2023年春季学期

软件设计实践

C++面向对象程序设计(1)

类与对象初步

谢 涛　　马 郓

本讲概览

- 用对象进行抽象
- 类的声明、定义和使用
  - 成员变量与成员函数
  - 成员的访问范围
- 对象的生命周期
  - 构造函数
  - 复制构造函数
  - 类型转换构造函数
  - 析构函数

复杂的数据抽象

❑ 利用结构体来抽象复杂的数据

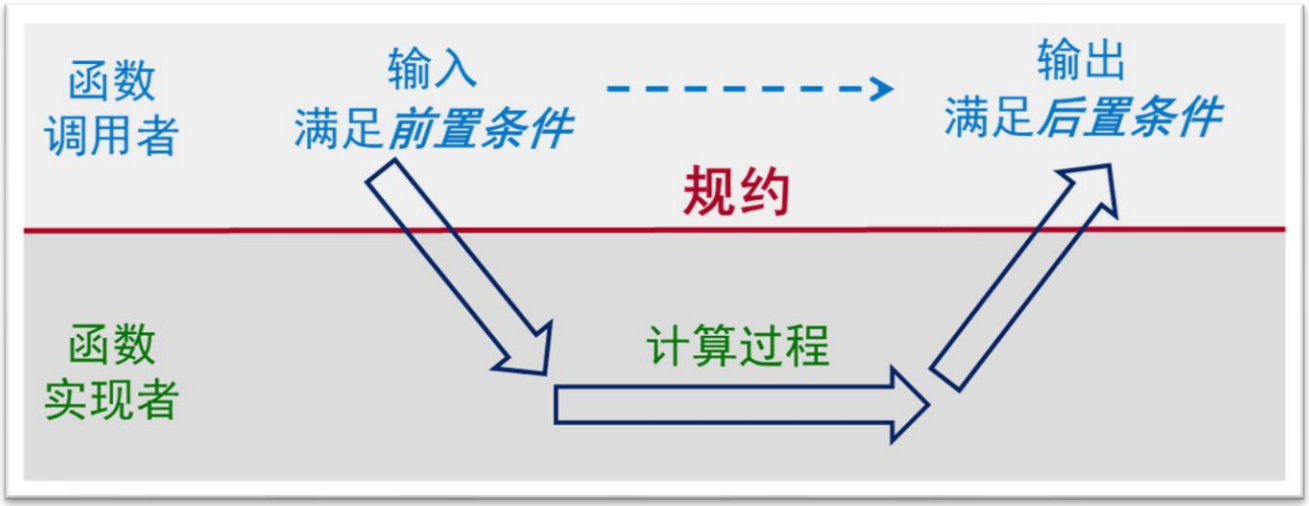
```
struct Rational {
  int numer; //分子
  int denom; //分母
};
```

```
struct Date {
  int year;  //年
  int month; //月
  int day;   //日
};
```

```
struct Card {
  char id[18];  //身份证号
  char name[10];//姓名
};
```

这样的数据抽象是否足够？有什么问题？

回顾函数对过程的抽象



复杂的数据抽象

❑利用结构体来抽象复杂的数据

不同的结构体定义

```
struct Rational {
  int numer; //分子
  int denom; //分母
};
```

输出分数“1/5”乘以“2/3”的结果

```
Rational r1, r2;
r1.numer = 1; r1.denom = 5;
r2.numer = 2; r2.denom = 3;
Rational r3;
r3.numer = r1.numer * r2.numer;
r3.denom = r1.denom * r2.denom;
cout << r3.numer << '/' << r3.denom << endl;
```

```
struct Rational {
  //0 分子, 1 分母
  int dimension[2];
};
```

```
Rational r1, r2;
r1.dimension[0] = 1; r1.dimension[1] = 5;
r2.dimension[0] = 2; r2.dimension[1] = 3;
Rational r3;
r3.dimension[0] = r1.dimension[0] * r2.dimension[0];
r3.dimension[1] = r1.dimension[1] * r2.dimension[1];
cout << r3.dimension[0] << '/' << r3.dimension[1] << endl;
```

```
struct Rational {
  //高4字节代表分子
  //低4字节代表分母
  long long dim;
};
```

```
Rational r1{0}, r2{0};
r1.dim |= (1LL<<32); r1.dim |= 5;
r2.dim |= (2LL<<32); r2.dim |= 3;
Rational r3{0};
r3.dim |= ((r1.dim >> 32) * (r2.dim >> 32)) << 32;
r3.dim |= (r1.dim << 32 >> 32) * (r2.dim << 32 >> 32);
cout << (r3.dim >> 32) << '/' << (r3.dim << 32 >> 32) << endl;
```

