C++

河北师范大学软件学院
Software College of Hebei Normal University

第九讲 类和对象（五）

C++备课组 丁盟

自我介绍

丁盟

qq：2622885094

上一讲教学目标

- 了解友元的概念
- 掌握友元函数的使用
- 掌握友元类的使用

本讲教学目标

- 掌握C++中对象数组的使用
- 理解C++中this指针的含义及使用
- 掌握C++中对象引用的使用

1 对象数组

2 对象及对象成员与指针

3 对象引用及对象的常引用

4 对象作为函数参数

对象数组

- 回忆我们曾经学过的数组

```
int arr[3];
struct student arr[3];
int *p[3];
int ((*p)[3])[5];
void (*p[5])( int x,int y );
```

类型相同的变量的集合

对象数组

- 对象数组的概念：

如果数组元素都为类类型，我们把这样的数组称为对象数组。

```
string  arr[9];
```

对象数组

- 声明格式：

和普通数组相同。

```
类名    数组名[元素个数]；
类名    数组名[行数][列数];
```

多维数组的格式与此类似

对象数组

- 初始化：
  - 回忆内置类型数组的初始化。

方式	例
全部	`int a[3] = { 1,2,3 };`
部分	`int a[3] = { 1,2 };`
省略长度	`int a[] = {1,2,3};`

  - 对象数组的初始化

方式	例
全部	Point a[3] = { Point(1,1), Point(2,2), Point(3,3)};
部分	Point a[3] = { Point(1,1), Point(2,2)};
省略长度	Point a[] = { Point(1,1), Point(2,2), Point(3,3)};

对象数组

[注意]

```
int arr[3];
```

内置类型的数组不赋初值则不初始化

```
Point arr[3];
```

对象数组如果不赋初值则自动调用默认构造函数初始化！

1 对象数组

2 对象及对象成员与指针

- 指向对象的指针
- 对象的const指针
- this指针

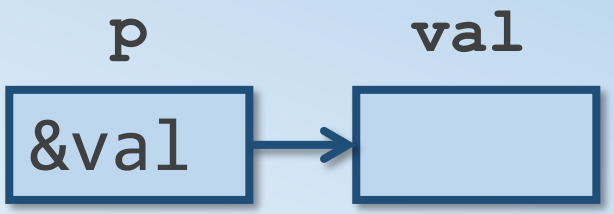
3 对象引用及对象的常引用

4 对象作为函数参数

指向对象的指针

- 回顾内置类型的指针

```
int     ival = 3;
float   fval = 4f;
char    cval = 'a';
double dval = 6.0;
int     * p_ival = &ival;
float   * p_fval = &fval;
char    * p_cval = &cval;
double * p_dval = &dval;
```

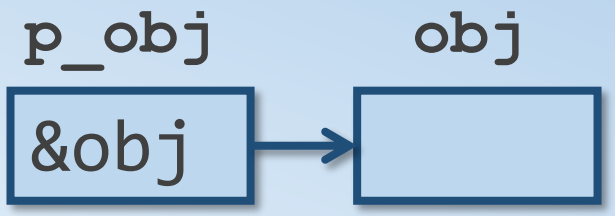


指向对象的指针与此也是类似的

指向对象的指针

- 对象指针的概念：对象的地址

```
class Atest {
public:
    Atest(int aX = 0) :m_iVal(aX){}
    void display() {cout << m_iVal << endl;}
private:
    int m_iVal;
};
int main(void) {
    Atest obj;
    Atest * p_obj = &obj;
    return 0;
}
```



