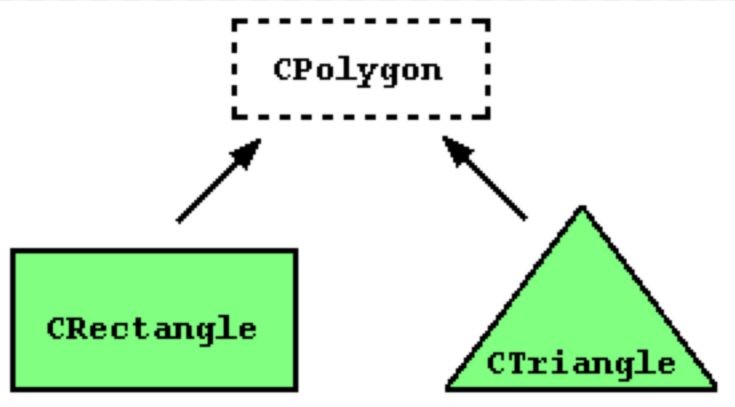
可执行文件

C++后缀,如所有.cc改为.o

类的重要概念和应用

继承 Inheritance

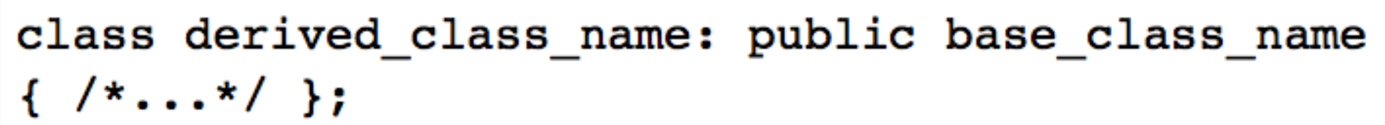
- ***Inheritance*** 是面向对象编程最重要的特性之一
- 类可以被扩展，即可以创建一个类使其保持“基类”的所有属性 ➔ ***inheritance***
- 关于 ***“基类” base class*** 和 ***“派生类” derived class***: 派生类继承基类的成员，以此为基础还可以添加新的成员。



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:            //derived class
    int area ()
      { return width * height; }
};                   //derived class
class Triangle: public Polygon {
  public:   int area ()
      { return width * height / 2; }
  };
```

```
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**

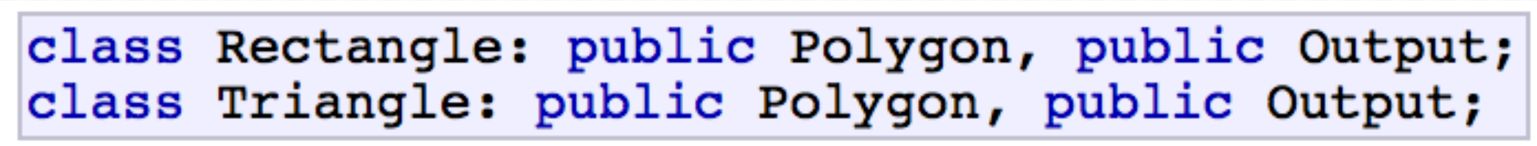
继承的要点

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

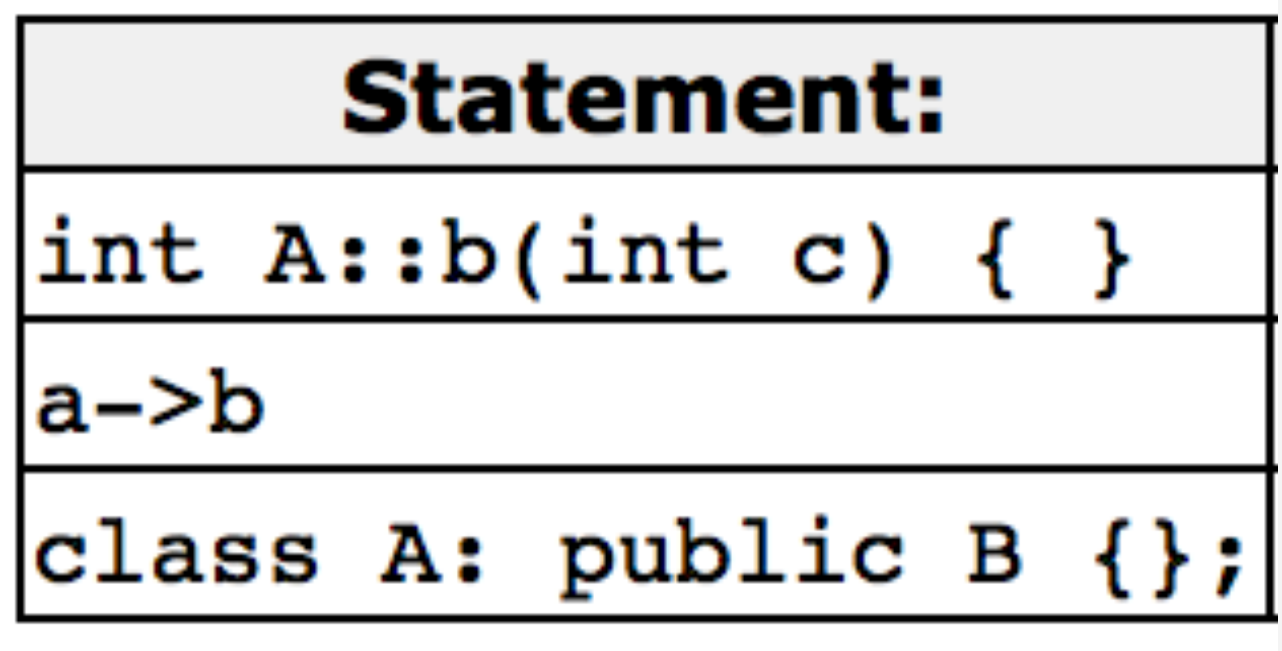
- 派生类可以继承自多个基类，不同基类之间用逗号分隔。



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

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 {};`

友元 Friendships

- In principle, ***private*** and ***protected*** members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to "friends".
- Friends are functions or classes declared with the ***friend*** keyword.
- A non-member function can access the private and protected members of a class if it is declared a friend of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend

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;
}
```

```
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*)

友元函数示例

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

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) {}
};
```

```
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

Polymorphism (多态性)

指向基类的指针

- One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class.
- Polymorphism is the art of taking advantage of this simple but powerful and versatile feature.

```
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 *inherited* members!**

ppoly1->area(); ✗

如何访问函数area()

- If **area()** is defined in base class **Polygon**…
- But the implementations of **area()** in **Rectangle** and **Triangle** 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;
}
```

question: what are the output?

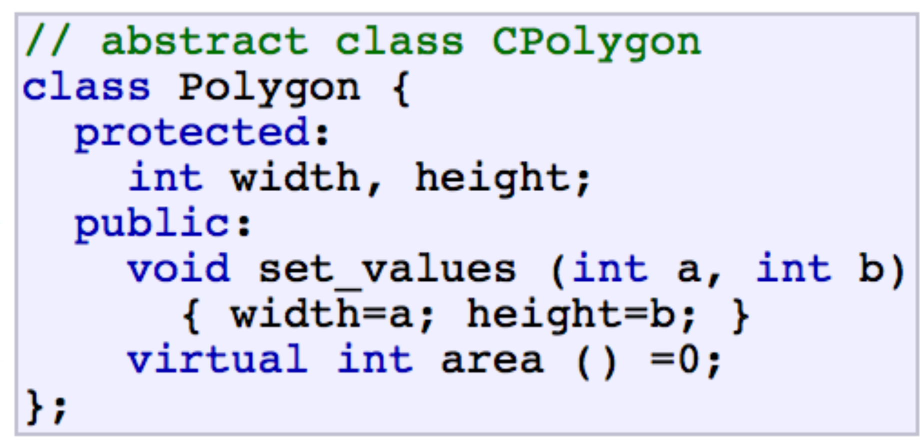
More words about virtual members

- Non-virtual members can also be redefined in derived classes
- But non-virtual members of derived classes cannot be accessed through a reference of the base class

- A class that declares or inherits a virtual function is called a **polymorphic class**