存在的问题：使用数据时需明确其具体的表示和实现方式！

复杂的数据抽象

❑ 利用函数实现数据抽象

➢ 通过函数规约将数据的使用和具体实现方式隔离

```
//Rational是表示分数的结构体类型，假设有以下三个函数
Rational rational(int n, int d);//根据分子和分母返回一个表示分数的变量
int numer(Rational r);          //获取分数变量r的分子
int denom(Rational r);          //获取分数变量r的分母
```

```
Rational mul_rationals(Rational x, Rational y) {
    return rational(numer(x) * numer(y), denom(x)
* denom(y));
}
```

```
void print_rational(Rational x) {
    cout << numer(x) << '/' << denom(x) << endl;
}
```

输出分数“1/5”乘以“2/3”的结果

```
Rational r1, r2;
r1 = rational(1, 5);
r2 = rational(2, 3);
Rational r3 = mul_rationals(r1, r2);
print_rational(r3);
```

无需了解数据的具体实现方式即可使用数据完成计算

复杂的数据抽象

❑ 利用函数实现数据抽象

➢ 通过函数规约将数据的使用和具体实现方式隔离

```
//Rational是表示分数的结构体类型，假设有以下三个函数
Rational rational(int n, int d);//根据分子和分母返回一个表示分数的变量
int numer(Rational r);          //获取分数变量r的分子
int denom(Rational r);          //获取分数变量r的分母
```

可以使用不同的数据结构（结构体）来实现函数 *问题?*

```
struct Rational {
  int numer; //分子
  int denom; //分母
};
```

```
Rational rational(int n, int d) {
    Rational r{n, d};
    return r;
}
int numer(Rational r) {
    return r.numer;
}
int denom(Rational r) {
    return r.denom;
}
```

```
struct Rational {
  //0 分子，1 分母
  int dimension[2];
};
```

```
Rational rational(int n, int d) {
    Rational r;
    r.dimension[0] = n;
    r.dimension[1] = d;
    return r;
}
int numer(Rational r) {
    return r.dimension[0];
}
int denom(Rational r) {
    return r.dimension[1];
}
```

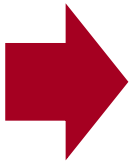
```
struct Rational {
  //高4字节代表分子
  //低4字节代表分母
  long long dim; };
```

```
Rational rational(int n, int d) {
    Rational r{0};
    r.dim |= ((long long)n<<32);
    r.dim |= d;
    return r;
}
int numer(Rational r) {
    return r.dim >> 32;
}
int denom(Rational r) {
    return r.dim << 32 >> 32;
}
```

❑ 类：将数据结构和操作该数据结构的函数“捆绑”在一起，即“封装”

```
struct Rational {
    int numer;
    int denom;
};
```

```
Rational rational(int n, int d) {
    Rational r{n, d};
    return r;
}
int numer(Rational r) {
    return r.numer;
}
int denom(Rational r) {
    return r.denom;
}
```



```
class Rational {
    int num;
    int den;

    void init(int n, int d) {
        num = n; den = d;
    }
    int numer() {
        return num;
    }
    int denom() {
        return den;
    }
};
```

❑ 类的成员（member）

- 成员变量
- 成员函数

说明：使用struct也可以定义类，但与用class定义的类略有区别（后面会讲到）。一般习惯采用class来定义类，这也是很多面向对象语言所采用的。

对象

- 通过类，可以定义变量
  - C++中，类的名字就是用户自定义的类型的名字
  - 可以像使用基本类型那样来使用它

类 `Rational` `r`; 对象

- 通过类定义的变量，称为类的实例，又叫“对象”
- 对象使用 . 运算符可以访问成员变量和成员函数

```
r.init(2, 3);   r.num;   r.denom();
```

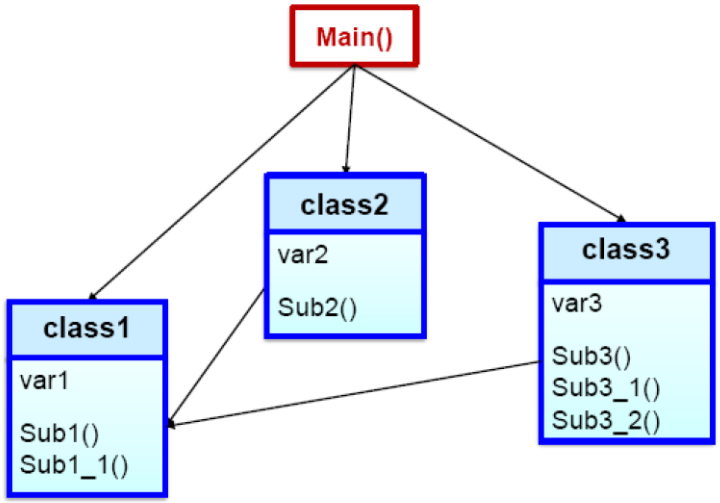
- 使用对象可简化程序的编写
  - 减少全局名字的使用
  - 将有关系的变量和函数合并到一起，使程序结构更清晰