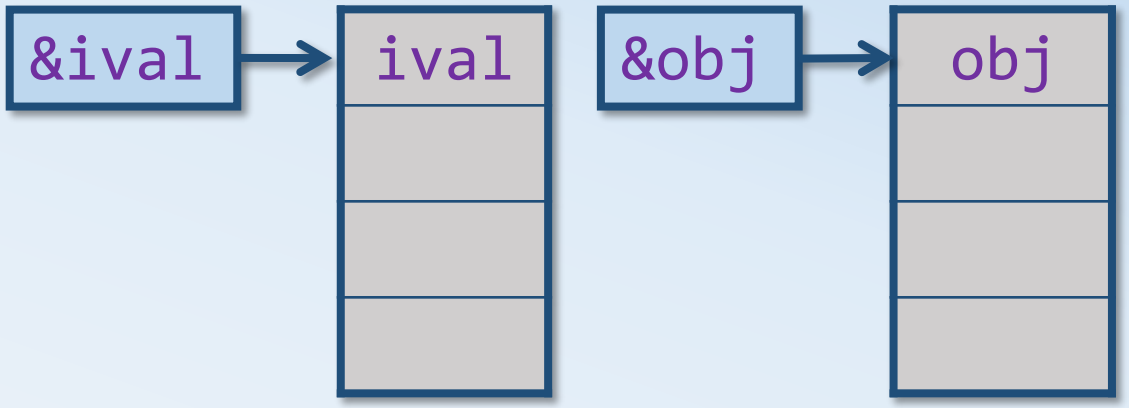
可见：对象的指针与内置类型的指针本质是相同的

指向对象的指针

[说明]

对象的地址(指针)与内置类型变量的地址(指针)本质相同都表示对象的首地址。

```
int ival;
int * p = &ival;
Atest obj;
Atest * pobj = &obj;
```



指向对象的指针

[注意]

1. 定义方法与内置类型指针相同。
2. 取地址的方法都是通过&。
3. 任何类型指针在32位机中都占用4byte。

指向对象的指针

- 引用对象成员的方法

  法一： 对象名.成员

  法二： (*指针变量名).成员
  指针变量 ->成员

  法三： (对象数组名+下标)->成员
  (*(对象数组名+下标)).成员

指向对象的指针

```
class Point {
public:
    Point( double aX = 0,
           double aY = 0 );
    double getX() const;
    double getY() const;
private:
    double m_dX;
    double m_dY;
};

double Point::getX() const {
    return m_dX;  }

double Point::getY() const {
    return m_dY; }
```

```
Point::Point( double aX, double aY ) {
    m_dX = aX;
    m_dY = aY;
}
int main(void) {
    Point array[3] =
         { Point(3,4), Point(5,8) };
    Point * p = array;
    for(; p < array+3;  p++) {
        cout << "("
        << p->getX()  << ","
        << (*p).getY() << ")"
        << endl;
    }
    return 0;
}
```

```
(3,4)
(5,8)
(0,0)
```

1 对象数组

2 对象及对象成员与指针

- 指向对象的指针
- 对象的const指针
- this指针

3 对象引用及对象的常引用

4 对象作为函数参数

对象的const指针

- 声明：与内置类型的const指针规则相同

```
const Point Kobj(3,4); // const 对象也必须初始化
Point obj; // 调用默认构造函数

Point * p3 = &Kobj; // error

const Point * Kp1 = &obj;
Point *p2 = &obj;
*Kp1 = Kobj; // error

const Point * const KKp1 = &Kobj;
cosnt Point * const KKp2 = &obj;
```

