



四川大學  
SICHUAN UNIVERSITY

# 复习课

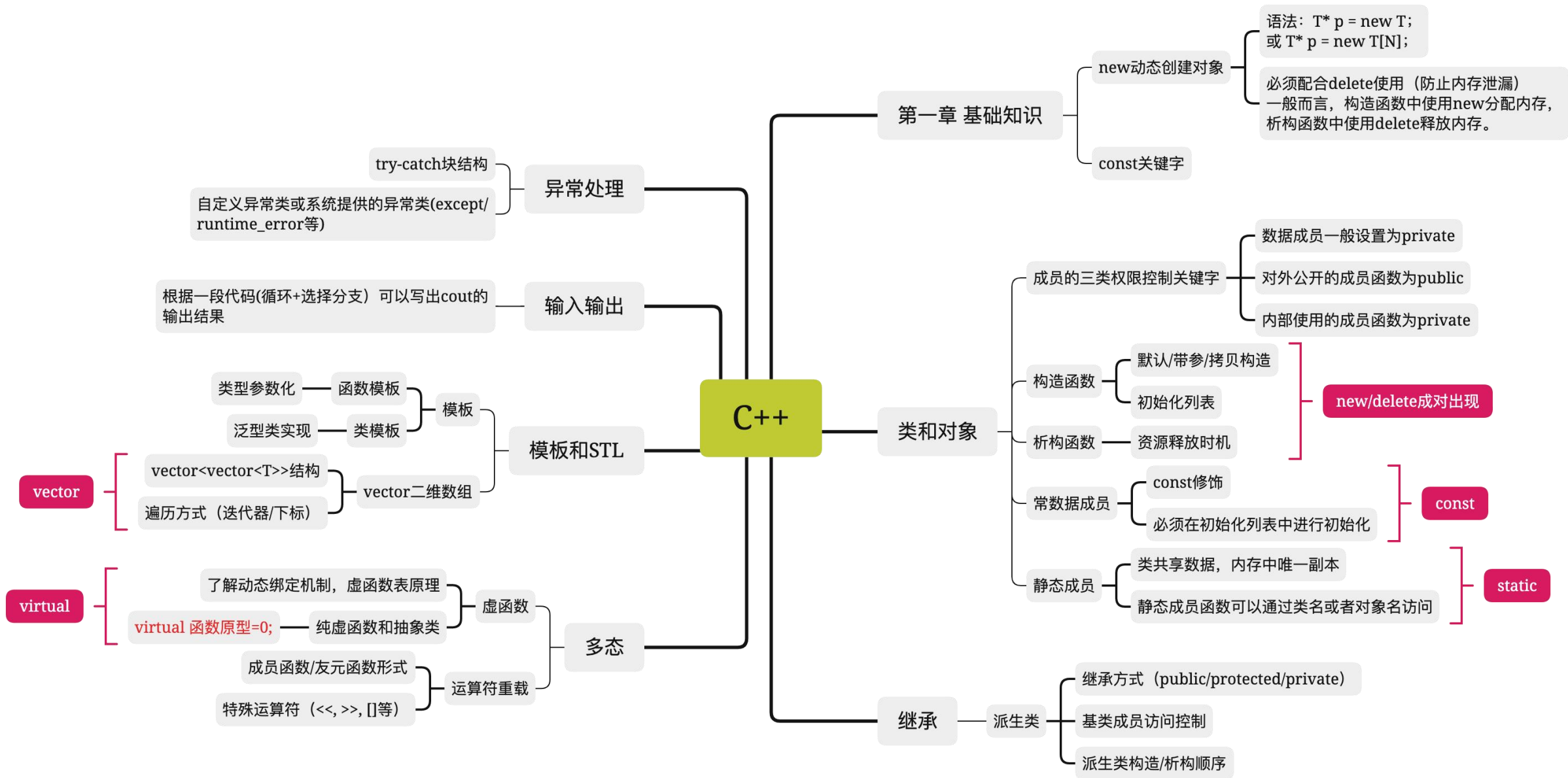
计算机学院（软件学院、智能科学与技术学院）

周 佩

❖ 主观题，与2024年期末的HardDisk考题类似，**逐步完成程序实现预期游戏功能。**

- 填空题，填写关键字/参数/创建对象等；1行代码左右，~10题\*2分/题
- 构造函数/析构函数/成员函数/类的实现，~7题\*(5-10)分/题
- 分析程序的输出，8分
- 其他：静态方法声明、创建对象，~10分