面向对象程序设计

- 对象不仅为编写程序提供了便利，更重要的是提供了一种新的抽象手段来组织大型复杂程序
  - 相比于函数，对象抽象了“值”用于表示对象自身的状态
    - 对象的行为无需依赖全局变量
  - 相比于普通变量，对象抽象了“计算过程”用于表示对象自身所具备的行为
    - 对象之间可以直接交互来完成功能，无需依赖外部函数
- 基本方法
  - 将某类客观事物共同特点（属性）归纳出来，形成一个数据结构（可以用多个变量描述事物的属性）；将这类事物所能进行的行为也归纳出来，形成一组函数；利用类将这些变量和函数捆绑在一起
  - 分析不同类之间的关系，利用对象之间的交互实现功能

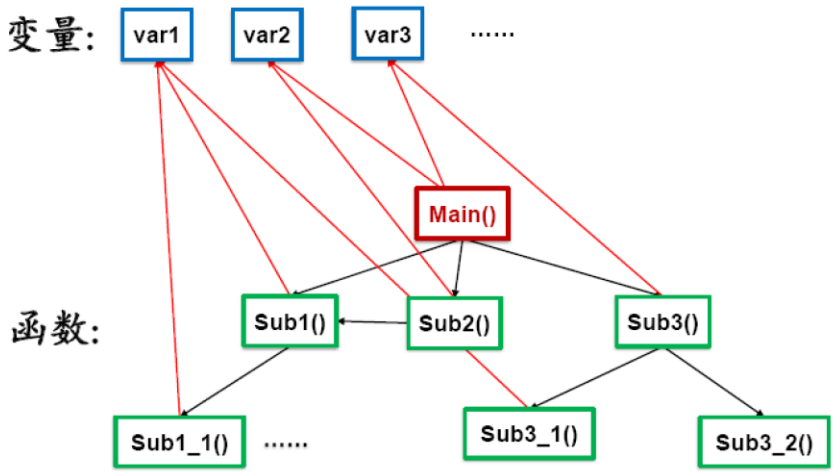
不同的程序设计方法

面向对象程序设计



- 特别适合可以拆分成多个独立部分、不同部分之间相互交互的系统开发
  - 例如，社交网络中不同用户之间的交互，游戏中不同角色的交互

结构化程序设计



- 对输入数据和输出数据之间的变换关系提供了更直观和自然的描述
  - 例如，算法题，数据多步处理流程、业务流程

不同的程序设计方法没有优劣之分，要根据问题特点选取适当的方法

本讲概览

- 用对象进行抽象
- 类的定义和使用
  - 成员变量与成员函数
  - 成员的访问范围
- 对象的生命周期
  - 构造函数
  - 复制构造函数
  - 类型转换构造函数
  - 析构函数

类的定义

- C++中类的定义方法如右图所示
  - 访问范围说明符有public、private、protected三种，无顺序要求、可出现多次
    - 不写则默认为private
  - 成员变量的定义与普通的变量定义相同
  - 成员函数的定义也与普通的函数声明相同
    - 一个类的成员函数之间可以互相调用
    - 成员函数中可以访问成员变量
    - 成员函数可以重载，也可以设定参数的默认值
    - 成员函数的实现可以位于类的定义之外
  - 成员变量和成员函数出现的先后次序没有规定

```
class 类名 {
   访问范围说明符:
     成员变量1
     成员变量2
     ……
     成员函数1
   访问范围说明符:
     成员变量
     成员函数
     ……
   访问范围说明符:
     成员变量
     ……
};
```

❑ 例：输入矩形的长和宽，输出面积和周长

```
class CRectangle {
public:
    int w, h;
    int area() {
        return w * h;
    }
    int perimeter() {
        return 2 * (w + h);
    }
    void init(int w_, int h_) {
        w = w_;
        h = h_;
    }
}; //注意必须有分号
int main() {
    int w, h;
    CRectangle r;
    cin >> w >> h;
    r.init(w, h);
    cout << r.area() <<' '<<
r.perimeter();
    return 0;
}
```

```
class CRectangle {
public:
    int w, h;
    int area(); // 成员函数在此仅声明
    int perimeter();
    void init(int w_, int h_);
};

int CRectangle::area() {
    return w * h;
}
int CRectangle::perimeter() {
    return 2 * (w + h);
}
void CRectangle::init(int w_, int h_) {
    w = w_;
    h = h_;
}
```

在类的定义外实现成员函数
函数名前加 **类名::**

对象的定义

- 通过类可以定义对象

```
CRectangle r;
```

- 每个对象有各自的存储空间用于保存其成员变量
  - 一个对象的某个成员变量被改变了，不会影响到另一个对象

- 成员函数并非每个对象各自存有一份
  - 和普通函数一样在内存中只有一份，但是可以作用于不同的对象

访问对象的成员

❑ 方法1：对象名.成员名

```
CRectangle r1, r2;
r1.w = 5;
r2.init(3,4);
```

init函数作用在r2上，即init函数执行期间访问的w和h是属于r2这个对象的，执行r2.init不会影响到r1

❑ 方法2：指针->成员名

```
CRectangle r1, r2;
CRectangle* p1 = &r1;
CRectangle* p2 = &r2;
p1->w = 5;
p2->init(3,4); //init作用在p2指向的对象上
```