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; ☺

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;
}
```

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

小结

- C++
- 类
- g++编译C++程序
- 用Makefile编译C++程序
- ROOT功能初试

作业

- 对刚才的类添加新的功能
  计算它可以容纳的最大的球的体积，并输出。
- 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26，27，29，31，33页的例子。

粒子物理与核物理实验中的数据分析

王 喆　　杨振伟

清华大学

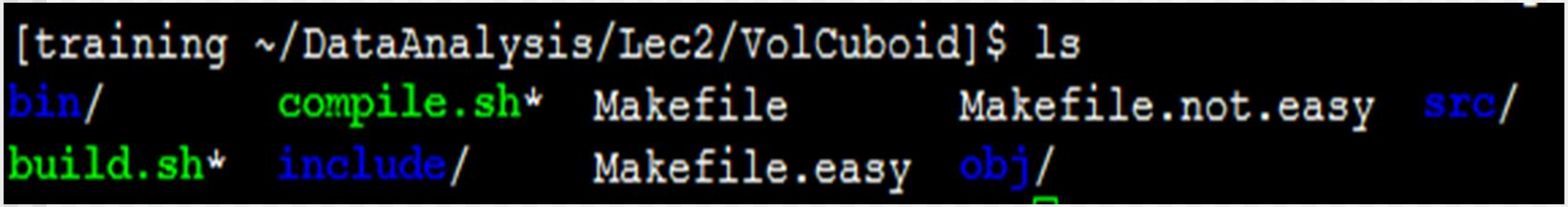
第4讲：C++

Outline

- 重新讨论类
- operator
- make & Makefile
- gdb
- STL

例子：Lec2/VolCuboid/

文件及目录结构：



```
[training ~/DataAnalysis/Lec2/VolCuboid]$ ls
bin/       compile.sh*  Makefile       Makefile.not.easy  src/
build.sh*  include/     Makefile.easy  obj/
```

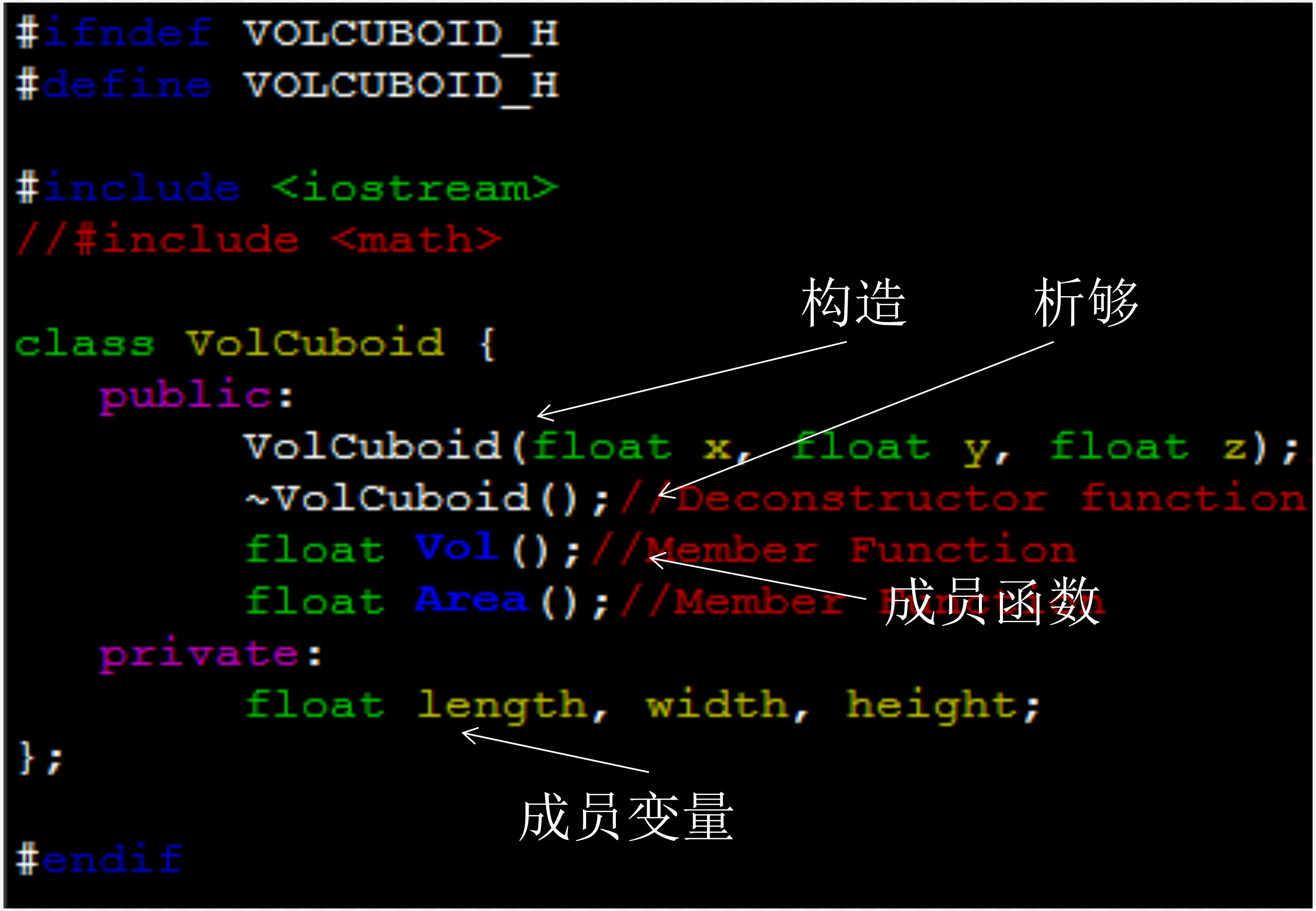
Linux下标准的C++程序项目一般把

1. 源文件
2. 头文件
3. 目标文件
4. 可执行文件

放在不同目录，便于维护管理。

C++类（定义，头文件）

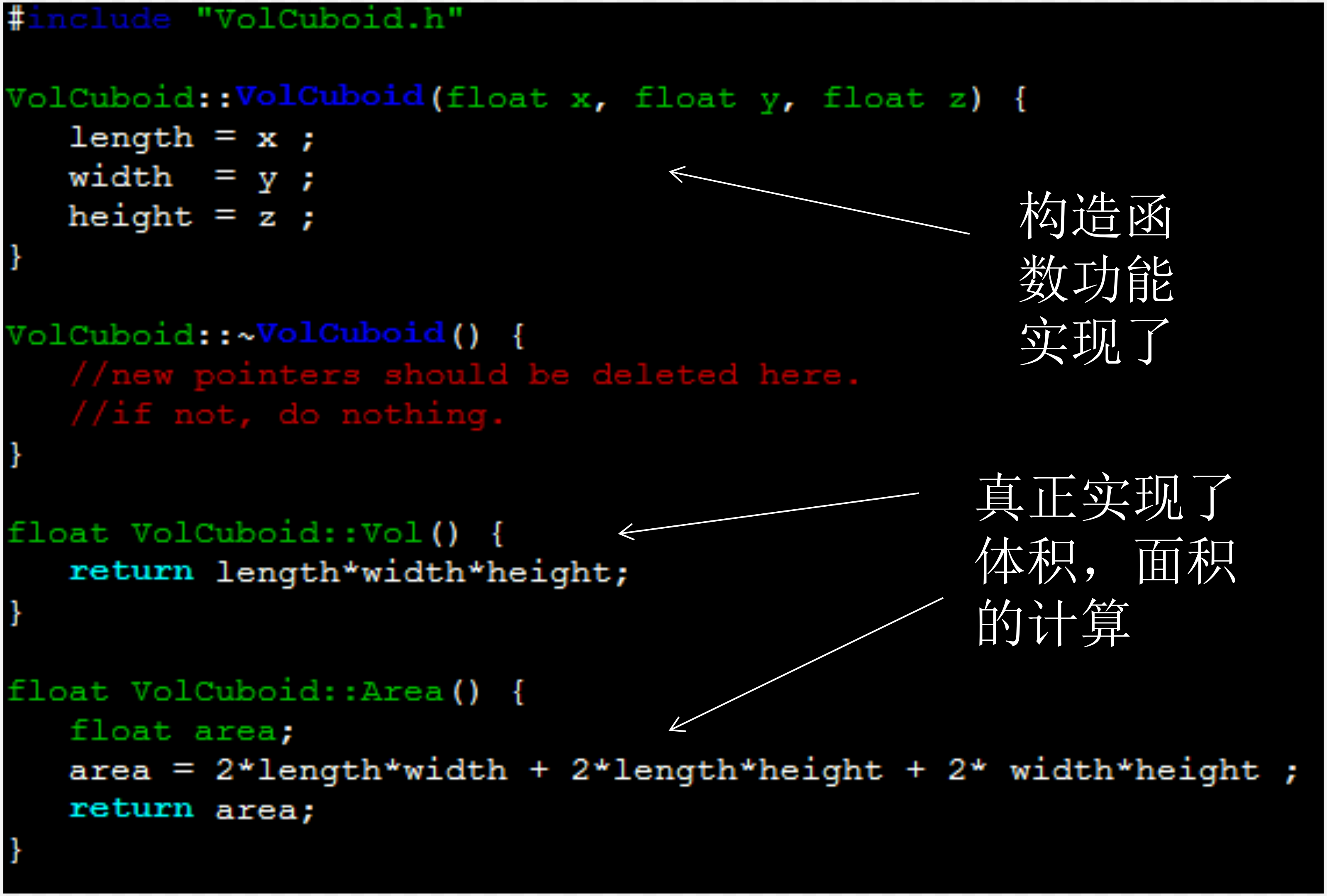
Lec2/VolCuboid/include/VolCuboid.h