const对象只能被const指针或引用指向

非const对象可被const指针指向也可被非const指针指向

不能被修改

1 对象数组

2 对象及对象成员与指针

- 指向对象的指针
- 对象的const指针
- this指针

3 对象引用及对象的常引用

4 对象作为函数参数

this指针

❖ 问题的产生：

每个对象中的数据成员都分别占有存储空间，如果对同一个类定义了N个对象，则有N组同样大小的空间以存放N个对象中的数据成员，但是不同的对象都调用同一个函数代码段。

那么，当不同对象的成员函数引用数据成员时，怎么能保证引用的是指定对象的数据成员呢？

为了确定究竟是哪个对象在引用当前成员函数，C++引入了this指针。

this指针也是指向对象的指针，只不过它指向的是当前对象。

this指针

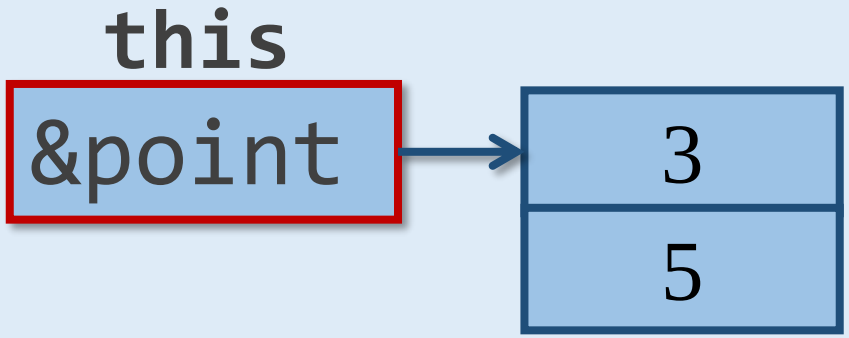
- 什么是this指针：

this指针是指向当前对象的指针变量，每个成员函数都含有一个指向本类对象的this指针。

this指针

- 一般格式：

```
class Point {
public:
    Point(int aX = 0, int aY = 0);
    void print();
private:
    int m_iX;
    int m_iY;
};
int main(void) {
    Point point(3,5);
    point.print();
    return 0;
}
```



this指针

[注意]

- 对象引用成员函数时，在函数的参数表中会自动添加一个该对象的指针

```
void Point::print(Point *const this)
```

- 系统会为每个一成员函数自动添加一个this指针

```
a1.print();  ➔  Point::print(&a1)
```

this指针

- 通常编程者不必人为的在形参中添加this指针，编译不必将对象的地址传给this指针
- this指针不能显示的定义，我们只能使用它，通常如果希望成员函数返回本类对象或者本对象地址时显示的使用this指针

this指针

- 如果成员函数返回本类对象的引用或指针则访问成员的方式有些特殊

```
class Test {
public:
    Test &print1( ) {
        cout << "print1";
        return *this;
    }
    Test *print2( ) {
        cout << "print2";
        return this;
    }
private:
    int m_iVal;
};
```

```
int main(void)
{
    Test obj;
    obj.print1( ).print2( );
    obj.print2( )
            ->print1( );

    return 0;
}
```

this指针

❖this指针与静态成员函数

类的静态成员函数与静态数据成员都是属于类的，不是属于对象的，其不含有this指针，所以也就无法访问非静态数据成员(为什么静态成员函数不能调用非静态数据成员)。

this指针

```
class Point
{
public:
    static void print();
    void show();

private:
    static int m_iCount;
    int m_iX;
    int m_iY;
};

int Point::m_iCount = 0;
```

```
void Point::show() {
    cout << m_iX << endl;
    cout << m_iY << endl;
}
void Point::print() {
    cout << m_iCount << endl;
}
int main(void){
    Point point;
    point.print();<->Point::print();
    point.show();
    <->  Point::show(&point);
    return 0;
}
```

this指针

编译前

```
class Point {
public:
    static void print();
    void show();
private:
    static int m_iCount;
    int m_iX;
    int m_iY;
};
```

编译后

```
void Point::print();
void Point::show(Point * const this);
```

调用时

```
int main(void) {
    Point point;
    point::print();
    point.show();
->  Point::show(&point);

    return 0;
}
```

this指针

❖this指针与const函数

非静态const成员函数的const修饰不是修饰该函数的，而是修饰隐式this指针的，全局函数没有this指针，自然不能在其后加const修饰。

this指针

```
class Point {
public:
    void print() const;
    void set(int aX=0,
             int aY=0);
private:
    int m_iX;
    int m_iY;
};
void Point::print() const
{
    cout << m_iX << " "
         << m_iY << endl;
}
```

```
void Point::set(int aX,
                int aY)
{
    m_iX = aX;
    m_iY = aY;
}
int main(void)
{
    Point point;
    point.set(5, 10);
    point.print();

    return 0;
}
```

this指针

编译前

```
class Point
{
public:
    void print() const;
    void set(int aX,int aY);
private:
    int m_iX;
    int m_iY;
};
```

调用时

```
int main(void)
{
    Point point;
    point.set(&point, 5, 10);
    point.print(&point);

    return 0;
}
```

编译后

```
void Point::print(const Point * const this);
void Point::set(Point * const this, int aX,int aY);
```