访问对象的成员

- 方法3：引用名.成员名

```
CRectangle r2;
CRectangle &rr = r2;
rr.w= 5;
rr.init(3, 4); //rr的值变了，r2的值也变
```

```
void printRectangle(CRectangle &r) {
    cout<< r.area() << " " << r.perimeter();
}

CRectangle r3;
r3.init(3, 4);
printRectangle(r3);
```

成员的访问范围

- 在类的定义中，用下列访问范围关键字来说明类成员可被访问的范围
  - private: 私有成员，该成员只能在同一个类的成员函数内被访问
  - public: 公有成员，可以在任何地方被访问
  - protected: 保护成员，以后再介绍
  - 如果某个成员前面没有上述关键字，则缺省地被认为是私有成员

```
class Person {
    int nAge;          //私有成员
    char szName[20]; //私有成员
public:
    void setName(const char *name) {
        strcpy(szName, name);
    }
};
```

成员的访问范围

- 在类的成员函数内部，能够访问：
  - 当前对象的全部属性、函数
  - 同类的其它对象的全部属性、函数
- 在类的成员函数以外的地方，只能够访问该类对象的公有成员
- 设置私有成员可实现“信息隐藏”，作用：
  - 强制对成员变量的访问一定通过成员函数进行（规约）
  - 方便修改成员变量的类型等属性->只需更改成员函数（可扩展性）

```
class CEmployee {
private:
    char szName[30];
public:
    int salary;
    void setName(const char *name);
    void getName(char *name);
    void averageSalary(CEmployee e1, CEmployee e2);
};
void CEmployee::setName(const char *name) {
    strcpy(szName, name); // ok
}
void CEmployee::getName(char *name) {
    strcpy(name, szName); // ok
}
void CEmployee::averageSalary(CEmployee e1,
CEmployee e2) {
    cout << e1.szName; // ok，访问同类其他对象私有成员
    salary = (e1.salary + e2.salary) / 2;
}

int main() {
    CEmployee e;
    strcpy(e.szName, "Tom1234567889"); //编译错误
    e.setName("Tom"); // ok
    e.salary = 5000;  // ok
    return 0;
}
```

成员的访问范围

使用struct和class定义类的区别（1/3）

- 使用struct定义的类，如果未说明是公有还是私有的成员，则默认是公有成员

```
class Person {
    int nAge;          //私有成员
    char szName[20]; //私有成员
public:
    void setName(const char *name) {
        strcpy(szName, name);
    }
};
```

```
struct Person {
    int nAge;          //公有成员
    char szName[20]; //公有成员
private:
    void setName(const char *name) {
        strcpy(szName, name);
    }
};
```

本讲概览

- 用对象进行抽象
- 类的定义和使用
  - 成员变量与成员函数
  - 成员的访问范围
- **对象的生命周期**
  - **构造函数**
  - **复制构造函数**
  - **类型转换构造函数**
  - **析构函数**

回顾：变量的生命周期

- 声明和定义→内存分配→初始化→使用→销毁
- 静态内存分配 vs 动态内存分配
  - 通过new为变量动态分配的内存空间必须使用delete进行手动释放，否则会造成内存泄露
- 全局变量、静态变量、局部变量、函数传参、函数返回值
  - 全局变量：未明确初始化时会自动被初始化为全0，在程序结束前被销毁
  - 局部变量：未明确初始化时初始值随机，在函数执行结束后被销毁
  - 函数传参：执行复制初始化，在函数执行结束后被销毁
  - 函数返回值：自动生成临时变量（开启返回值优化 RVO 则不生成），执行复制初始化，返回完成后立即被销毁

对象的初始化

- 下面的“初始化”有什么问题？

```
class CRectangle {
public:
    int w, h;
    int area() {
        return w * h;
    }
    int perimeter() {
        return 2 * (w + h);
    }
    void init(int w_, int h_) {
        w = w_;
        h = h_;
    }
}; //注意必须有分号
int main() {
    int w, h;
    CRectangle r;
    cin >> w >> h;
    r.init(w, h);
    cout << r.area() <<' '<<
r.perimeter();
    return 0;
}
```

- 初始化函数的名称由编码人员自行命名，给类的使用者带来额外学习成本
- 每次定义对象后需要显式调用初始化函数，如果没调用可能会导致错误

构造函数

- 构造函数是一种特殊的成员函数，用于初始化对象
  - 函数名字必须与类名完全相同，可以有参数，但不能有返回值（注意：void也不行！），即

```
类名 (参数列表) {
    一些需要在初始化时执行的内容
}
```