```
#ifndef VOLCUBOID_H
#define VOLCUBOID_H

#include <iostream>
//#include <math>

class VolCuboid {
   public:
        VolCuboid(float x, float y, float z);
        ~VolCuboid();//Deconstructor function
        float Vol();//Member Function
        float Area();//Member Function
   private:
        float length, width, height;
};

#endif
```

C++类（实现，执行，源文件）

Lec2/VolCuboid/src/VolCuboid.cc



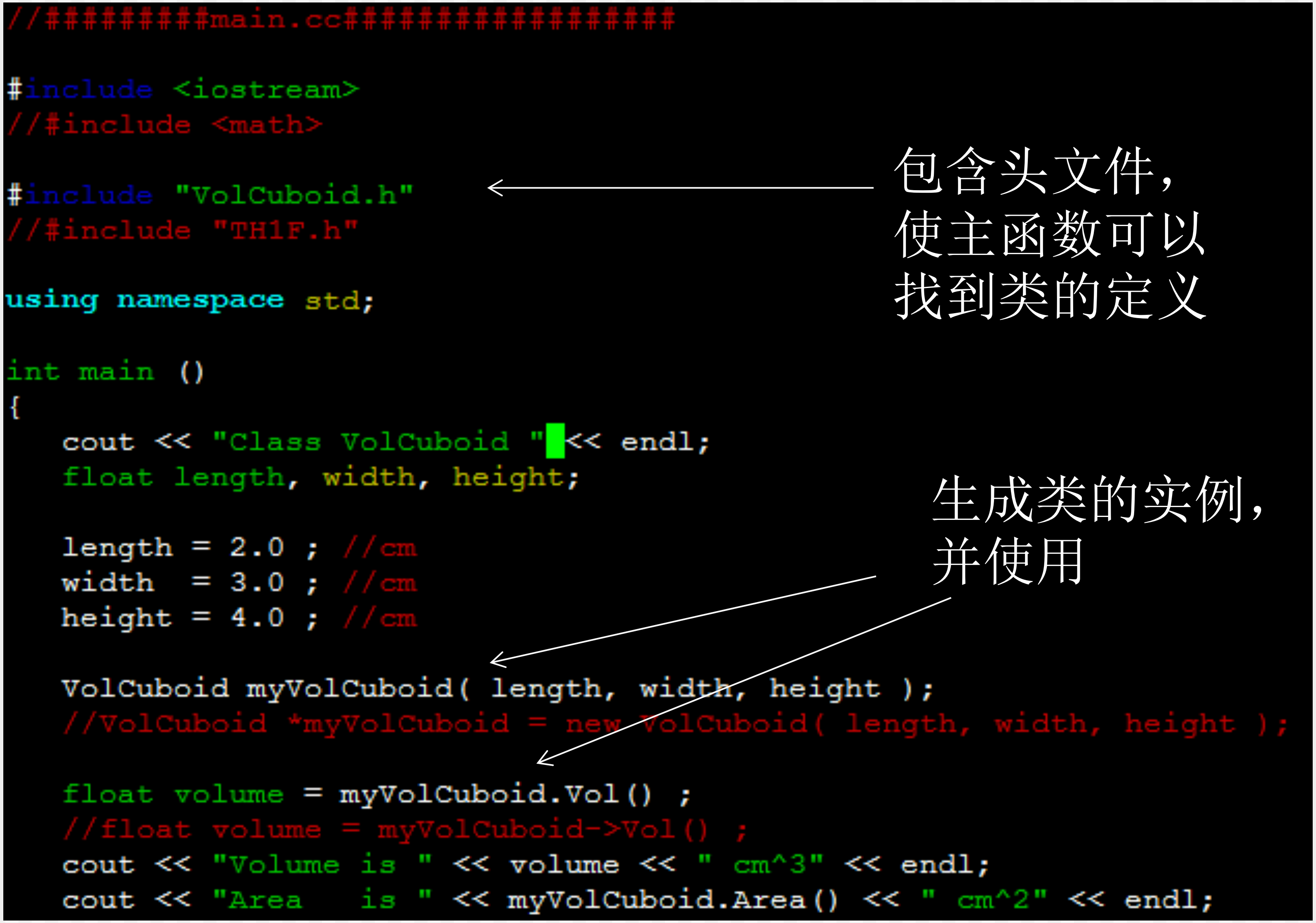
```
#include "VolCuboid.h"

VolCuboid::VolCuboid(float x, float y, float z) {
   length = x ;
   width  = y ;
   height = z ;
}

VolCuboid::~VolCuboid() {
   //new pointers should be deleted here.
   //if not, do nothing.
}

float VolCuboid::Vol() {
   return length*width*height;
}

float VolCuboid::Area() {
   float area;
   area = 2*length*width + 2*length*height + 2* width*height ;
   return area;
}
```

构造函数功能实现了

真正实现了体积，面积的计算

C++类（使用，主函数）

Lec2/VolCuboid/src/main.cc



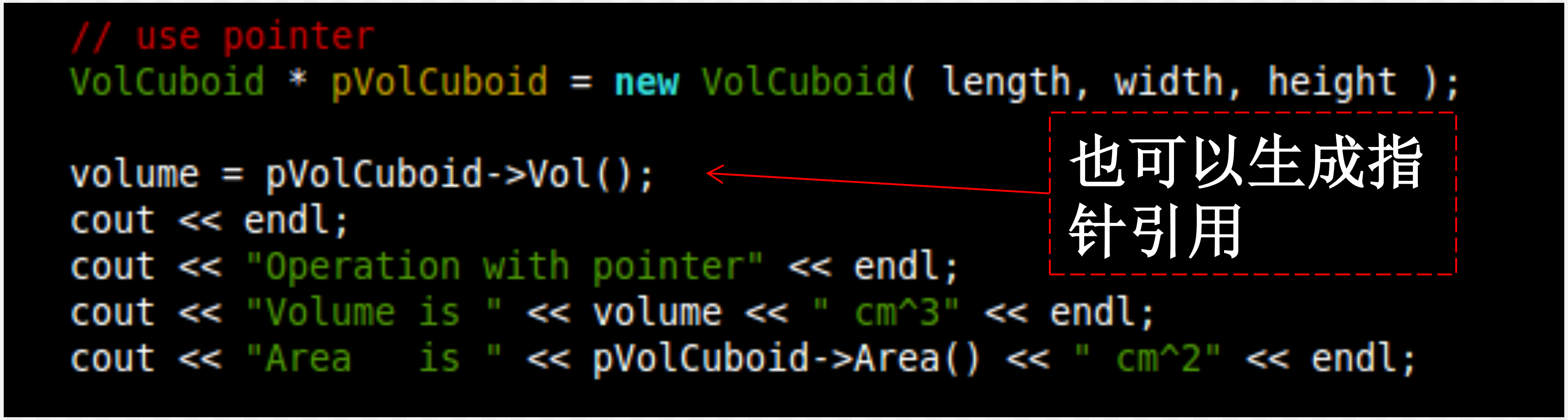
```
//#########main.cc##################

#include <iostream>
//#include <math>

#include "VolCuboid.h"
//#include "TH1F.h"

using namespace std;

int main ()
{
   cout << "Class VolCuboid " << endl;
   float length, width, height;

   length = 2.0 ; //cm
   width  = 3.0 ; //cm
   height = 4.0 ; //cm

   VolCuboid myVolCuboid( length, width, height );
   //VolCuboid *myVolCuboid = new VolCuboid( length, width, height );

   float volume = myVolCuboid.Vol() ;
   //float volume = myVolCuboid->Vol() ;
   cout << "Volume is " << volume << " cm^3" << endl;
   cout << "Area   is " << myVolCuboid.Area() << " cm^2" << endl;
```

包含头文件，
使主函数可以
找到类的定义

生成类的实例，
并使用

C++类（指针）

Lec2/VolCuboid/src/main.cc



```
// use pointer
VolCuboid * pVolCuboid = new VolCuboid( length, width, height );

volume = pVolCuboid->Vol();
cout << endl;
cout << "Operation with pointer" << endl;
cout << "Volume is " << volume << " cm^3" << endl;
cout << "Area   is " << pVolCuboid->Area() << " cm^2" << endl;
```

也可以生成指针引用

指针里面放的是一个数据在计算机内存中的地址。例如大家知道我的办公室地址804就可以找到我了。

为什么要使用指针：一个object在内存里面的复制操作太消耗资源了。传递一个单值的指针则很方便

常量，常指针

```
const int pi = 3.1415926;
const VolCuboid & ref = aVolCuboid;
```

变量pi, ref不得更改

