Assignment 8

一、概念题

1. 简述多继承的含义；在多继承中，什么情况下会出现二义性？C++是怎样消除二义性的？（请举一个简单例子配合说明）

多继承是指派生类可以有两个或者两个以上的直接基类。

命名冲突：当多个基类中包含同名的成员时，他们的派生类中就会出现命名冲突。

比方说类A存在Print()，B存在Print()，C同时继承了A和B，调用Print函数就会出现冲突，解决方式是类用基类名受限，使用A::Print()来访问。

重复继承问题，如果直接基类有公共的基类，则会出现重复继承，这样公共基类成员在多继承的派生类中就有多个拷贝。

比如存在类A存在成员x，类B，C，D满足：B和C继承A，D继承B和C。这样D就有两个x成员B::x和C::x，C++通过虚基类来解决，最终D只有一个成员x。

2. 继承和组合相较彼此有什么优缺点？你觉得它们各自适用于什么样的场景?

继承：子类可以自动获得父类的接口，调用子类时不需要纠结方法命名，可以实现多态，代码显得更简洁。但是继承层次过深、过复杂，也会影响到代码的可维护性。

组合：不破坏封装，整体类与局部类松耦合，彼此相对独立。若是继承，子类必须继承父类方法，限制较大；父类的改动会影响子类；具有较好的扩展性。继承制在成员较多时可能形成多代层次，复杂且易出错，组合则没有这个缺点；支持动态组合，整体对象可以选择不同的局部对象。因为类与类之间相对独立，组合也会比继承更加灵活，修改更加容易；整体类可以对局部类包装形成新的接口；调用新方法不再受到继承中命名等诸多限制；代码复用能力更强，可以避免继承与封装的矛盾。

当类之间继承结构比较稳定时，并且层次比较浅，关系不复杂，利用继承确实比较方便。其他场景能使用组合就是用组合。

言之有理即可。

二、编程题

1. 在以下调用中，给出类的构造和析构顺序并解释原因。

```
//
// Created by Will on 2021/6/8.
//
#include <stdio.h>
class Object {
 public:
  Object(){
    printf("Construct Object\n");
```

```cpp
  }
  ~Object(){
    printf("Destruct Object\n");
  }
};
class Base : public Object {
 public:
  Base(){
    printf("Construct Base\n");
  }
  ~Base(){
    printf("Destruct Base\n");
  }
};
class Derived1 : virtual public Base {
 public:
  Derived1(){
    printf("Construct Derived1\n");
  }
  ~Derived1(){
    printf("Destruct Derived1\n");
  }
};
class Derived2 : virtual public Base {
 public:
  Derived2(){
    printf("Construct Derived2\n");
  }
  ~Derived2(){
    printf("Destruct Derived2\n");
  }
 private:
  Object o;
};
class Mid : public Derived1, public Derived2 {
 public:
  Mid(){
    printf("Construct Mid\n");
  }
  ~Mid(){
    printf("Destruct Mid\n");
  }
};
class Final : public Mid, public Object, public Derived1 {
 public:
  Final(){
    printf("Construct Final\n");
  }
  ~Final(){
    printf("Destruct Final\n");
  }
 private:
  Derived2 d2;
};
int main() {
  {
    Final f;
  }
  return 0;
}
```

构造顺序：基类>成员>自身，同时虚基类只构造一次。

```
Construct Object
Construct Base
Construct Derived1
Construct Object
Construct Derived2
Construct Mid
Construct Object
Construct Derived1
Construct Object
Construct Base
Construct Object
Construct Derived2
Construct Final
```

析构顺序：与构造顺序相反。

```
Destruct Final
Destruct Derived2
Destruct Object
Destruct Base
Destruct Object
Destruct Derived1
Destruct Object
Destruct Mid
Destruct Derived2
Destruct Object
Destruct Derived1
Destruct Base
Destruct Object
```

2. 仿照课堂上的例子，使用通用指针实现归并排序算法，可以对double数组进行排序。

```
#include <stdio.h>
#include <string.h>
#include <cstdlib>
int double_compare(const void *p1, const void *p2) {
  if (*(double *) p1 > *(double *) p2)
    return 1;
  return -1;
}

/*
通用归并排序算法（从小到大）
参数:
        base: 需要排序的数据内存首地址。
       count: 数据元素个数
element_size: 一个数据元素所占内存大小
         cmp: 比较两个元素的函数
*/
char *tmp;
void merge_sort(void *base, unsigned int count, unsigned int element_size,
                int (*cmp)(const void *, const void *)) {
  unsigned int mid = count / 2;
  int left = 0, right = 0, result_index = 0;
  char *el, *er;

  // one element
  if (count <= 1)
    return;

  if (count == 2) {
    el = (char *) base;
    er = (char *) base + element_size;
    if (cmp(el, er) > 0) {
      // tmp[0] = el, el = er, er = tmp[0]
      memcpy(tmp, el, element_size * sizeof(char));
      memcpy(el, er, element_size * sizeof(char));
      memcpy(er, tmp, element_size * sizeof(char));
    }
    return;
  }

  merge_sort(base, mid, element_size, cmp);
  merge_sort((char *) base + element_size * mid, count - mid, element_size, cmp);

  // merge

```

```c
  while (left < mid && right < count - mid) {
    el = (char *) base + left * element_size;
    er = (char *) base + element_size * (mid + right);
    if (cmp(el, er) < 0) {
      memcpy(tmp + result_index * element_size, el, element_size * sizeof(char));
      result_index++;
      left++;
    } else {
      memcpy(tmp + result_index * element_size, er, element_size * sizeof(char));
      result_index++;
      right++;
    }
  }

  while (left < mid) {
    el = (char *) base + left * element_size;
    memcpy(tmp + result_index * element_size, el, element_size * sizeof(char));
    result_index++;
    left++;
  }
  while (right < count - mid) {
    er = (char *) base + element_size * (mid + right);
    memcpy(tmp + result_index * element_size, er, element_size * sizeof(char));
    result_index++;
    right++;
  }

  // copy
  memcpy((char *) base, (char *) tmp, count * element_size * sizeof(char));
}

int main() {
  double array[] = {1.0, 2.1, 9.4, 0.3, 0.6, 5.4, 0.01, 5.41, -0.1, -0.3, 0.5,5.8, 6.3, -8.4,4.9, 7.2};
  tmp = (char *) malloc(16 * sizeof(double) * sizeof(char));
  merge_sort(array, 16, sizeof(double), double_compare);
  free(tmp);
  for (double a : array) {
    printf("%f\n", a);
  }
  return 0;
}
```