  - 作用：不必专门再写初始化函数，也不必担心忘记调用初始化函数；避免对象未被初始化就使用而导致程序出错
  - 如果定义类时没写构造函数，编译器生成一个默认无参数的构造函数
    - 默认构造函数无参数，不做任何操作
    - 如果定义了构造函数，则编译器不生成默认的无参数的构造函数
  - 对象生成时构造函数自动被调用。对象一旦生成，就再也不能在其上执行构造函数
  - 可以通过函数重载为一个类编写多个构造函数

❑ 示例

```
class Complex {
private:
    double real, imag;
public:
    Complex(double r, double i = 0); //构造函数
};
Complex::Complex(double r, double i) {
    real = r;
    imag = i;
}

Complex c1;                          // error, 没有参数
Complex *pc = new Complex;           // error, 没有参数
Complex c2(2);                       // OK, Complex c2(2,0)
Complex c3(2, 4), c4(3, 5);          // OK
Complex *pc = new Complex(3, 4); // OK
```

构造函数

- 可以通过重载定义多个构造函数

```
class Complex {
private:
    double real;
    double imag;
public:
    void set(double r, double i);
    Complex(double r, double i);
    Complex(double r);
    Complex(Complex c1, Complex c2);
};
Complex::Complex(double r, double i) {
    real = r, imag = i;
}
Complex::Complex(double r) {
    real = r, imag = 0;
}
Complex::Complex(Complex c1, Complex c2) {
    real = c1.real + c2.real, imag = c1.imag + c2.imag;
}

Complex c1(3), c2(1, 0), c3(c1, c2);
```

构造函数

- 构造函数最好是public的，private构造函数不能直接用来初始化对象

```
class CSample{
private:
    CSample() { }
};
int main(){
    CSample Obj; //err. 唯一构造函数是private
}
```

- 思考：如何写一个类，使该类只能有一个对象？
  - 单例模式

复制构造函数

- 如果一个构造函数只有一个参数，这个参数是对同类对象的引用，这样的构造函数称为复制构造函数
  - 形如 X::X(X&) 或 X::X(const X&) 二者选一，后者能以常量对象作为参数
- 如果没有定义复制构造函数，那么编译器生成默认复制构造函数
  - 默认的复制构造函数依次对每个成员都复制初始化
    - 如果是基础数据类型，则直接复制过去；如果是结构体类型，则调用它的复制构造函数
  - 如果定义了自己的复制构造函数，则不会生成默认的复制构造函数
- 不允许有形如 X::X(X) 的构造函数

复制构造函数

```
class Complex {
private:
    double real, imag;
};
Complex c1;      //调用默认的构造函数
Complex c2(c1); //调用默认的复制构造函数，将c2初始化成和c1一样
```

```
class Complex {
public :
    double real, imag;
    Complex(){}
    Complex(const Complex &c) {
        real = c.real;
        imag = c.imag;
        cout << "Copy Constructor called";
    }
};
Complex c1;
Complex c2(c1); //调用自己定义的复制构造函数
                //输出Copy Constructor called
```

复制构造函数

❑ 复制构造函数起作用的三种情况

- ➢ 当用一个对象去初始化同类的另一个对象时

```
Complex c2(c1);
Complex c2 = c1; //此处是初始化语句，不是赋值语句
```

- ➢ 如果函数有一个参数是类的对象，那么该函数被调用时，类的复制构造函数将被调用
  - 潜在的问题是什么？如何解决？

```
void func(Complex c) {}
int main() {
    Complex c1;
    func(c1);
    return 0;
}
```

- ➢ 如果函数的返回值是对象，则函数返回时，类的复制构造函数被调用
  - 经过优化的编译器可能不会生成临时对象

```
A func() {
    A b(4);
    return b;
}
int main() {
    cout << func().v << endl;
    return 0;
}
```

复制构造函数

- 注意：对象间赋值并不导致复制构造函数被调用
  - 赋值并不是初始化

```
class CMyclass {
public:
    int n;
    CMyclass() { }; //构造函数
    CMyclass(const CMyclass &c) { n = 2 * c.n; } //复制构造函数
};
int main() {
    CMyclass c1, c2;
    c1.n = 5;
    c2 = c1;
    CMyclass c3 = c1;
    cout << "c2.n=" << c2.n << ",";
    cout << "c3.n=" << c3.n << endl;
    return 0;
}
```