- ❖ 第1章 基础知识 new动态创建对象
- ❖ 第2章 类和对象 构造函数/析构函数/静态成员/常数  
据成员
- ❖ 第3章 继承 派生类
- ❖ 第4章 多态 虚函数 ~~运算符重载~~
- ❖ 第5章 ~~模板~~
- ❖ 第6章 STL **vector**二维数组的使用
- ❖ 第7章 输入输出 写输出结果
- ❖ 第8章 异常处理



❖ 常对象成员 { 常数据成员  
常成员函数

## ❖ (1) 常数据成员

- 常数据成员的声明和作用与普通的常变量类似
- 在程序运行过程中常数据成员的值不能修改
- 常变量在声明的同时必须初始化
- 注意**常数据成员**在初始化时必须使用构造函数的参数**初始化列表**

## ❖ (1) 常数据成员

- 例：若**Box**类中的数据成员**length**声明为常数据成员，其构造函数：

不能使用这种形式进行初始化

```
Box::Box(float L, float w, float h)
{   length = L;   width = w; height = h; }
```

×

```
Box::Box(float L, float w, float h):length(L)
{   width = w;   height = h; }
```

✓



- ❖ 指针指向的是内存中对象所占用空间的首地址
- ❖ 定义指向对象的指针的一般形式是：

类名 \*指针名；

- ❖ 通过指向对象的指针访问对象的成员有两种形式
  - 通过 “->”调用对象的公用成员函数
  - 通过 “\*” 运算符得到对象，然后再使用 “.”运算符调用对象的公用成员函数

【例2.11】对象指针举例：定义一个长方形类，要求能设定长和宽的值，并提供查询、计算面积和输出信息的功能。

## 【例2.11】对象指针举例（长方形类）。

```
class Rectangle
```

```
{public:
```

```
    Rectangle (){ length=0; width=0; }
```

```
    Rectangle ( double l, double w){ length=l; width=w; }
```

```
    ~ Rectangle (){};
```

```
    void setSize( double l, double w) { length=l; width=w; }
```

```
    double getLength( ) {return length;}
```

```
    double getWidth( ) {return width;}
```

```
    double getArea( );
```

```
    void print();
```

```
private:
```

```
    double length;
```

```
    double width;
```

```
};
```

```
double Rectangle:: getArea( )
```

```
{ return length*width; }
```

```
void Rectangle::print()
```

```
{ cout<<"长方形的长是"<< length<<"宽是"<< width  
    <<"面积是"<< getArea( )<<endl;
```

```
}
```

```
int main( )
```

```
{
```

```
    Rectangle *rp= new Rectangle(10,5);
```

```
    rp->print();
```

```
    rp->setSize( 45, 20 );
```

```
    cout<<"长45宽20的长方形的面积是"<<
```

```
    rp->getArea()<<endl;
```

```
    delete rp;
```

```
    return 0;
```

```
}
```



❖ 在 C++ 中，对象指针初始化时，最佳赋值为 **nullptr**，其次为 **NULL**

❖ nullptr 与 NULL 的区别

■ nullptr: `MyClass* obj = nullptr; // 清晰、安全、现代`

- C++11 引入的关键字，类型为 `std::nullptr_t`，可隐式转换为任何指针类型（如 `int*`、`char*` 等）。
- 不会被误认为整型（如 0），避免函数重载时的二义性。

■ NULL:

- 传统 C/C++ 中的宏定义，通常是 0 或 `(void*)0`。
- 可能被编译器当作整型处理，导致意外的函数重载匹配。

```

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

class Student {
public:
    string name;
    int score;
    Student* next; // 指向下一个学生

    // 构造函数
    Student(string n, int s) : name(n), score(s),
next(nullptr) {}

    // 打印当前学生信息
    void printStudent() {
        cout << "Name: " << name << ", Score: " <<
score << endl;
    }
};

```

2025-6-15

```

int main() {
    // 动态创建两个学生
    Student* student1 = new Student("Alice", 90);
    Student* student2 = new Student("Bob", 85);

    // 链接两个学生
    student1->next = student2;

    // 遍历链表并打印
    Student* current = student1;
    while (current != nullptr) {
        current->printStudent();
        current = current->next;
    }

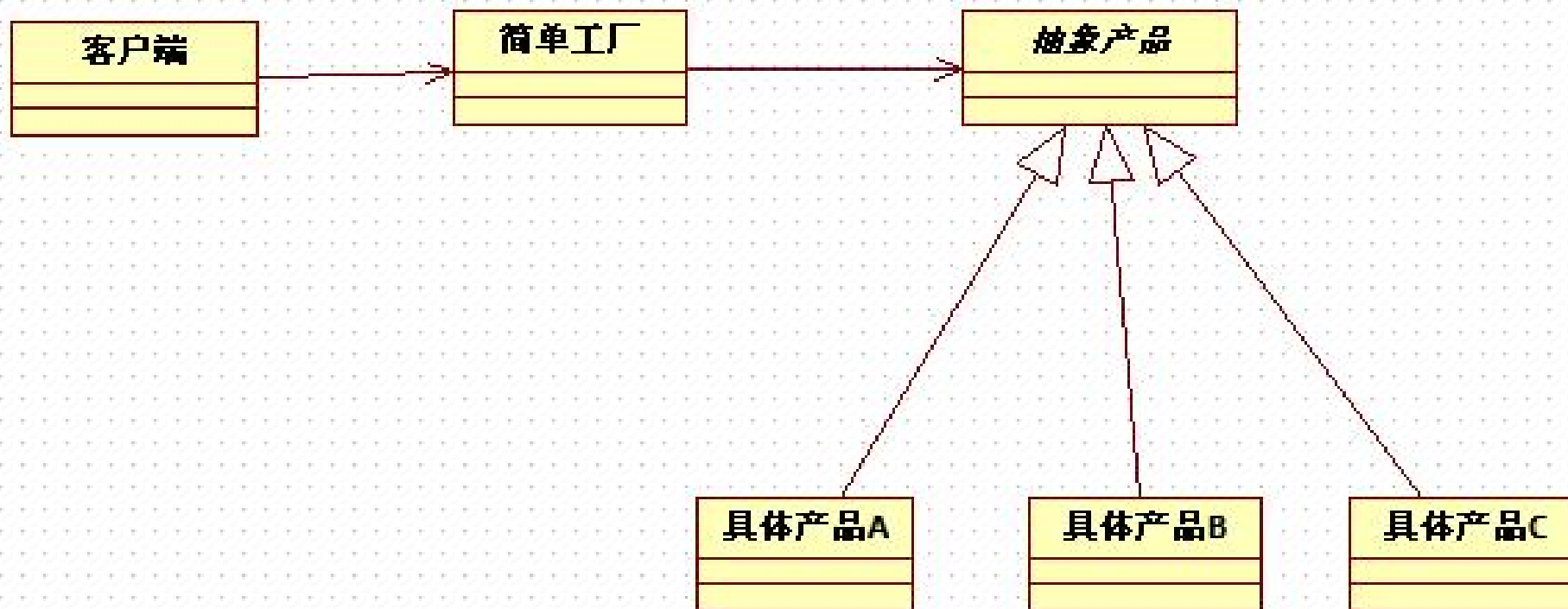
    // 释放内存
    delete student1;
    delete student2;
    student1 = nullptr; // 可选, 防止悬垂指针
    student2 = nullptr;
    return 0;
}