```
VolCuboid::Area() const
```

Area函数不得修改类VolCuboid的成员变量

```
const VolCuboid* pVol;
VolCuboid* const pVol;
```

指针所指的内容是常量
指针本身是常量

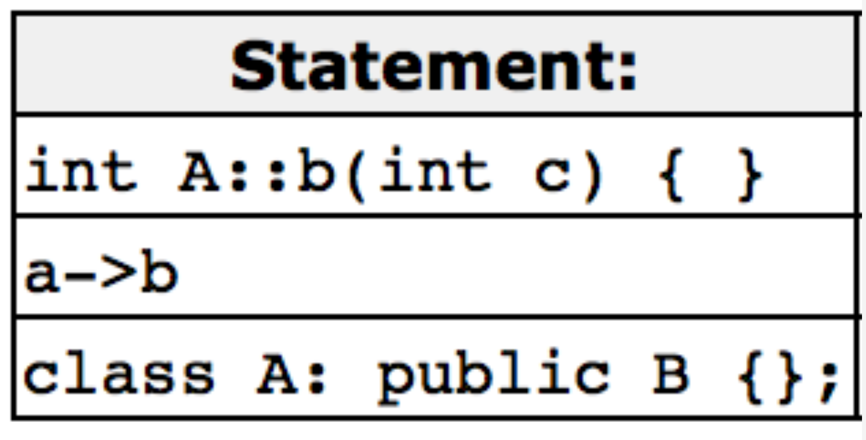
```
const VolCuboid* const pVol;
```

指针的值和指针所指的内容全部是常量

这样做的目的：设计意图明确，可读性更强，减少出错概率

一些基本概念的测试

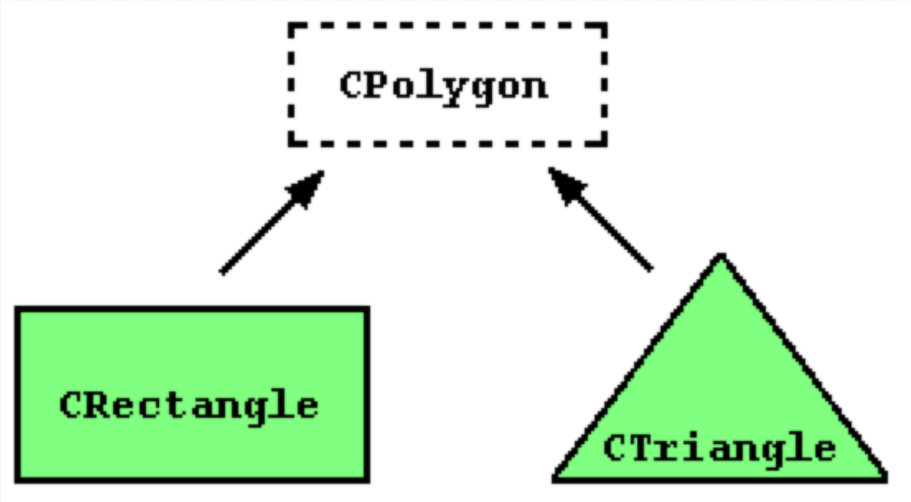
下面这些表达式是什么意思？
名字空间?



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

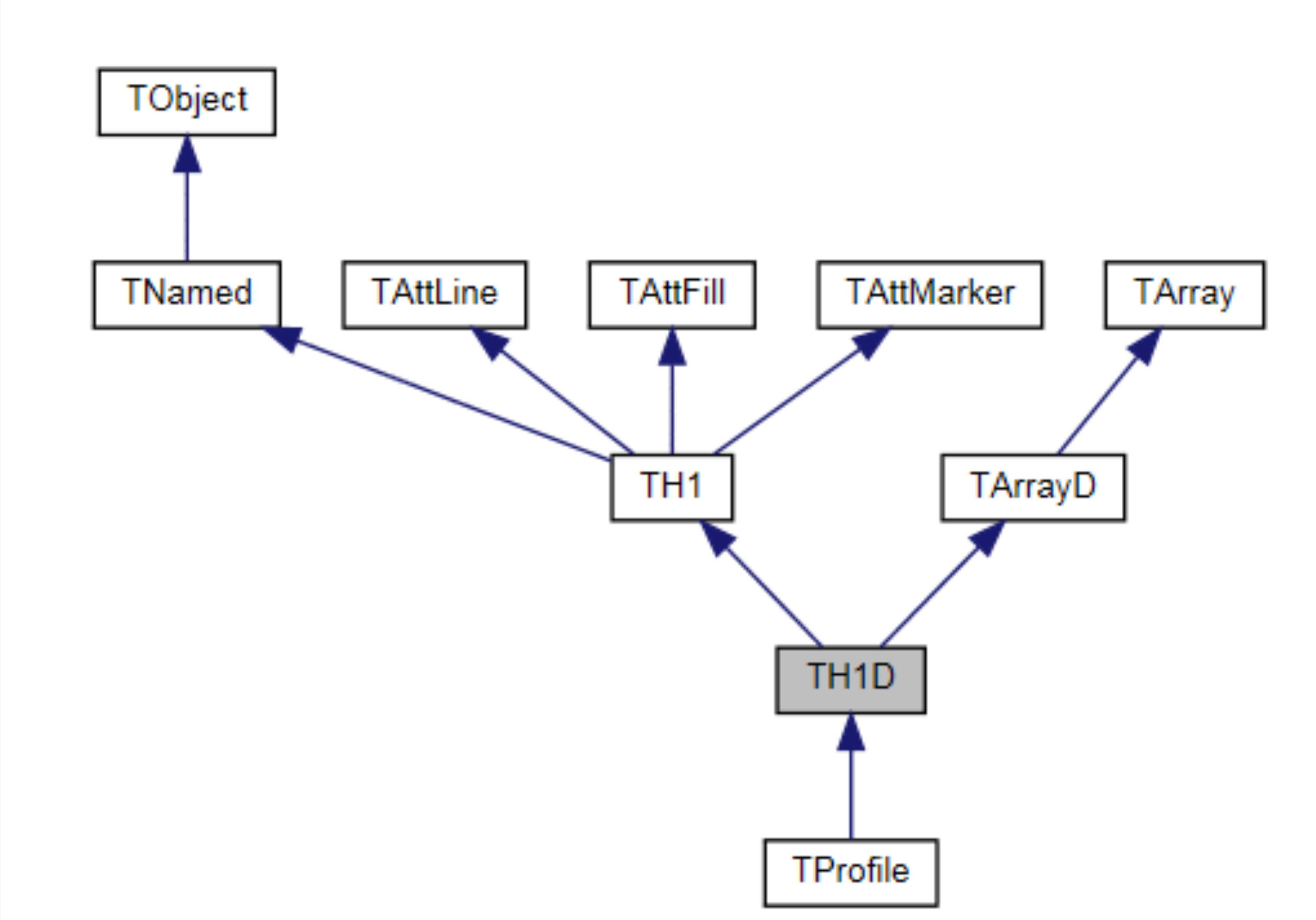
类，继承，虚函数，多态

类：让我们更方便的描述一些物体，对象，问题
类继承：方便的分层次，模块化，设计意图明确



2017/3/9

一个root的关于继承的例子



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;
}
```

question: what are the output?

虚基类，虚函数：Geant4的基本实现原理

```
模拟函数（一个形体，一个粒子，位移）
{
        一个形体的辐射长度?
        一个粒子的电荷?
        一个粒子在一个形体内的位移?
}
```

程序基础设计者可以完成如上的框架型代码，其中**一个形体，一个粒子，全部是基类指针**

```
使用该函数：
模拟函数（正方形铁块，电子，位移）
```

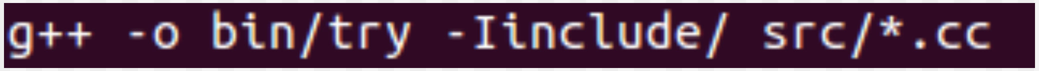
实际使用时：一个形体，一个粒子被赋予派生类指针

虚基类和虚函数很好的完成这一设想

编译，链接

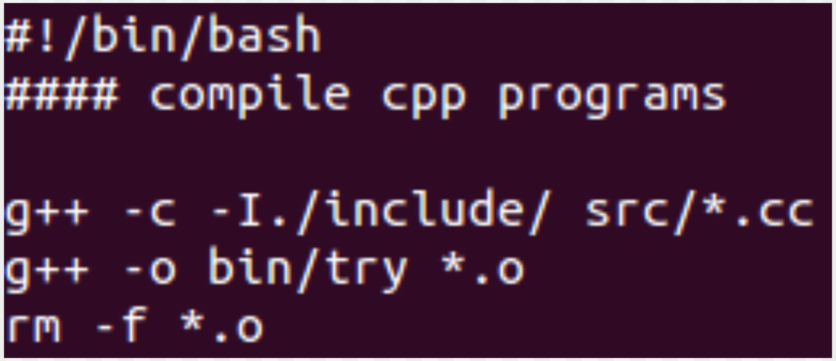
在Lec2/VolCuboid/中展示了多种编译链接的方式

build.sh:



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

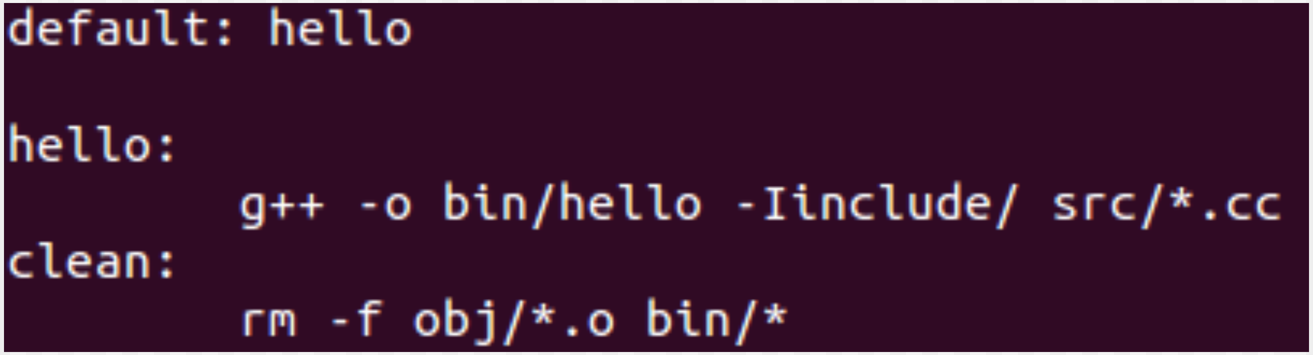
compile.sh:



```
#!/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)*
```

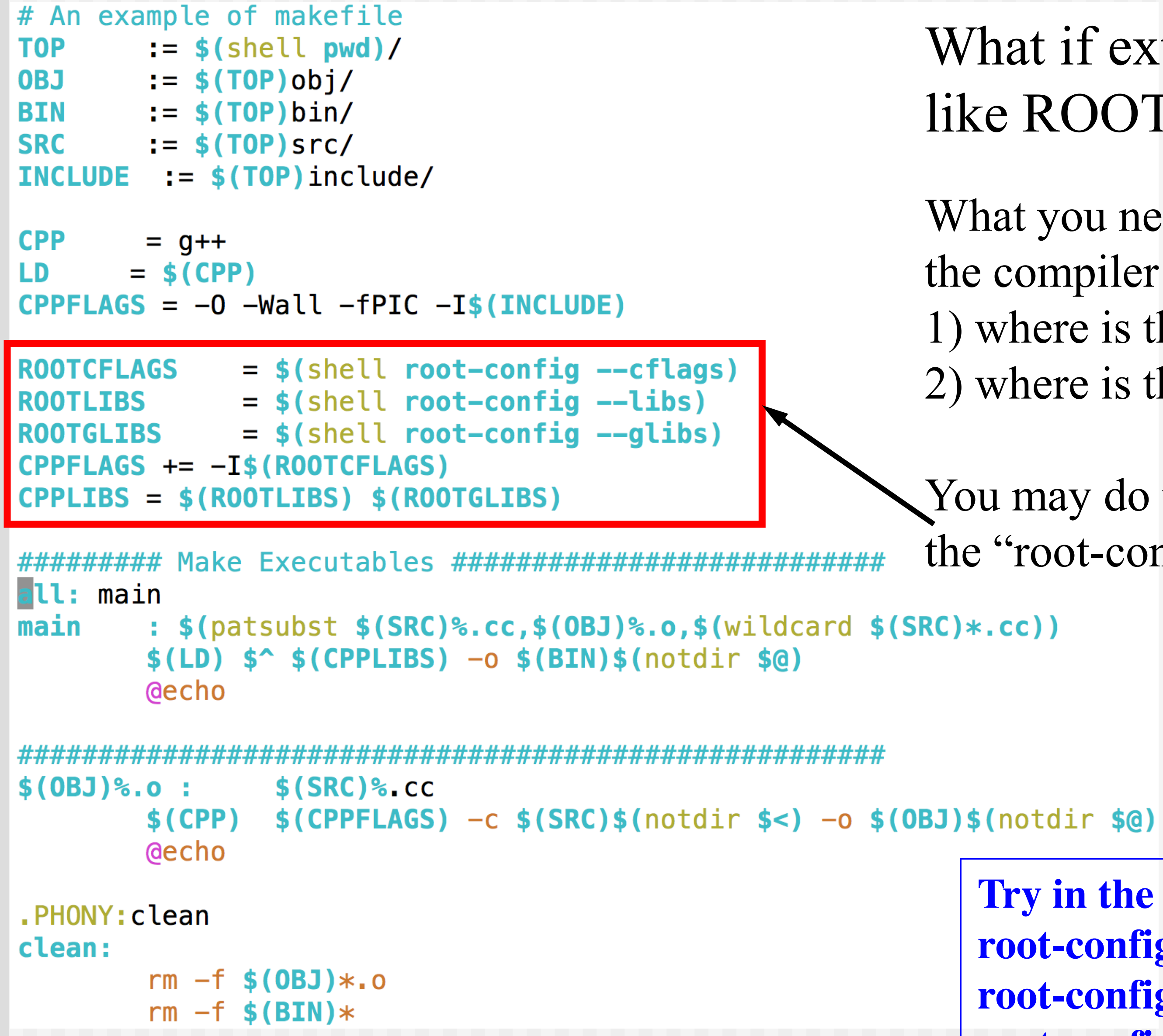
头文件或者库文件目录

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

Operators in class

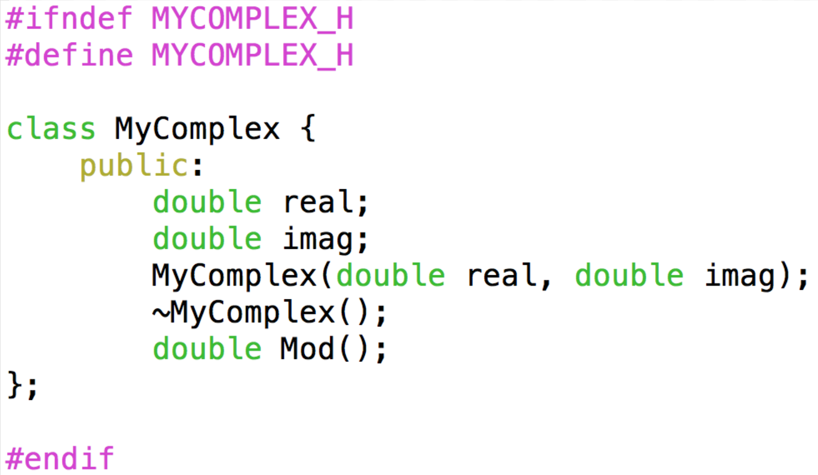
Operator overload in class is useful.
E.g., you have a class “MyComplex” for complex numbers, and use it to instantiate two complex numbers:

MyComplex c1(1.0,2.0), c2(2.0,4.0);

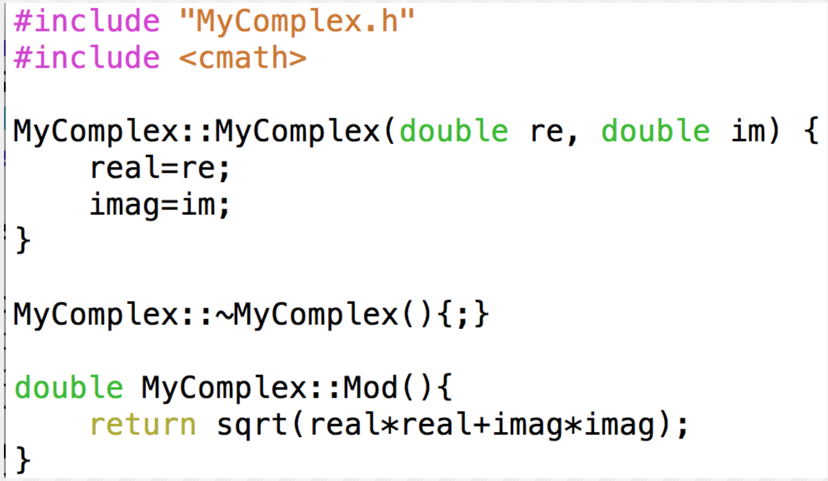
You may want to assign the plus of them to “sum”:

MyComplex sum=c1+c2;