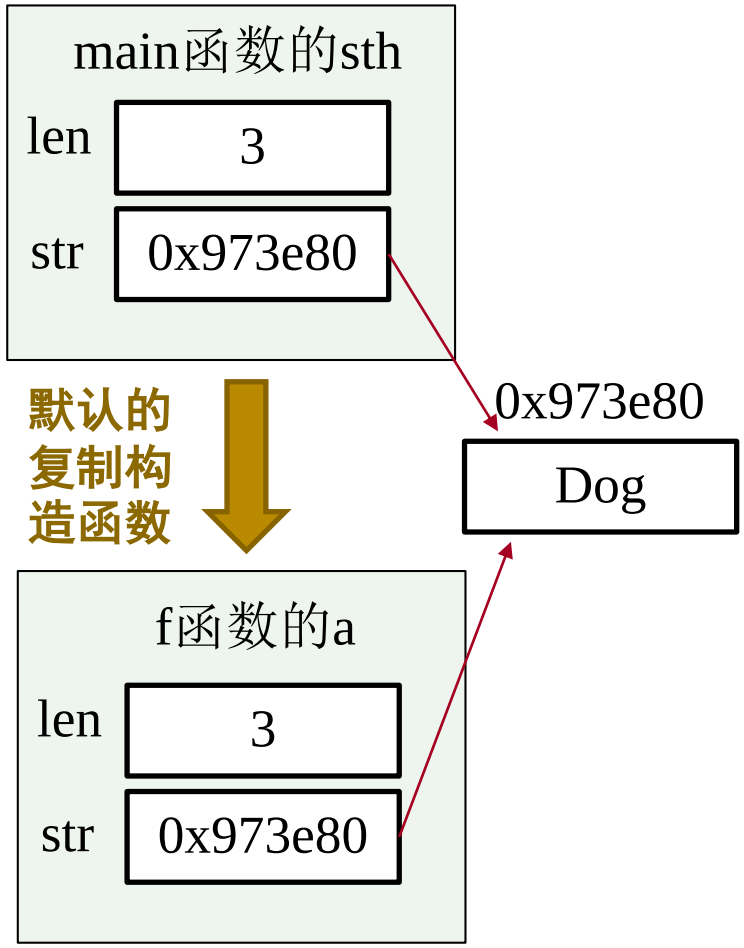
输出结果 c2.n=5, c3.n=10

浅复制与深复制

❑ 下面的程序输出结果是？

```
class String {
public:
    char *str;
    int len;
    String(const char* initVal) {
        len = strlen(initVal);
        str = new char[len + 1];
        for (int i = 0; i <= len; i++)
            str[i] = initVal[i];
    }
};
void f(String a) {
    a.str[0] = 'B';
}
int main() {
    String sth("Dog");
    f(sth);
    cout << sth.str << endl;
}
```

调用默认的复制构造函数



当成员中含有指针时，把指针当成普通变量一样直接复制，称这种复制方法为“浅复制”

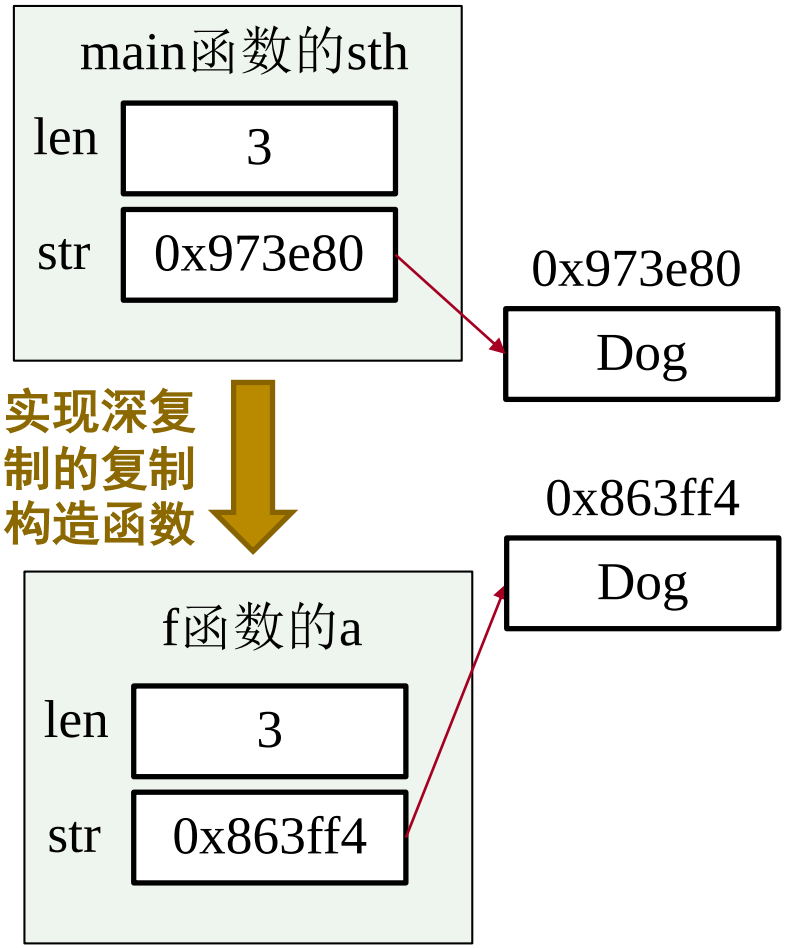
浅复制与深复制

❑ 通过复制构造函数实现深复制

- ➢ 不复制指针的值，而是将指针所指向的内存进行复制，然后让两个指针分别指向这两片不同的内存

```
String(const String& initVal) {
    len = initVal.len;
    str = new char[len + 1];
    for (int i = 0; i <= len; i++)
        str[i] = initVal.str[i];
}
```

```
void f(String a) {
    a.str[0] = 'B';
}
int main() {
    String sth("Dog");
    f(sth);
    cout << sth.str << endl;
}
```