1 对象数组

2 对象及对象成员与指针

3 对象引用及对象的常引用

4 对象作为函数参数

对象引用及对象的常引用

- 回忆内置类型的引用

```
int val = 3;
const int &Kval = val;
const int &Kv = Kval;//可以给别名起别名
```

必须初始化，不能再作为其它变量的引用

```
const int val = 3;
int &Kval = val; //ERROR
const int &Kval = val;
```

常变量只能被const指针或引用指向

```
const int &Kval = 3;
```

const引用可以指向字面值常量

对象引用及对象的常引用

```
class Test
{
public:
    Test(int aX = 0)
    {
        m_iVal = aX;
    }
    void display()
    {
        cout << m_iVal;
    }
Private:
    int m_iVal;
};
```

```
int main(void) {
    Test obj1;
    Test &ref = obj1;
    const Test &KRef = obj1;
    const Test obj2;
    const Test &KRef1 = obj2;
    // 下面哪个对？
    const Test * KP = &obj2;
    Test * const KP = &obj2;

    Test &ref2 = obj2;//Error
}
```

const对象obj2只能被const引用或const指针指向

对象引用及对象的常引用

```
Point p1(10,20), p2(30,40);   Point & pr = p1;
void f() {
    cout << "p1=" << p1.GetX() << "," << p1.GetY() << "  ";
    cout << "p2=" << p2.GetX() << "," << p2.GetY() << "  ";
    cout << "pr=" << pr.GetX() << "," << pr.GetY() << endl;
}
int main(void) {
    cout << "original p1,p2,pr" << endl;
    f();   pr = p2;
    cout << "after pr=p2, p1,p2,pr:" << endl;
    f();   pr = Point(100,200);
    cout << "after pr=Point(100,200),  p1,p2,pr:" << endl;
    f();
    return 0;
}
```

对象引用及对象的常引用

```
original p1,p2,pr
p1=10,20  p2=30,40  pr=10,20

after pr=p2, p1,p2,pr:
p1=30,40  p2=30,40  pr=30,40

after pr=Point(100,200),  p1,p2,pr:
p1=100,200  p2=30,40  pr=100,200

请按任意键继续. . .
```