```

10

- ❖ 简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类（这些产品类继承自一个父类或接口）的实例。
- ❖ 简单工厂模式的工厂类一般是使用静态方法，通过接收的参数不同来返回不同的对象实例。

<https://www.cnblogs.com/qrxqrx/articles/6678872.html>



```

#include <iostream>
using namespace std;

class Product{//抽象产品
public:
    int op1,op2;
public:
    void setOpt(int o1,int o2){
        op1=o1;
        op2=o2;
    }
    virtual double getResult(){
        double result=0;
        return result;
    }
};

class AddProduct:public Product{
public:
    double getResult(){
        return op1+op2;
    }
};

class SubProduct:public Product{
public:
    double getResult(){
        return op1-op2;
    }
};

```

2025-6-15

```

class Creator{//工厂
public:

```

```

    static Product * CreatePro1()
    { oper = new AddProduct();
      oper.id = 1;
      return oper;}

```

```

    static Product * CreatePro2()
    { oper = new SubProduct();
      oper.id = 2;
      return oper;}

```

```

private:
    int id;

```

```

};

```

```

int main()
{

```

```

    int a=10,b=5;

```

```

    //客户给出操作参数

```

```

    Product * op = Creator::CreatePro1();

```

```

    op->setOpt(a,b);

```

```

    cout << op->getResult() << endl;//得到产品

```

```

    Product * op = Creator::CreatePro2();

```

```

    op->setOpt(a,b);

```

```

    cout << op->getResult() << endl;//得到产品

```

```

    return 0;

```

13

# 多态实例：几何形体处理程序

## ❖ 几何形体处理程序:

输入若干个几何形体的参数，要求按面积排序输出。输出时要指明形状。

## ❖ Input: 第一行是几何形体数目n(不超过100)，每行以一个字母c开头。

- 若c是'R'，代表矩形，本行后面跟着两个整数，分别是矩形的宽和高；
- 若c是'C'，代表圆，本行后面跟着一个整数代表其半径；
- 若c是'T'，代表三角形，本行后面跟着三个整数，带包三条边的长度；

## ❖ Output:

- 按面积从小到大依次输出每个几何形体的种类及面积。每行一个几何形体，
- 输出格式为：  
形体名称：面积



### Sample Input:

```
3
R 3 5
C 9
T 3 4 5
```

### Sample Output

```
Triangle:6
Rectangle:15
Circle:254.34
```



# 多态实例：几何形体处理程序

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
using namespace std;
class CShape
{
public:
    virtual double Area() = 0; //纯虚函数
    virtual void PrintInfo() = 0;
};
class CRectangle:public CShape
{
public:
    int w, h;
    virtual double Area();
    virtual void PrintInfo();
};
```

```
class CCircle:public CShape
{
public:
    int r;
    virtual double Area();
    virtual void PrintInfo();
};
class CTriangle:public CShape
{
public:
    int a,b,c;
    virtual double Area();
    virtual void PrintInfo();
};
```

## 多态实例：几何形体处理程序

```
double CRectangle::Area() {  
    return w * h;  
}  
void CRectangle::PrintInfo() {  
    cout << "Rectangle:" << Area() << endl;  
}  
double CCircle::Area() {  
    return 3.14 * r * r;  
}  
void CCircle::PrintInfo() {  
    cout << "Circle:" << Area() << endl;  
}  
double CTriangle::Area() {  
    double p = ( a + b + c ) / 2.0;  
    return sqrt(p * ( p - a ) * ( p - b ) * ( p - c ));  
}  
void CTriangle::PrintInfo() {  
    cout << "Triangle:" << Area() << endl;  
}
```

# 多态实例：几何形体处理程序