类型转换构造函数

- C++是弱类型语言，在表达式中经常会发生类型转换，主要有两种形式
  - 显式类型转换：在程序中利用类型转换运算符明确表示类型转换，如 类型(表达式) 或 (类型)表达式
  - 隐式类型转换：编译器发现某个类型不适合当前语境，从而自动进行类型转换，如
    - 用 A 类型初始化 B 类型的变量
      - 声明并定义 B 类型变量，其中初始化值是 A 类型的
      - 函数形参期望是 B 类型的，而实参是 A 类型的
      - 函数返回值类型期望是 B，而 return 语句中的表达式是 A 类型的
    - 某个运算符期待 B 类型操作数，但实际给出的操作数却是 A 类型的
    - 在 if while 和 for 的条件表达式中，可以将 A 类型转换到 bool

类型转换构造函数

- 构造函数可以用于实现类型转换
  - 被转换的类型与构造函数的参数类型匹配时会自动调用
  - 编译器会自动调用构造函数，建立一个无名的临时对象

```
class Complex {
public:
    double real, imag;
    Complex(int i) { real = i; imag= 0; }
    Complex(const Complex &c) { real = c.real; imag = c.imag; }
};

int main (){
    Complex c1(7);              //小括号初始化，调用构造函数
    Complex c2 = c1;            //用c1初始化c2，调用复制构造函数
    Complex c3 = (Complex) 9; //显式类型转换，调用构造函数生成临时对象
    Complex c4 = Complex(9);  //显式类型转换，调用构造函数生成临时对象
    Complex c5 = 9;             //隐式类型转换，调用构造函数生成临时对象
    c5 = c1;                 //赋值运算，不调用任何构造函数
    c5 = (Complex) 12; //显式类型转换，调用构造函数生成临时对象，再进行赋值运算
    c5 = Complex(12);  //显式类型转换，调用构造函数生成临时对象，再进行赋值运算
    c5 = 12;                 //隐式类型转换，调用构造函数生成临时对象，再进行赋值运算
    return 0;
}
```

编译器优化，没有调用复制构造函数

类型转换构造函数

- 使用explicit关键字修饰构造函数，可使该构造函数只能用于显式类型转换，不能用于隐式类型转换
  - 可以用于隐式类型转换的构造函数（即没有用explicit修饰的构造函数）又被称为类型转换构造函数

```
class Complex {
public:
    double real, imag;
    explicit Complex(int i) {
        cout<< "IntConstructorcalled" << endl;
        real = i; imag= 0;
    }
    Complex(double r, double i) { real = r; imag= i; }
};
int main (){
    Complex c1(7, 8);
    Complex c2 = Complex(12);  //显式类型转换
    c1 = 9;            //编译错误，没有隐式转换构造函数
    c1 = Complex(9); //ok，可以被显式转换
    cout<< c1.real << "," << c1.imag << endl;
    return 0;
}
```

类型转换构造函数

❑ 对象数组的初始化

```
class CSample{
    int x;
public:
    CSample() {
        cout << "Constructor 1 Called" << endl;
    }
    CSample(int n) {
        x = n;
        cout << "Constructor 2 Called" << endl;
    }
};
int main() {
    CSample array1[2];
    cout << "step1" << endl;
    CSample array2[2] = {4, 5}; //调用转换构造函数
    cout << "step2" << endl;
    CSample array3[2] = {3};    //调用转换构造函数
    cout << "step3" << endl;
    CSample *array4 = new CSample[2];
    delete[] array4;
    return 0;
}
```

输出结果

```
Constructor 1 Called
Constructor 1 Called
step1
Constructor 2 Called
Constructor 2 Called
step2
Constructor 2 Called
Constructor 1 Called
step3
Constructor 1 Called
Constructor 1 Called
```

类型转换构造函数

❑ 对象数组的初始化

```
class Test {
public:
    Test(int n) { }          //(1)
    Test(int n, int m) { }   //(2)
    Test() { }               //(3)
};
```