1 对象数组

2 对象及对象成员与指针

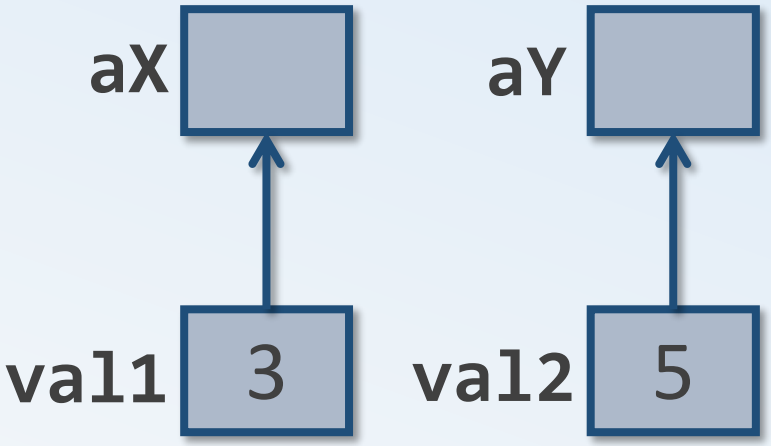
3 对象引用及对象的常引用

4 对象作为函数参数

- 对象作函数参数
- 对象指针作函数参数
- 对象引用作函数参数

对象作函数参数

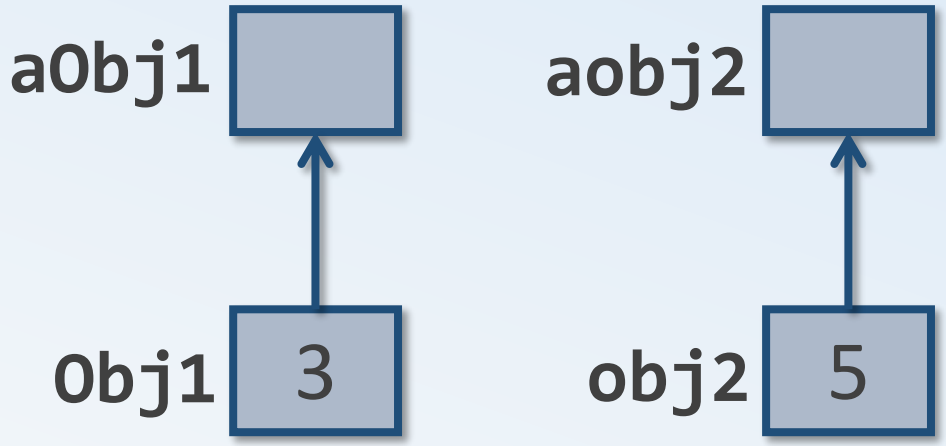
- 回忆变量作为函数参数
  - 形参开辟内存单元，值传递




```
void swap(int aX, int aY) {
     int temp = aX;
     aX = aY;
     aY = temp;
}
int main(void) {
     int val1 = 3, val2 = 5;
     swap(val1,val2);
     return 0;
}
```

对象作函数参数

- 对象作为函数参数
  - 形参开辟内存单元，值传递
  - 调用拷贝构造函数



- 特点：值传递
- 缺点：开销大

```
void swap(Atest aObj1,
          Atest aObj2) {
    Atest temp = aObj1;
    aObj1 = aObj2;
    aObj2 = temp;
}
int main(void) {
    Atest obj1(3), obj2(5);
    swap(obj1, obj2);
    obj1.display();
    obj2.display();
    return 0;
}
```

1 对象数组

2 对象及对象成员与指针

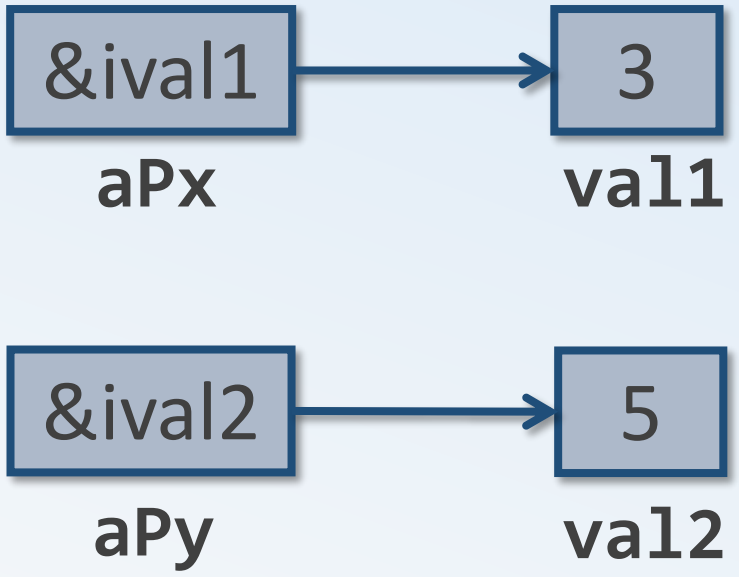
3 对象引用及对象的常引用

4 对象作为函数参数

- 对象作函数参数
- 对象指针作函数参数
- 对象引用作函数参数

对象指针作函数参数

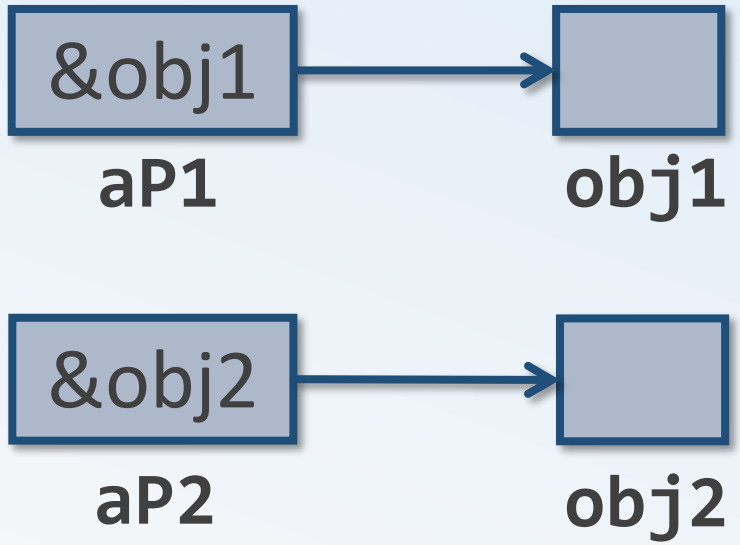
- 回忆指针变量作为函数参数
  - 传递变量地址
  - 形参开辟内存单元，值传递




```
void  swap(int * aPx,
           int * aPy) {
    int temp = *aPx;
    *aPx = *aPy;
    *aPy = temp;
}
int main(void) {
    int val1 = 3;
    int val2 = 5;
    swap(&val1, &val2);
    return 0;
}
```