```
CShape * pShapes[100];
```

```
int MyCompare(const void * s1, const void * s2);
```

```
int main()
```

```
{
```

```
    int i; int n;
```

```
    CRectangle * pr;
```

```
    CCircle * pc;
```

```
    CTriangle * pt;
```

```
    cin >> n;
```

```
    for( i = 0; i < n; i ++ ) {
```

```
        char c; cin >> c;
```

```
        switch(c) {
```

```
            case 'R':
```

```
                pr = new CRectangle();
```

```
                cin >> pr->w >> pr->h;
```

```
                pShapes[i] = pr;
```

```
                break;
```

```
            case 'C':
```

```
                pc = new CCircle();
```

```
                cin >> pc->r;
```

```
                pShapes[i] = pc; break;
```

```
            case 'T':
```

```
                pt = new CTriangle();
```

```
                cin >> pt->a >> pt->b >> pt->c;
```

```
                pShapes[i] = pt; break;
```

```
        }
```

```
    }
```

```
    qsort(pShapes,n,sizeof( CShape*), MyCompare);
```

```
    for( i = 0; i < n; i ++)
```

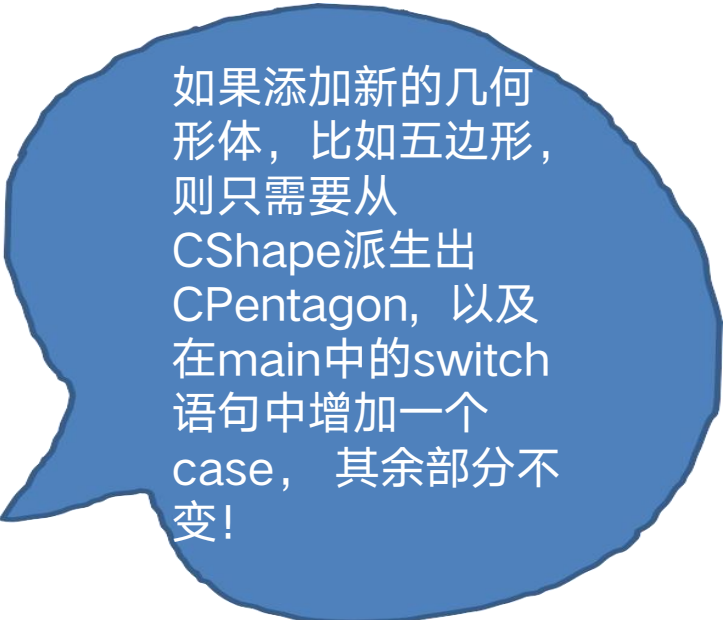
```
        pShapes[i]->PrintInfo();
```

```
    return 0;
```

```
}
```

## 多态实例：几何形体处理程序

```
int MyCompare(const void * s1, const void * s2)
{
    double a1,a2;
    CShape * * p1 ; // s1,s2 是 void * , 不可写 “* s1” 来取得s1指向的内容
    CShape * * p2;
    p1 = ( CShape * * ) s1; //s1,s2指向pShapes数组中的元素，数组元素的类型是CShape *
    p2 = ( CShape * * ) s2; // 故 p1,p2都是指向指针的指针，类型为 CShape **
    a1 = (*p1)->Area();      // * p1 的类型是 Cshape * ,是基类指针，故此句为多态
    a2 = (*p2)->Area();
    if( a1 < a2 )
        return -1;
    else if ( a2 < a1 )
        return 1;
    else
        return 0;
}
```



如果添加新的几何形体，比如五边形，则只需要从CShape派生出CPentagon，以及在main中的switch语句中增加一个case，其余部分不变！



- ◎用基类指针数组存放指向各种派生类对象的指针，然后遍历该数组，就能对各个派生类对象做各种操作，是很常用的做法



### ❖ 题目要求:

- 设计抽象基类**Shape**，包含纯虚函数**getArea()**
- 派生**Circle**和**Rectangle**类实现多态
- 使用**vector<vector<Shape\*>>**存储不同组的图形
- 实现:
  - 动态内存分配
  - 多态调用
  - 两种遍历方式
  - 内存释放



# 二维指针向量操作 (动态内存+多态)



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

// 抽象基类

