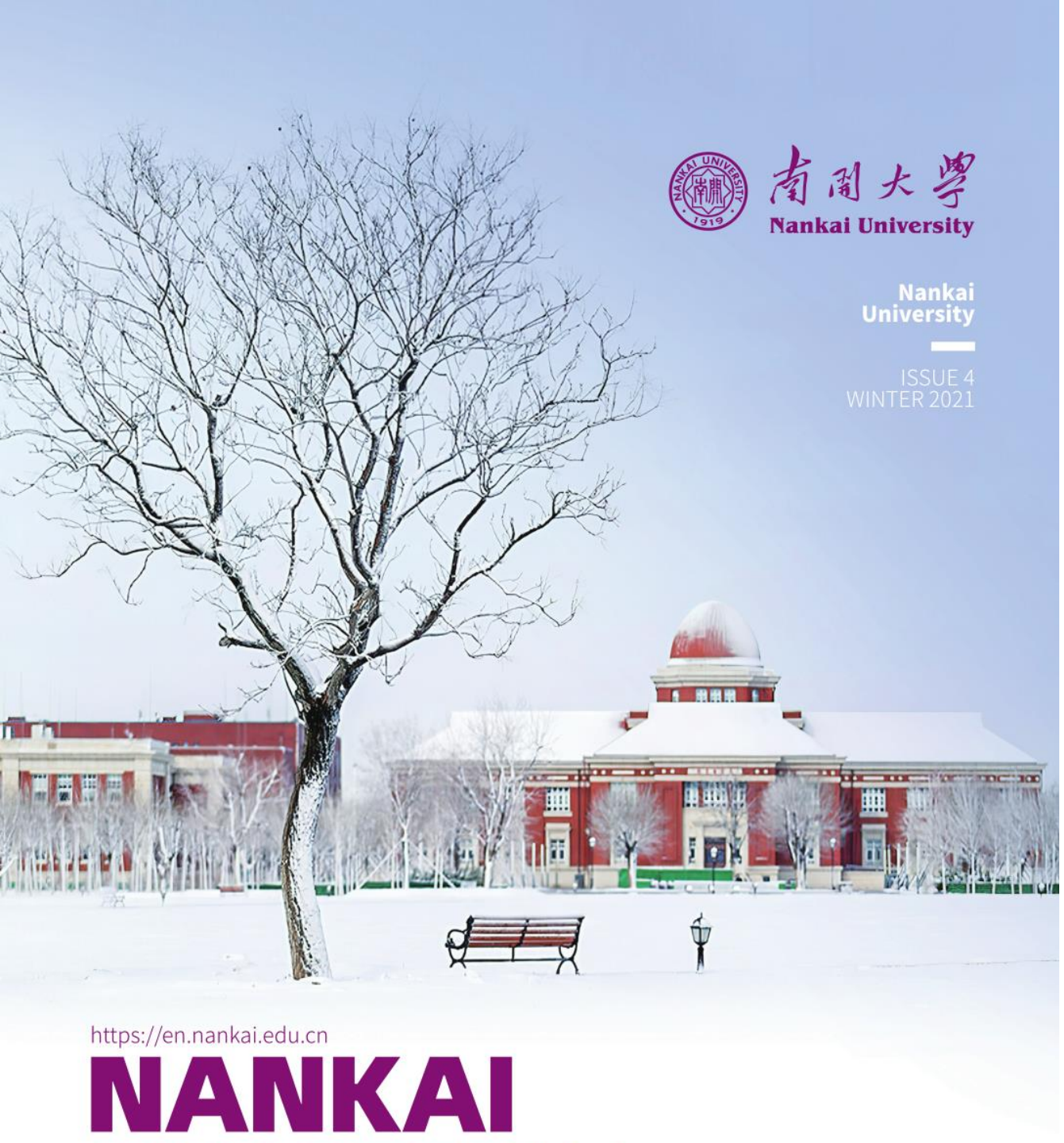

03-构造与析构函数

龚成

cheng-gong@nankai.edu.cn

构造函数

类的实例化

初始化与构造函数

空构造函数

拷贝构造函数

赋值构造函数

重载构造函数

禁用构造函数

类的实例化

- 类的实例化指的是从类说明对象的过程。
- 按如下方式来说明对象（即该类的变量或称该类的实例）：

  <自定义类类型名> <对象名 1>, … ,<对象名 *n*>;

