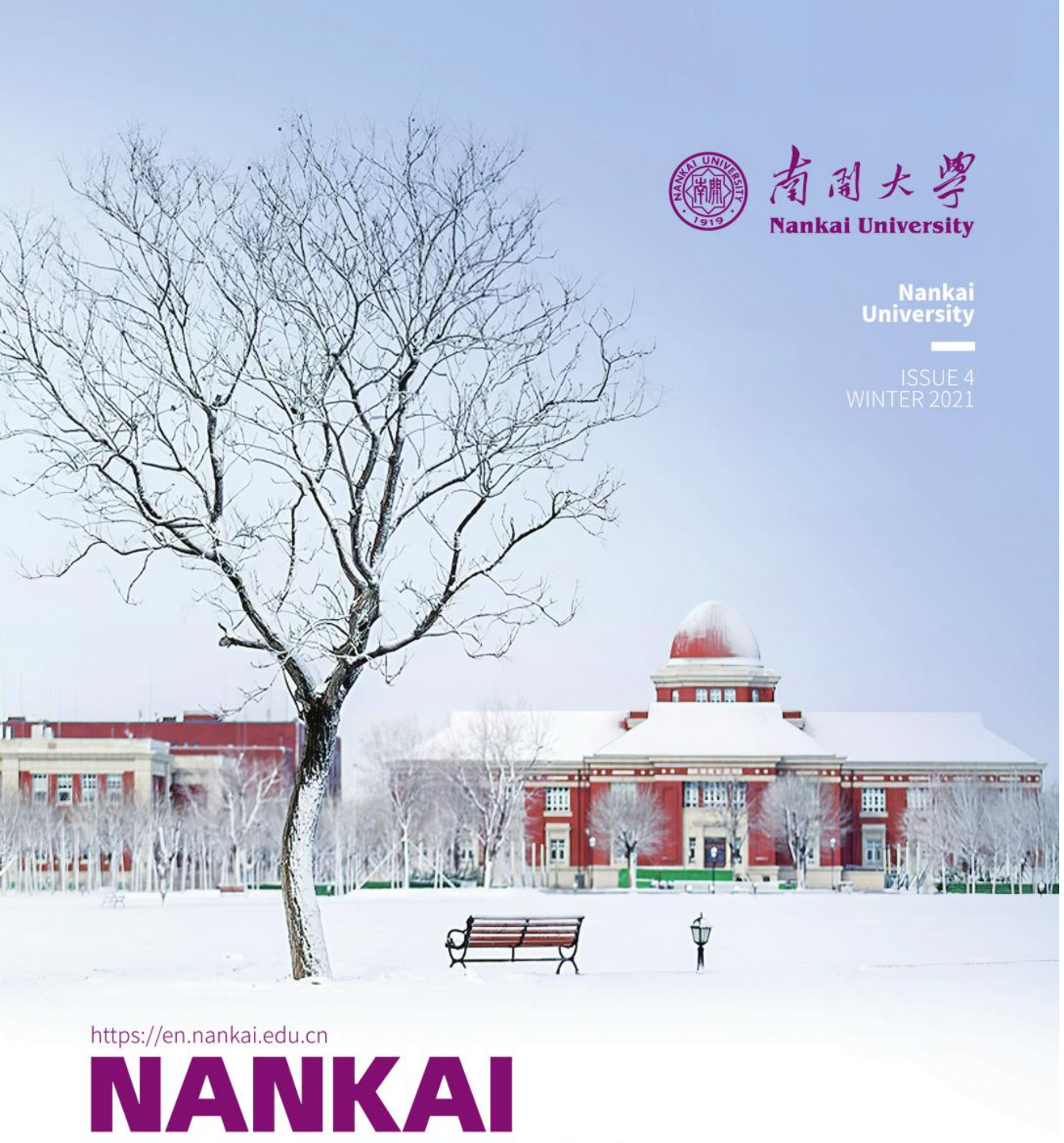

引用和复制构造函数

龚成

cheng-gong@nankai.edu.cn

引用

- 引用不仅像数组和指针那样依赖于已有的类型，而且它还依赖于一个已有的变量。从直观上说，一个引用变量是一个已定义的变量的别名。
- 引用变量的说明格式与指针变量说明相似：

  〈类型名〉&〈变量名〉=〈对象变量名〉；
- 与指针说明的区别是：
  - 用符号“&”代替符号“*”。
  - **赋初值部分不可缺省。（为什么？）**

```
int size=5，color;
int  &  refs=size;
int  &  refc=color;
```

引用与指针

- C++语言中增加引用这种数据形式，主要是用于在一定的范围内，代替或改进指针的作用。虽然在说明方式上十分相似，但在概念上却有着明显的不同。其区别最主要有两点：
  - 指针表示的是一个对象变量的地址，而引用则表示一个对象变量的别名。因此在程序中表示其对象变量时，前者要通过取内容运算符*，而后者可直接代表。
  - 指针是可变的，它可以指向变量 m，也可以指向变量 n，而引用变量只能在定义时一次确定，不可改变。
- 引用类型变量与其他类型变量不同，**它没有自己的值和地址空间**，只是作为另一个变量的别名，在它的生存期期间两个名字绑定在一起，因此，引用类型的使用是有限制的：
  - 引用类型变量不能被引用。
  - 引用类型不能组成数组。
  - 引用类型不能定义指针。

引用与指针

- 在底层，引用就是一个常量指针。考虑以下的C++代码：

```
int x = 10;
int& ref = x;
```

- 在底层，ref实际上是一个指向x的常量指针，我们不能改变ref指向的对象，这就是为什么引用必须在创建时被初始化，并且不能被重新赋值。
- 与常量指针不同的是，你不能直接获取引用的地址，也不能进行指针运算。这使得引用在使用上更安全，因为你不能误用它来访问无效的内存。
- 引用只是一个已存在对象的别名，它并不存储任何数据，因此理论上引用本身不会占用额外的内存空间。在底层，引用是通过指针来实现的。当我们创建一个引用时，编译器会在内存中创建一个常量指针，这个指针指向被引用的对象，这个指针会占用一定的内存空间。

引用作为函数参数

- 引用型参数在函数被调用时，**相应的实参必须是对应类型的变量或对象**；**在调用函数体运行前，生成该实参的引用变量**；在整个函数体运行过程中，这个**引用变量相当于作为实参的变量或对象的别名**，直到函数调用结束返回。优点是：
  - 可以把函数外的变量以别名的形式引入到函数体内参加运算，非常方便，这种方式比用指针解决这个问题**更合理**。
  - 不必在调用时创建与实参变量或对象对应的值的参数变量，当实参变量或对象占用内存较多时，可以**节省内存**。
  - 用指针也可以实现类似于引用调用的效果，但由于指针可以改变内容，任意赋值，因此它不如引用型**参数安全**。

引用作为函数返回值

- 当把函数的返回类型说明为引用型时，这个函数返回的不仅仅是某一变量或对象的值，而且返回了它的“别名”，该函数的调用也可以被赋值。例如：

```
int & maxr (int & m, int & n) {
 if (m>n) return m;
 return n; }
maxr (a, b)= 10;
```

- 函数的调用本身也可以作为变量和对象来使用。

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
```