```
class Shape {
public:
    virtual double getArea() const = 0;
    virtual void print() const = 0;
    virtual ~Shape() {} // 基类虚析构
};
```

// 派生类1

```
class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}
    double getArea() const override
    { return 3.14159 * radius * radius; }
    void print() const override
    { cout << "Circle(r=" << radius << ")"; }
};
```

// 派生类2

```
class Rectangle : public Shape {
    double w, h;
public:
    Rectangle(double width, double height) : w(width),
    h(height) {}
    double getArea() const
    { return w * h; }
    void print() const
    { cout << "Rectangle(" << w << "x" << h << ")"; }
};
```

# 二维指针向量操作 (动态内存+多态)



```
int main() {  
    // 创建二维指针向量 (3组图形)  
    vector<vector<Shape*>> groups = {  
        { new Circle(2.0), new Rectangle(3,4) },  
        { new Rectangle(5,5), new Circle(1.5) },  
        { new Circle(3.0), new Rectangle(2,6), new Circle(4.0) }  
    };  
  
    // 方法1: 下标访问  
    cout << "=== 下标遍历 ===" << endl;  
    for (int i = 0; i < groups.size(); ++i) {  
        cout << "Group " << i+1 << ":\n";  
        double total = 0;  
        for (int j = 0; j < groups[i].size(); ++j) {  
            groups[i][j]->print();  
            cout << " area=" << groups[i][j]->getArea() << endl;  
            total += groups[i][j]->getArea();  
        }  
        cout << "Total area: " << total << "\n\n";  
    }  
}
```

2025-6-15

// 方法2: 迭代器访问

```
cout << "=== 迭代器遍历 ===" << endl;  
auto it = groups.begin();  
while (it != groups.end()) {  
    cout << "Group " << it - groups.begin() + 1 << ":\n";  
    double total = 0;  
    for (auto shape_it = it->begin(); shape_it != it->end(); ++shape_it) {  
        (*shape_it)->print();  
        cout << " area=" << (*shape_it)->getArea() << endl;  
        total += (*shape_it)->getArea();  
    }  
    cout << "Total area: " << total << "\n\n";  
    ++it;  
}  
  
// 内存释放  
for (auto& group : groups) {  
    for (auto& pShape : group) {  
        delete pShape;  
    }  
}  
  