- 若在该类的说明中含有带参数的构造函数，则在说明对象的同时，要给出具体实参初始化该对象（参看 7.3 一节内容）。此时说明对象的方式应改为：

  <自定义类类型名> <对象名 1> (<实参表 1>), … , <对象名 *n*> (<实参表 *n*>);

```
class Person{
public:
    std::string name;
    int age;
    Person(){}
    Person(std::string name, int age){
        this->name = name;
        this->age = age;
    }
};
```

```
int main(){
    Person p1, p2("zhansan",23),p3;
    return 0;
}
```

类的实例化

- 与其他变量及结构变量类似，也可以**说明类对象的数组**（数组分量为对象）以及**指向对象的指针**。另外，对象还可进行如下的操作与使用：
  - **同类型的对象间可以相互赋值**；
  - **对象可作为函数参数**（如，对象作形参，对象指针作函数参数等）；
  - **函数的返回值可以是对象**（或指向对象的指针）；
  - 可以在**一个类中说明其他类的对象作为其数据成员**等。

初始化与构造函数

- **公有数据成员初始化：**如果一个类的数据成员是公有的，那么其对象的初始化与一般变量、结构变量或变量数组的初始化没有什么区别。

```
class address{
    public:
        long  telenum;
        char  addr[30];
};

class person{
    public:
        char  name[15];
        int  age;
        address  paddr;
};

person p1={"Zhang Hua",23,{2475096,"NanKai University"}};
```

初始化与构造函数

- **公有的初始化函数**：在类中设置公有的初始化函数完成此项任务，比如在类 point 中，可以专门设计一个初始化函数：

```
class point {
        private:
            float xcoord,ycoord;
        public:
            void initpoint( ){
                xcoord=0; ycoord=0;
        };
```

初始化与构造函数

- 大多数类的数据成员是**私有的或保护的**，不能采用这种方式。
- **构造函数进行对象初始化**：对象也被称为类变量，一个类的对象是这个类的一个实例。和变量一样，它也可以为其数据成员赋初值。不过对象的初始化情况比较复杂，可以有下列多种不同的方式，其中最重要的方式是构造函数。

初始化与构造函数

- 为了更好地解决对象的初始化问题，C++规定在类的说明中可以包含一个或多个特殊的**公有函数成员——构造函数**。构造函数具有下列特征：
  - 函数名与类名相同。
  - 无函数（返回）类型说明。
  - 构造函数在一个新的对象被建立时，该对象所隶属类的构造函数自动地被调用，对这个对象完成初始化工作。
  - 在上一条中提到的新对象的建立包括两种情况：在对象说明语句中，用 new 函数建立新的动态对象时。
  - 如果一个类说明中没有给出显式的构造函数，系统将自动给出一个缺省的（隐式的）构造函数：<类名>(void){}
  - 如果说明中包括多个构造函数，一般它们有不同的参数表和函数体。系统在（自动）调用构造函数时按照一般函数重载的规则选择其中之一。

初始化与构造函数

- **举例：** 在 class point 中添加显式的构造函数：
- 前一个**无参的构造函数**把每个新对象一个平面上的点初始化为（0.0,0.0）
- 第2个构造函数则为新的对象点赋予由用户指定的初值。

```
class point {
    private:
        float xcoord,ycoord;
    public:
        point(void){
            xcoord=0.0;
            ycoord=0.0;
        }
        point(float ix,float iy){
            xcoord=ix;
            ycoord=iy;
        }
```

初始化与构造函数

- 如果这两个构造函数同时存在，程序员在说明新的对象时可任意选择：

```
point p(2.0,1.5), q;
```

- 上面的对象说明语句建立了两个对象 p 和 q，对于点 p，用参数 2.0 和 1.5通过第 2 个构造函数进行初始化，而点 q 没有提供初始化的值，它将自动选择第 1 个构造函数，把 xcoord、ycoord 初始化为 0.0。

```
class point {
    private:
        float xcoord,ycoord;
    public:
        point(void){
            xcoord=0.0;
            ycoord=0.0;
        }
        point(float ix,float iy){
            xcoord=ix;
            ycoord=iy;
        }
```

空构造函数

- 空构造函数是默认的构造函数，即使什么构造函数都不写也会自动补全。
- 空构造函数默认没有参数，什么也不干（类似例子中的Person()）。
- 一旦自定义任意的构造函数，则空构造函数失效。即任意定义的构造函数会导致失去空构造函数，若想使用必须重新显式定义空的构造函数。

```
class Person{
public:
	std::string name;
	int age;
	Person(){}
	Person(std::string name, int age){
		this->name = name;
		this->age = age;
	}
};
```