对象指针作函数参数

- 对象指针作为函数参数
  - 传递对象地址
  - 形参开辟内存单元，值传递
  - 不会调用拷贝构造函数




```
void swap(Atest * aP1,
          Atest * aP2) {
    Atest temp = *aP1;
    *aP1 = *aP2;
    *aP2 = temp;
}
int main(void) {
    Atest obj1(3), obj2(5);
    swap(&obj1, &obj2);
    obj1.display();
    obj2.display();
    return 0;
}
```

1 对象数组

2 对象及对象成员与指针

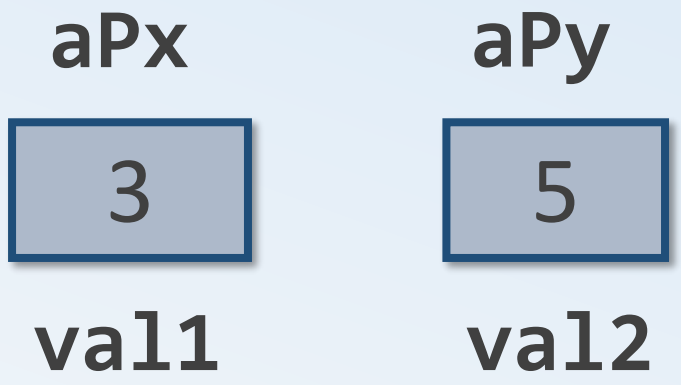
3 对象引用及对象的常引用

4 对象作为函数参数

- 对象作函数参数
- 对象指针作函数参数
- 对象引用作函数参数

对象引用作函数参数

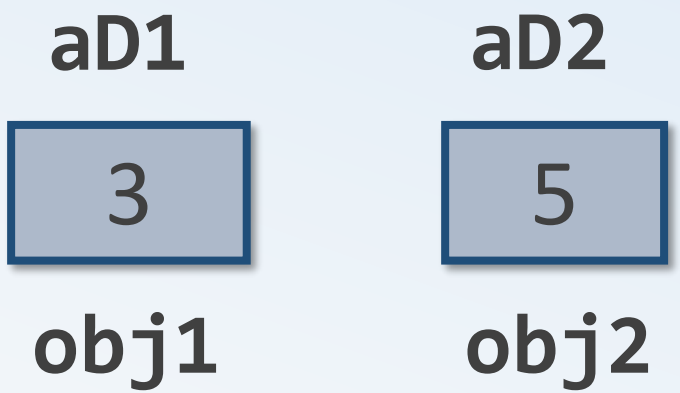
- 回忆变量引用作为函数参数
  - 形参不开辟内存




```
void swap(int & aX,
          int & aY) {
    int temp = aX;
    aX = aY;
    aY = temp;
}
int main(void) {
    int val1 = 3
    int val2 = 5;
    swap(val1,val2);
    return 0;
}
```

对象引用作函数参数

- 对象引用作函数参数
  - 形参不开辟内存
  - 不会调用拷贝构造函数
  - 推荐使用




```
void swap(Atest &aD1,
          Atest &aD2) {
    Atest temp = aD1;
    aD1 = aD2;
    aD2 = temp;
}
int main(void) {
    Atest obj1(3), obj2(5);
    swap(obj1, obj2);
    obj1.display();
    obj2.display();
    return 0;
}
```

本讲教学目标

- 掌握C++中对象数组的使用
- 理解C++中this指针的含义及使用
- 掌握C++中对象引用的使用


THANKS