return 0;  
}
```

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

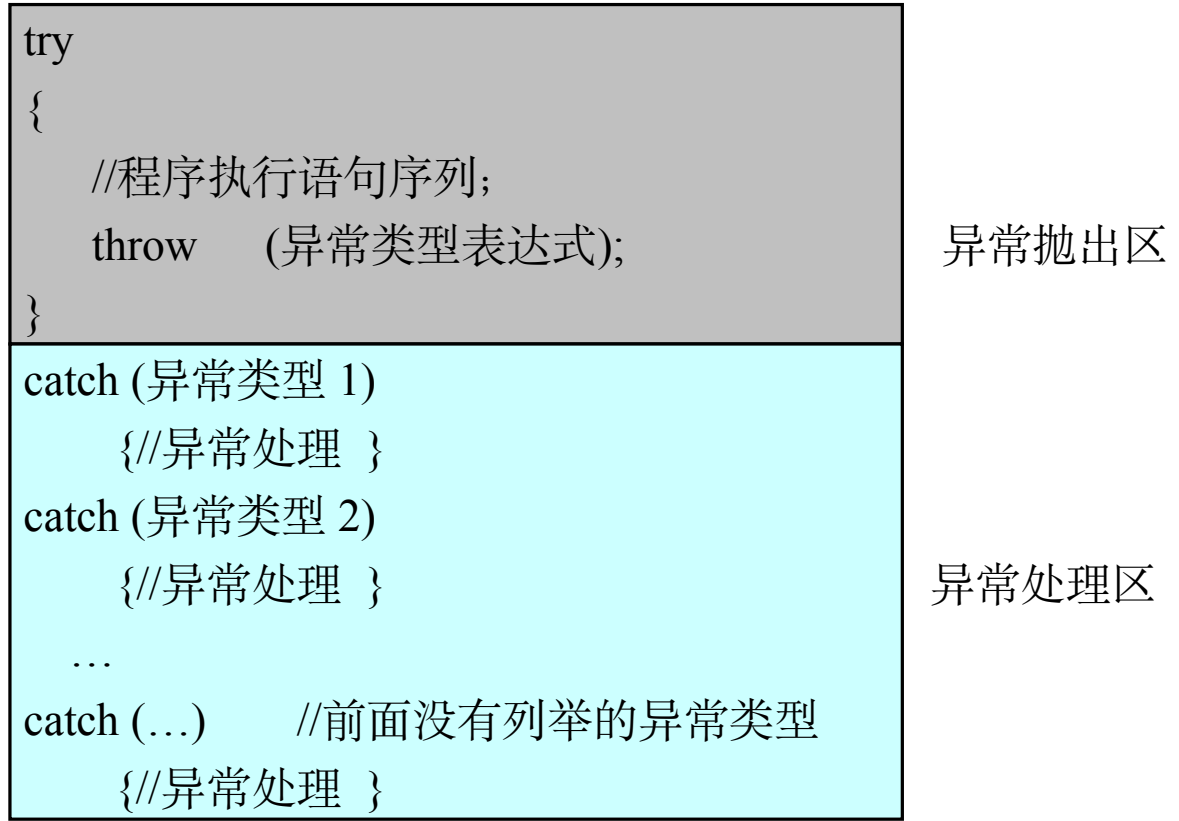
```
int main() {
    int n = 4;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if ((i + j) % 3 == 0) {
                cout << "@";
            } else if (j % 2 == 1) {
                cout << "* ";
            } else {
                cout << "# ";
            }
        }
        cout << endl;
    }
    return 0;
}
```

## ❖考察点:

- 嵌套循环的执行顺序
- 条件分支的优先级判断
- 模运算的实际应用
- 输出格式控制（空格与换行）

```
* @ * #
@ # * @
* # @ #
* @ * #
```

❖ 异常处理机制的主体有两大部分：异常抛出区**try**，异常处理区**catch**，二者作为一个整体出现，构成**try-catch**结构。它们不能单独使用，也不能在其间插入语句。



//try-throw-catch 后继续执行的语句

```

#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;
class BankAccount {
private:
    double balance;
public:
    BankAccount(double b) : balance(b) {}
    void withdraw(double amount) {
        if (amount > balance) {
            throw runtime_error("Insufficient funds: cannot withdraw " +
                                to_string(amount) + " from balance " + to_string(balance));
        }
        balance -= amount;
        cout << "Withdrew " << amount << ". New balance: " << balance << endl;
    }

    double getBalance() const { return balance; }
};

```

```

int main() {
    BankAccount account(1000.0);

    try {
        cout << "Current balance: " << account.getBalance() << endl;

        // 正常取款
        account.withdraw(500.0);

        // 尝试超额取款 - 会抛出异常
        account.withdraw(600.0);

        // 这行不会执行
        cout << "This message won't be displayed" << endl;
    }
    catch (const runtime_error& e) {
        cerr << "Error: " << e.what() << endl;
        cerr << "Transaction failed. Current balance remains: "
              << account.getBalance() << endl;
    }

    return 0;
}

```

下列程序运行结果为:

```
class A{
```

```
public:
```

```
    ~A( ){cout<<"A"<<"\n"; }
```

```
};
```

```
char fun0() {
```

```
    A A1;
```

```
    throw('E');
```

```
    return '0';
```

```
}
```

```
int main(){
```

```
    try{
```

```
        cout<<fun0()<<"\n";}
```

```
    catch(char c) {
```

```
        cout<<c<<"\n";}
```

```
    return 0;
```

```
}
```

A  
E