拷贝构造函数

- 拷贝构造函数是一种特殊的构造函数，具有一般构造函数的所有特性。拷贝构造函数只含有一个形参，而且其形参为本类对象的引用。拷贝构造函数的原型为：“<类名>（<类名>& ）;”。
- 拷贝构造函数的作用是使用一个已存在的对象去初始化另一个正在创建的新对象。
- 若用户没给出显式的拷贝构造函数，则系统会自动生成一个缺省的拷贝构造函数，它只进行对象间的“原样拷贝”（即位对位的拷贝，也称为“浅拷贝”）。
- 某些特殊情况下，用户必须在类定义中给出显式的拷贝构造函数以实现用户指定的“深拷贝”功能。

拷贝构造函数

- 在下述 3 种情况下，系统都将自动地去调用对象所属类的拷贝构造函数：
  - **当用已存在的对象来创建一个相同的新对象时：**

    〈类名〉〈对象名 2〉（〈对象名 1〉）；

    即在说明新对象〈对象名 2〉时，准备用已存在对象〈对象名 1〉来对其进行初始化。
  - **若对象作为函数的赋值参数，**在调用函数时，当刚进入被调函数处首先要进行实参和形参的结合，此时会**自动调用拷贝构造函数，以完成由实参对象来创建出一个相同的（局部于本函数的）**形参新对象。
  - **若函数的返回值是类的对象，**在执行被调函数的返回语句后（也即在函数调用完成返回时），系统也将**自动调用拷贝构造函数去创建一个与返回值相同的**临时新对象。
- 一般规定所创建的临时对象，**仅在创建它们的外部表达式范围内有效，表达式结束时，系统将调用析构函数去“销毁”该临时对象。**

拷贝构造函数

- 调用拷贝构造函数的程序“构架”示例如下：

```
class point {                          //自定义类 point
        int x,y;
    public:
        point(point& pt) {             //拷贝构造函数
            cout<<"Enter copy-constructor!"<<endl;
            …
        }
        …
};
void main() {
        point pt1(123,456);            //说明对象 pt1，将调用普通的两参构造函数
        point pt2=pt1;                 //与使用“point pt2(pt1);”效果相同（情况 1）
        func1(pt2);                    //情况 2，调用 func1 进行实参对象 pt2 与形参 pt 结合时
        pt2=func2();                   //情况 3，当调用 func2 结束返回一个对象时
}
void func1(point pt) {                 //point 型的类对象作为函数的赋值参数（情况 2）
        …
}
point func2() {                        //函数返回结果为 point 型的类对象（情况 3）
        point p1(606,808);
        …
        return p1;                     //返回 point 型的类对象
}
```

拷贝构造函数

- 在某些情况下，必须在类定义中给出显式的拷贝构造函数:
  - 假设在某类的普通构造函数中分配并使用了某些系统资源（例如通过 new 分配并使用了系统的堆空间），而且在该类的析构函数中释放了这些资源。
  - 如果用户没有对该类提供显式拷贝构造函数，则会出现两个对象拥有同一个资源即拥有同一块系统堆空间的情况。当对象析构时（两个对象各要被析构一次），则会遇到同一资源被释放两次（两次 delete 释放同一堆空间）的错误。

```
class person {
        char * pName;
  public:
        person (char *  pN){                    //普通构造函数
                pName=new char[strlen(pN)+1];   //动态申请了系统资源（堆空间）
                strcpy(pName,pN);               //将实参 pN 串拷贝到新分配的动态空间中
        }
        ~person (){                             //析构函数
                delete  pName;                  //释放系统资源
        }
};
```

```
person (person & p) {                                   //拷贝构造函数
        cout<<"Copying "<<p.pName<<" into its owm block"<<endl;
        pName=new char[strlen(p.pName)+1];              //动态申请自己的 pName 空间
        strcpy(pName, p.pName);                         //向自己的 pName 空间拷入内容
}
```

赋值构造函数

- 赋值构造函数是一种特殊的构造函数，具有一般构造函数的所有特性。
- **赋值构造函数是重载赋值运算符=来实现类对象之间的赋值初始化，因此赋值构造函数的原型是确定的，形参为类对象的引用：**
- **赋值构造函数的原型为：“<类名>& operator=(<类名>&);”。**

```
class Person{
public:
        std::string name;
        int age;
        Person(){}
        Person& operator=(Person& p){
                this->name = "hello";
                this->age = p.age;
                return *this; }
        Person(std::string name, int age){
                this->name = name;
                this->age = age; }
};
```

```
int main(){
        Person p1,p2("zhansan",23),p3;
        p1=p2;
        std::cout<<p1.name<<", "
                        <<p2.name<<std::endl;
        return 0;
}
```

```
hello, zhansan
```