It would be convenient if you overload “+”



```
#ifndef MYCOMPLEX_H
#define MYCOMPLEX_H

class MyComplex {
    public:
        double real;
        double imag;
        MyComplex(double real, double imag);
        ~MyComplex();
        double Mod();
};

#endif
```



```
#include "MyComplex.h"
#include <cmath>

MyComplex::MyComplex(double re, double im) {
    real=re;
    imag=im;
}

MyComplex::~MyComplex(){;}

double MyComplex::Mod(){
    return sqrt(real*real+imag*imag);
}
```

Operators in class

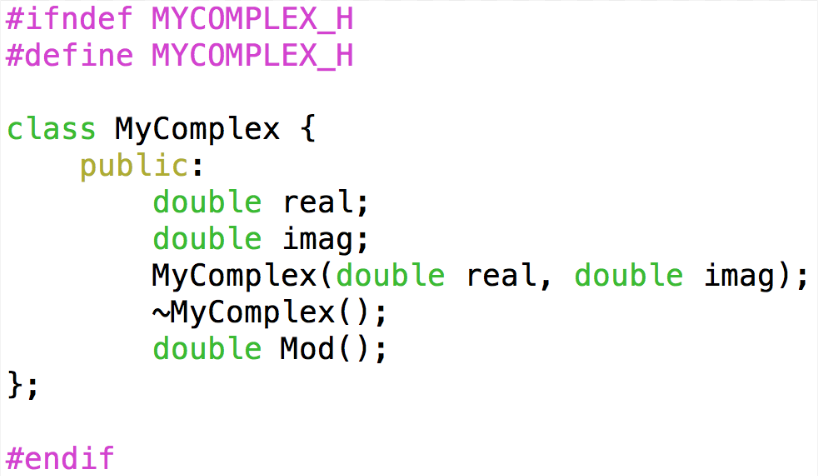
Operator overload in class is useful.
E.g., you have a class “MyComplex” for complex numbers, and use it to instantiate two complex numbers:

  MyComplex c1(1.0,2.0), c2(2.0,4.0);

You may want to assign the plus of them to “sum”:

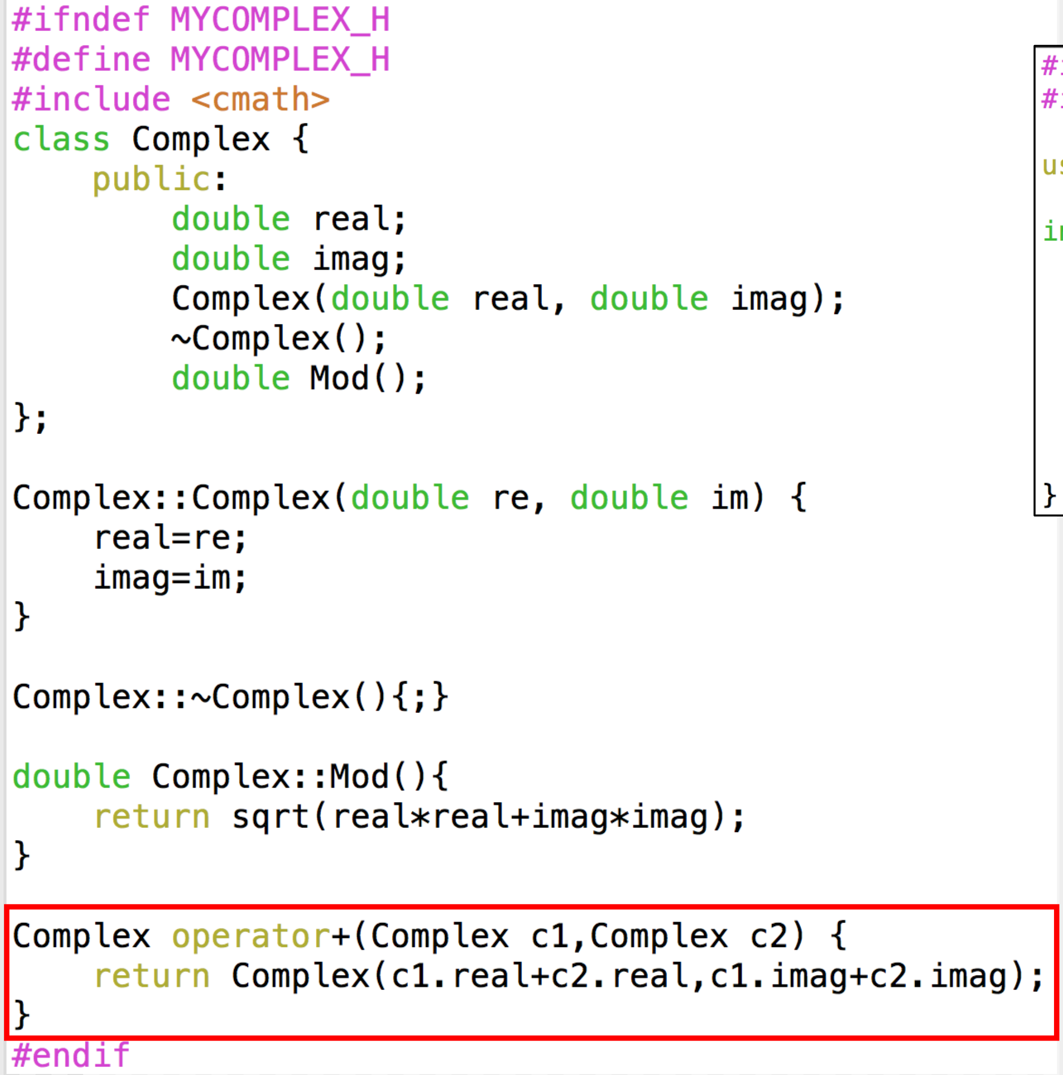
  MyComplex sum=c1+c2;

It would be convenient if you overload “+”

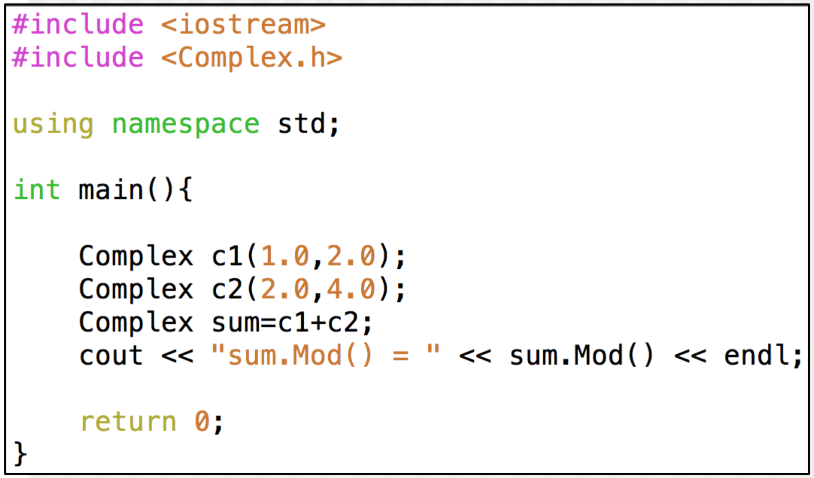


```
#ifndef MYCOMPLEX_H
#define MYCOMPLEX_H

class MyComplex {
    public:
        double real;
        double imag;
        MyComplex(double real, double imag);
        ~MyComplex();
        double Mod();
};

#endif
```

Overload of operator +



```
#ifndef MYCOMPLEX_H
#define MYCOMPLEX_H
#include <cmath>
class Complex {
    public:
        double real;
        double imag;
        Complex(double real, double imag);
        ~Complex();
        double Mod();
};

Complex::Complex(double re, double im) {
    real=re;
    imag=im;
}

Complex::~Complex(){;}

double Complex::Mod(){
    return sqrt(real*real+imag*imag);
}

Complex operator+(Complex c1,Complex c2) {
    return Complex(c1.real+c2.real,c1.imag+c2.imag);
}
#endif
```



```
#include <iostream>
#include <Complex.h>

using namespace std;

int main(){

    Complex c1(1.0,2.0);
    Complex c2(2.0,4.0);
    Complex sum=c1+c2;
    cout << "sum.Mod() = " << sum.Mod() << endl;

    return 0;
}
```

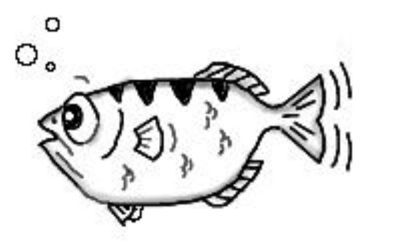
注意，注意：

C++已经有复数类了

#include <complex>

GDB: the GNU project debugger

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



GDB allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

What GDB can do:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

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
```

STL (Standard Template Library)

- A software library for the C++ programming language that influenced many parts of the C++ Standard Library
- It provides four components
  - **algorithms**
    - ✓ **Non-modifying sequence operations**
    - ✓ **Modifying sequence operations**
    - ✓ **Sorting**
    - ✓ **Merge**
    - ✓ **……**
  - **containers**
    - ✓ **array, deque, forward_list, list, map, queue, set stack, unordered_map, unordered_set, vector**
  - **functional**
  - **iterators**

http://www.cplusplus.com/reference/

STL vector

为了可以使用vector，必须在你的头文件中包含下面的代码：