下面的语句分别用哪些构造函数进行初始化？

编译器优化，均未调用复制构造函数

```
Test array1[3] = {1, Test(1,2)};
//三个元素分别用构造函数(1)、(2)、(3)初始化

Test array2[3] = {Test(2,3), Test(1,2), 1};
//三个元素分别用构造函数(2)、(2)、(1)初始化

Test* p = new Test[2]{1, {1,3}};
//两个元素分别用构造函数(1)、(2)初始化

Test* pArray[3] = {new Test(4), new Test(1,2)};
//两个元素指向的对象分别用构造函数(1)、(2)初始化
```

析构函数

- 一个特殊的成员函数
  - 名字与类名相同，在前面加 ~
    - 没有参数和返回值
    - 一个类最多只有一个析构函数
  - 析构函数在对象消亡时自动被调用
    - 可以定义析构函数来在对象消亡前做善后工作，比如释放分配的空间等
  - 如果定义类时没写析构函数，则编译器生成缺省析构函数
    - 缺省析构函数什么也不做
    - 如果定义了析构函数，则编译器不生成缺省析构函数

```
class String {
private:
    char *p;
public:
    String() {
        p = new char[10];
    }
    ~String() {
        delete[] p;
    }
};
```

析构函数和数组

- 对象数组生命期结束时，对象数组的每个元素的析构函数都会被调用

```
class Ctest{
public:
    ~Ctest() { cout<< "destructor called" << endl; }
};
int main() {
    Ctest array[2];
    cout << "End Main" << endl;
    return 0;
}
```

输出结果

```
End Main
destructor called
destructor called
```

析构函数

❑ 析构函数和delete、delete[ ]运算符

- ➢ delete 和delete[ ]运算导致析构函数调用
- ➢ new运算符对应delete运算符，new[ ]运算符对应delete[ ]运算符，不能错乱地使用
- ➢ 如果使用new[ ]运算符后使用delete运算符，是标准中未定义的行为

```
Ctest* pTest;
pTest= new Ctest;          //构造函数调用
delete pTest;              //析构函数调用
pTest= new Ctest[3];       //构造函数调用3次
delete []pTest;            //析构函数调用3次
```

析构函数

❑ 析构函数在对象作为函数返回值返回后被调用

```
class CMyclass {
public:
    ~CMyclass() {
        cout << "destructor" << endl;
    }
};
CMyclass obj;
CMyclass fun(CMyclass sobj) { //参数对象消亡也会导致析构函数被调用
    return sobj; //函数调用返回时生成临时对象
}
int main() {
    obj = fun(obj); //函数调用的返回值(临时对象) 被用过后
                    //该临时对象的析构函数被调用
    return 0;
}
```

输出结果

```
destructor
destructor
destructor
```

对象的生命周期

```
class Demo {
    int id;
public:
    Demo(int i) {
        id = i;
        cout<<"id="<<id<<" constructed"<<endl;
    }
    ~Demo() {
        cout<<"id="<<id<<" destructed"<<endl;
    }
};
Demo d1(1);
void Func() {
    static Demo d2(2);
    Demo d3(3);
    cout << "func" << endl;
}
int main() {
    Demo d4(4);
    cout << "main" << endl;
    {
        Demo d5(5);
    }
    Func();
    cout << "main ends" << endl;
    return 0;
}
```

输出结果

```
id=1 constructed
id=4 constructed
main
id=5 constructed
id=5 destructed
id=2 constructed
id=3 constructed
func
id=3 destructed
main ends
id=4 destructed
id=2 destructed
id=1 destructed
```

对象的生命周期

```
class CMyclass {
public:
    CMyclass(){};
    CMyclass(const CMyclass &c) {
        cout << "copy constructor" << endl;
    }
    ~CMyclass() {
        cout << "destructor" << endl;
    }
};
void fun(CMyclass obj_) {
    cout << "fun" << endl;
}
CMyclass c;
CMyclass Test() {
    cout << "test" << endl;
    return c;
}
int main() {
    CMyclass c1;
    fun(c1);
    Test();
    return 0;
}
```

输出结果

copy constructor
fun
destructor //参数消亡
test
copy constructor
destructor // 返回值临时对象消亡
destructor // 局部c1变量消亡
destructor // 全局变c量消亡

本讲小结

- 正确性/鲁棒性
  - 成员的访问范围可以实现信息隐藏，只将最少的信息暴露出去，从而可避免非预期的访问
  - 利用构造函数可以实现对象的强制初始化，避免非预期初始值
  - 利用析构函数可以实现对象销毁前的处理，避免内存泄漏等问题
- 可扩展性
  - 数据使用和实现的分离使得在不影响数据使用的前提下改变数据的实现方式
  - 对象将复杂系统分成独立地部分，不同部分可单独改变而不相互影响
  - 信息隐藏使得对象的变更可以尽量少地影响程序的其他部分
- 可复用性
  - 类实现了对相同的属性和操作的封装，通过类的实例化得到对象，实现复用
- 可理解性
  - 对象更加贴近现实世界的情况，使得复杂系统的结构更加清晰
  - 利用封装可以减少全局名字的使用

谢谢

欢迎在课程群里填写问卷反馈