赋值构造函数

- 在进行类对象之间的赋值时，会自动调用赋值构造函数进行对象间赋值。
- 与拷贝构造函数类似，**若用户没给出显式的赋值构造函数，则系统会自动生成一个缺省的赋值构造函数，它只进行对象间的“浅拷贝”**。

```
class Person{
public:
    std::string name;
    int age;
    Person(){}
    Person(std::string name, int age){
        this->name = name;
        this->age = age;
    }
};
```

```
int main(){
    Person p1,p2("zhansan",23),p3;
    p1=p2;
    std::cout<<p1.name<<", "
                <<p2.name<<std::endl;
    return 0;
}
```

```
zhansan, zhansan
```

重载构造函数

- 由于构造函数都是重载函数，其允许不同类型的形参列表，因此无论是拷贝构造函数还是赋值构造函数，或者特殊的类型转换符重载函数，其形参列表为本类对象时被默认调用，同时也允许不同类对象作为形参的形式：

```
Person(){}
Person& operator=(Person& p){
        this->name = p.name;
        this->age = p.age;
        return *this;
}
Person& operator=(Car& p){
        this->name = "hello";
        this->age = 10;
        return *this;
}
```

```
Person(std::string name, int age){
        this->name = name;
        this->age = age;
}
Person(Person& p){
        this->name = p.name;
        this->age = p.age;
}
Person(Car& c){
        this->name = "hello";
        this->age = 10;
}
```

禁用构造函数

- C++中使用 delete 关键字来禁用构造函数，从而阻止特定的构造函数被调用，以防止对象的拷贝、移动或默认构造等。
- 任何尝试使用被禁用的构造函数的代码都会导致编译错误。这是一种在**编译时**防止特定构造函数被调用的有效方式。

```
class Person{
public:
    std::string name;
    int age;
    // 禁用默认构造函数
    Person() = delete;
    // 禁用拷贝构造函数
    Person(Person&) = delete;
    // 禁用赋值构造函数
    Person& operator=(Person&) = delete;
}
```

析构函数

析构函数原型

析构过程

析构函数原型

- 与构造函数在对象生成时自动执行相对应，析构函数专门用来在对象的生存期结束时使用。它也是类的特殊函数成员，其特殊性质包括：
  - 析构函数名一律为～〈类名〉，如～point。
  - 无函数返回类型。
  - 无参数。
  - 一个类只可有一个析构函数，也可以缺省。
  - 在对象注销时，包括用 delete 函数释放动态对象时，系统自动调用析构函数。
  - 若某个类定义中没有给出显式的析构函数，则系统自动给出一个缺省的（隐式的）
  - 如下形式的析构函数：~<类名>(void){};
- 析构函数不可重载，因为其函数名和参数表都是确定的。

```
class Person{
public:
    ~Person(){ std::cout<<"Person destructor"<<std::endl; }
}
```

析构过程

- 构造函数和析构函数是类的要素，它们在一个对象的生存期的开始阶段和结束阶段起着举足轻重的作用。**承担着对象的初始化和收尾工作**。
- 形象地说，每当对象“**诞生**”时（通过对象说明语句或通过new 来建立或生成新对象时），系统都将自动调用对象所属类的构造函数，以完成对象的初始化工作；
- 而每当对象“**死亡**”前（当对象退出其说明区域即作用域或使用 delete 释放动态对象时），系统都将自动调用对象所属类的析构函数，以完成对象撤销前的工作。

```
class Person{
public:
      ~Person(){
        std::cout<<"Person destructor"<<std::endl;
      }
};
```

```
int main(){
      Person p1,p2,p3;
      return 0;
}
```

析构过程

- 注意：在说明类对象指针时，不涉及类的实例化，因此不存在类对象的构造和析构过程。
- 只有在使用new和delete进行类对象的动态生成和释放时，才会出发类的构造和析构函数

```
class Person{
public:
        ~Person(){
          std::cout<<"Person destructor"<<std::endl;
        }
};
```

```
int main(){
        Person p1,p2,p3;
        Person* ptr1, ptr2;
        ptr2 = new Person();
        delete ptr2;
        return 0;
}
```