```
#include <vector>
```

vector属于std命名域的，因此需要通过命名限定，如下完成你的代码：

```
using std::vector;
vector<int> c;
```

或者连在一起，使用全名：

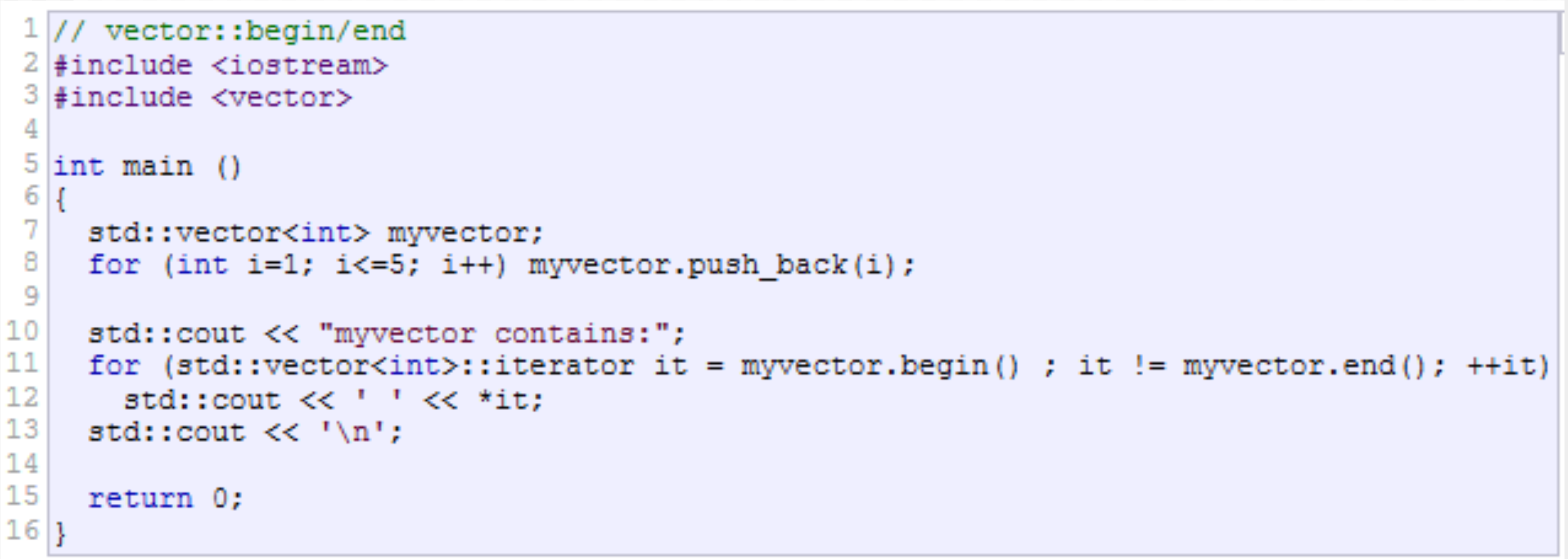
```
std::vector<int> c;
```

c.max_size()
返回容器中数据的数量。
c.pop_back()
删除最后一个数据。
c.push_back(elem)
在尾部加入一个数据。

vector和数组效率是差不多的，vector是可变长的，其他的一些操作更方便。

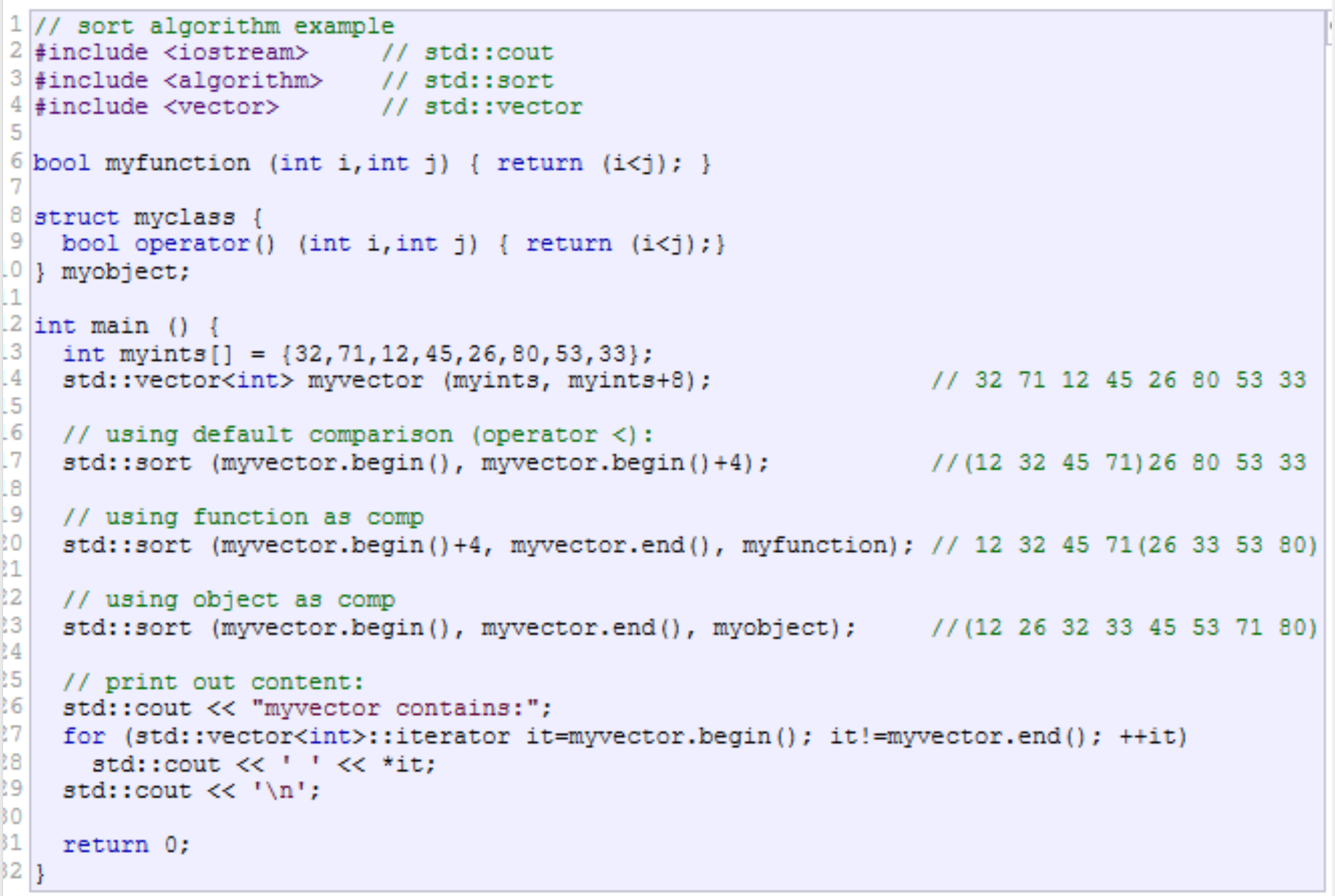
vector的循环，迭代子

std::vector<*int*>::iterator
或者
std::vector<*int*>::const_iterator



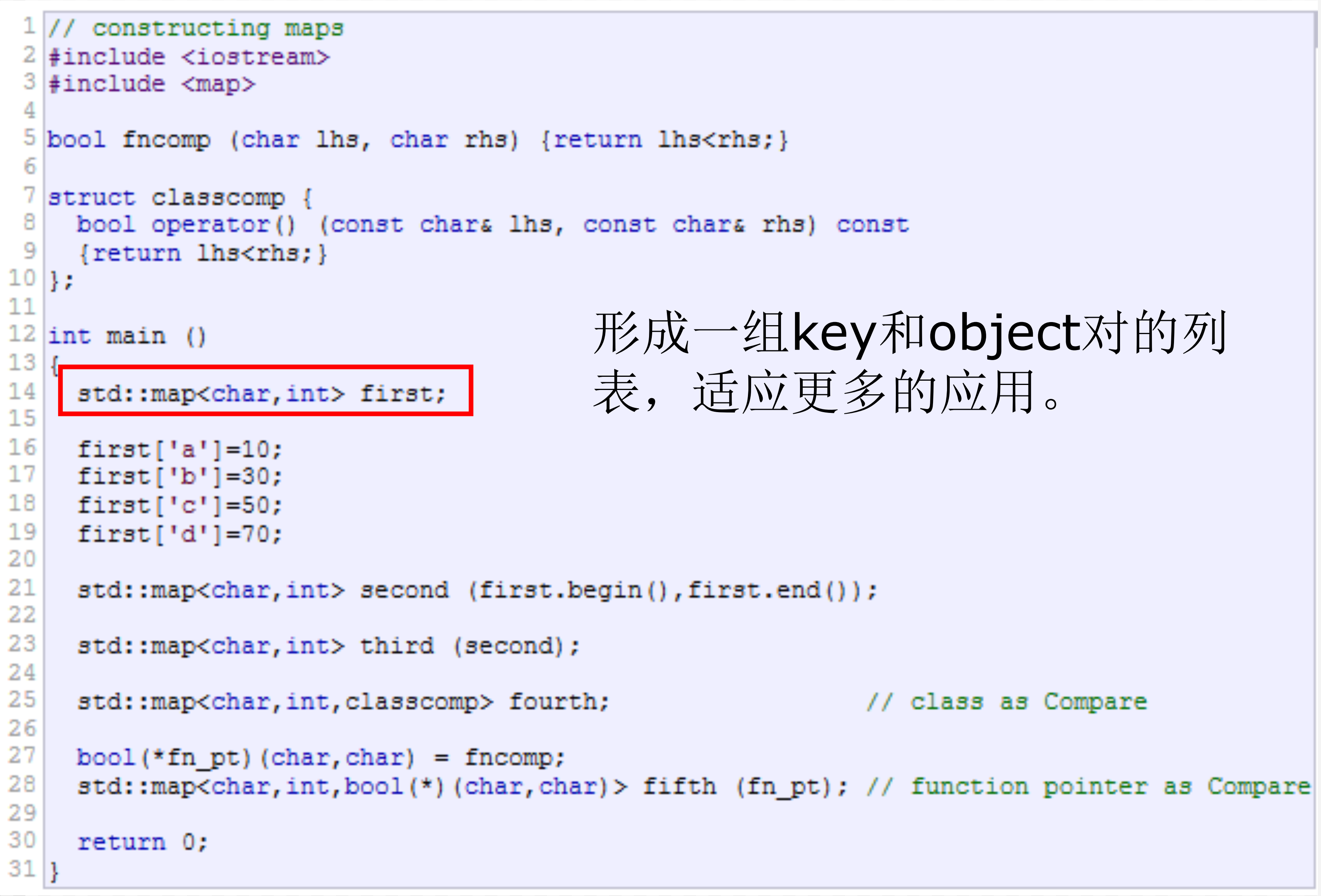
```
// vector::begin/end
#include <iostream>
#include <vector>

int main ()
{
  std::vector<int> myvector;
  for (int i=1; i<=5; i++) myvector.push_back(i);

  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

STL vector更方便，例如，排序，sort



```
// sort algorithm example
#include <iostream>     // std::cout
#include <algorithm>    // std::sort
#include <vector>       // std::vector

bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;

int main () {
  int myints[] = {32,71,12,45,26,80,53,33};
  std::vector<int> myvector (myints, myints+8);               // 32 71 12 45 26 80 53 33

  // using default comparison (operator <):
  std::sort (myvector.begin(), myvector.begin()+4);           //(12 32 45 71)26 80 53 33

  // using function as comp
  std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71(26 33 53 80)

  // using object as comp
  std::sort (myvector.begin(), myvector.end(), myobject);     //(12 26 32 33 45 53 71 80)

  // print out content:
  std::cout << "myvector contains:";
  for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

STL map

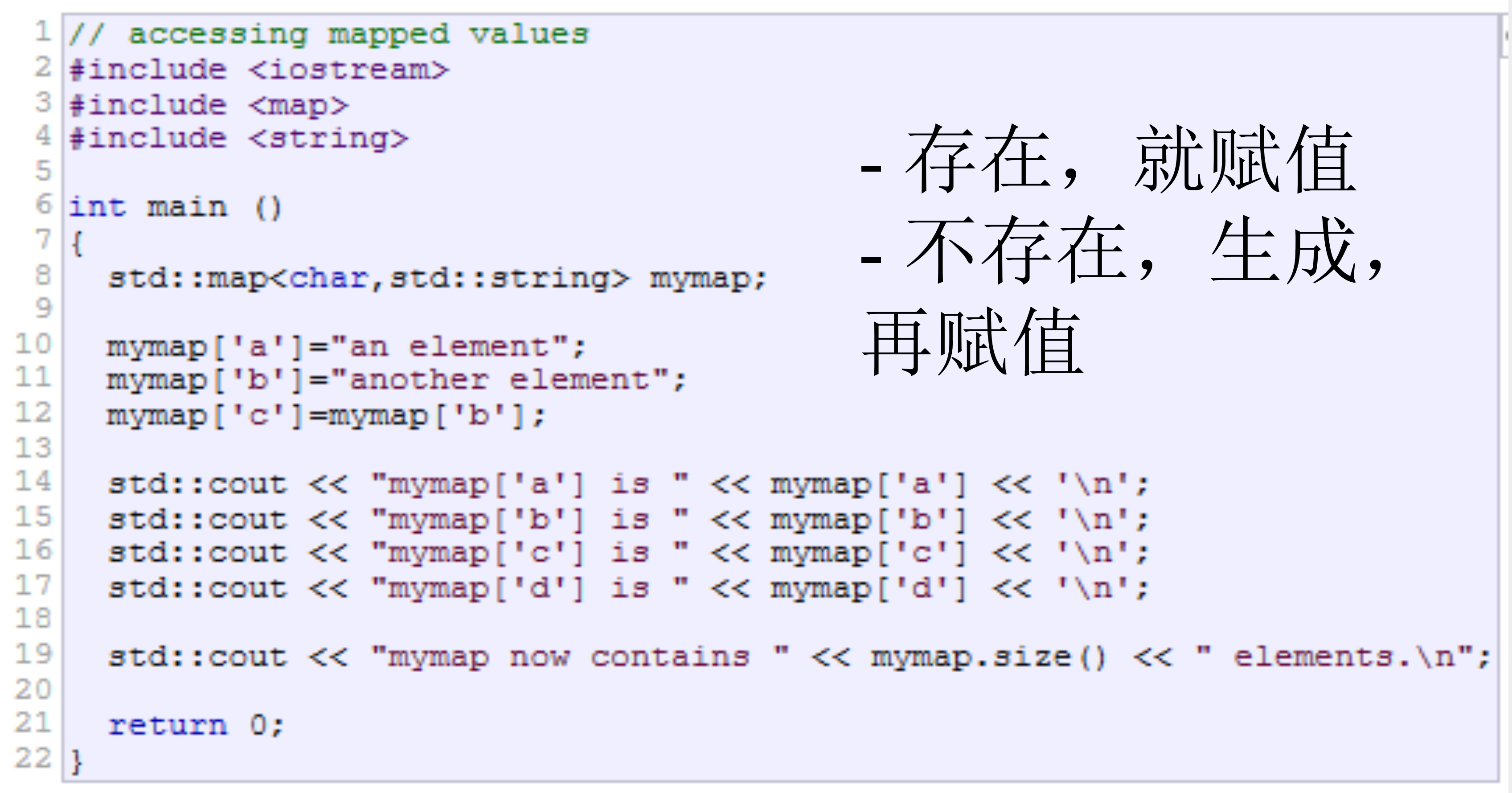


```
// constructing maps
#include <iostream>
#include <map>

bool fncomp (char lhs, char rhs) {return lhs<rhs;}

struct classcomp {
  bool operator() (const char& lhs, const char& rhs) const
  {return lhs<rhs;}
};

int main ()
{
  std::map<char,int> first;

  first['a']=10;
  first['b']=30;
  first['c']=50;
  first['d']=70;

  std::map<char,int> second (first.begin(),first.end());

  std::map<char,int> third (second);

  std::map<char,int,classcomp> fourth;                 // class as Compare

  bool(*fn_pt)(char,char) = fncomp;
  std::map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as Compare

  return 0;
}
```

形成一组key和object对的列表，适应更多的应用。

map的一系列功能

find，insert，operator[]，rbegin，rend



```
// accessing mapped values
#include <iostream>
#include <map>
#include <string>

int main ()
{
  std::map<char,std::string> mymap;

  mymap['a']="an element";
  mymap['b']="another element";
  mymap['c']=mymap['b'];

  std::cout << "mymap['a'] is " << mymap['a'] << '\n';
  std::cout << "mymap['b'] is " << mymap['b'] << '\n';
  std::cout << "mymap['c'] is " << mymap['c'] << '\n';
  std::cout << "mymap['d'] is " << mymap['d'] << '\n';

  std::cout << "mymap now contains " << mymap.size() << " elements.\n";

  return 0;
}
```

- 存在，就赋值
- 不存在，生成，再赋值

作业

1. 编辑，并运行第26页关于vector的程序
   打印程序，及运行结果

2. 对第12页的程序，利用-g编译选线，用dgb调试，验证确实调用了派生类的成员函数Area()