面向对象程序设计

第一章(选择 填空)

- 面向对象三大特征：继承，封装，多态
- 函数重载 : **同一个函数名**可以对应**多个**函数实现
- **带默认参数的函数**
  - 举例

```
class A{
    A(int a = 0) {
        cout << a << endl;
    }
}

int fun(int i = 0) {
    cout << i << endl;
}

int main() {
    fun(); // 这样是可以的因为规定如果没有传参就用默认值 所以输出0
    fun(1); // 输出1

    // 这里调用的就是上面的构造函数输出0
    // 这一句其实可以写成A a();联系上面的fun()示例
    A a;
}
```

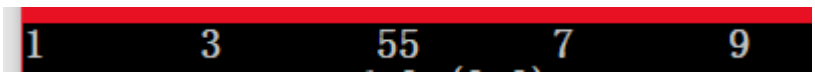
  - 默认参数一定是从右向左进行规定
    见：练习题2选择8
  - 保证在函数调用之前给出默认值
    并且声明与实现中**只能有一个位置**给出默认参数值

- 二义性问题，可能默认参数中的数据类型有相似之处，编译器不知道选择哪个函数

```
int add(int x=5, int y=6) {return x+y; }
float add(int x=5, float y=10.0) { return x+y; }
int main() {
    int a; float b;
    a= add(10,20); // 产生二义性
    b= add(10);
    return 0;
}
```

- **引用(可能会有代码结果填空)**
  - 例题

```
#include <iostream>
using namespace std;
int& f(int index,int a[]){
    int &r=a[index];
    return r;
}
int main(){
    int a[]={1,3,5,7,9};
    f(2,a)=55;
    for(int i=0;i<5;i++)
        cout<<a[i]<<"\t";
    return 0;
}
```



- 用途：可以作为函数参数的返回值
- **内联函数(inline)(选择填空)**
  - 消除了宏定义的不安全性，直接进行函数行为的替换
  - 类内实现的函数会尽可能自动设定为内联函数
- **::作用域运算符**

- ```
  #include <iostream>
  float a = 13.5;
  int main(){
      int a = 5;
      std::cout<<a<<std::endl;        // 输出局部变量
      std::cout<<::a<<std::endl;     //输出全局变量
      return 0;
  }
  ```
  注意： "::"不能访问函数中的局部变量。

- ## new 与 delete(了解)

  - 动态创建数组：

    new <类型说明符> [<算术表达式>]

    示例：int *p2 = new int[5];

    销毁堆对象

    delete <指针名>

    示例：delete p1;

    销毁动态创建的数组

    delete [] <指针名>

    示例：delete [] p2;

    注意：delete运算符后面的中括号中不能写如何数据。

第二章(选择 填空 代码结果填空)

- ## 类的访问权限(代码结果填空)

```
/*
定义一个长方形类CRect，其数据成员包括颜色，左上角坐标，长和宽，其成员函数包括改变矩形的颜色(SetColor)和大小（SetSize），移动矩形到新的位置（Move），绘出矩形（Draw）
*/

#include <iostream>
```
```

```cpp
#include <string>
using namespace std;
class CRect{
private:
    string color; int left; int top; int length; int width;
public:
    void SetColor(string c); void SetSize(int l, int w);
    void Move(int t,int l);  void Draw();
};
void CRect::SetColor(string c){color=  c;}
void CRect::SetSize(int l, int w){
    length=l;  width = w;
}
void CRect::Move(int t,int l){
    top = t;   left = l;
}
void CRect::Draw(){
   cout << " (" << left << "," << top << ")" << endl;
   cout<< length << "," << width << endl;
   cout<< color << endl;
}
int main(){
    CRect r;
    r.SetColor("Red");
    r.Move(10,20);
    r.SetSize(100,200);
    r.Draw();
    r.Move(50,50);
    r.SetColor("Blue");
    r.Draw();
    return 0;
}

/*
结果
(20, 10)
100, 200
Red
(50, 50)
100, 200
```

```
Blue
*/
```

- **strust与class的区别**是：如果不指定访问权限，前者缺省的访问权限是公有的，而后者是私有的。

- **构造函数 析构函数(选择 填空 代码结果填空)**
  - 看模拟题1，2差不多了
- **拷贝构造函数(代码结果填空)**
  - 形式： 类名(const 类名& 引用名)
  - 每个类如果没有说明都会有默认拷贝构造函数
  - 下面这个例题最好记一下答案(因为解释起来有点麻烦。。。想听的私聊。。。)(ppt上还有一个例题最好也看一下)

```
class TPoint{
public:
    TPoint(int x,int y) {X=x;Y=y;}
    TPoint(TPoint &p);
    ~TPoint() {cout<<"Destructor called."<<endl;}
    int Xcoord() {return X;}
    int Ycoord() {return Y;}
private:
    int X,Y;
};
TPoint::TPoint(TPoint &p){
    X=p.X; Y=p.Y;
    cout<<"Copy_initialization Constructor called.\n";
}
TPoint fun(TPoint Q){
    cout<<"OK! "<<endl;
    int x,y;
    x=Q.Xcoord()+10;  y=Q.Ycoord()+15;
    TPoint R(x,y);
    return R;
}
int  main(){
    TPoint M(12,20),P(0,0),S(0,0);
    TPoint N(M); P=fun(N);     S=M;
```

```
    cout<<"P="<<P.Xcoord()<<", " <<P.Ycoord()<<endl;
    cout<<"S="<<S.Xcoord()<<", " <<S.Ycoord()<<endl;
    return 0;
}
/*
    答案:
   Copy_initialization Constructor called.
   Copy_initialization Constructor called.
   OK!
   Copy_initialization Constructor called.
   Destructor called.
   Destructor called.
   Destructor called.
   P=22,35
   S=12,20
   Destructor called.
   Destructor called.
   Destructor called.
   Destructor called.
*/
```

- 使用拷贝构造函数的三种情况

  （1）明确表示由一个对象初始化另一个对象时；

  例如：TPoint N(M);

  （2）当对象作为函数实参传递给函数形参时（传数据值调用）；

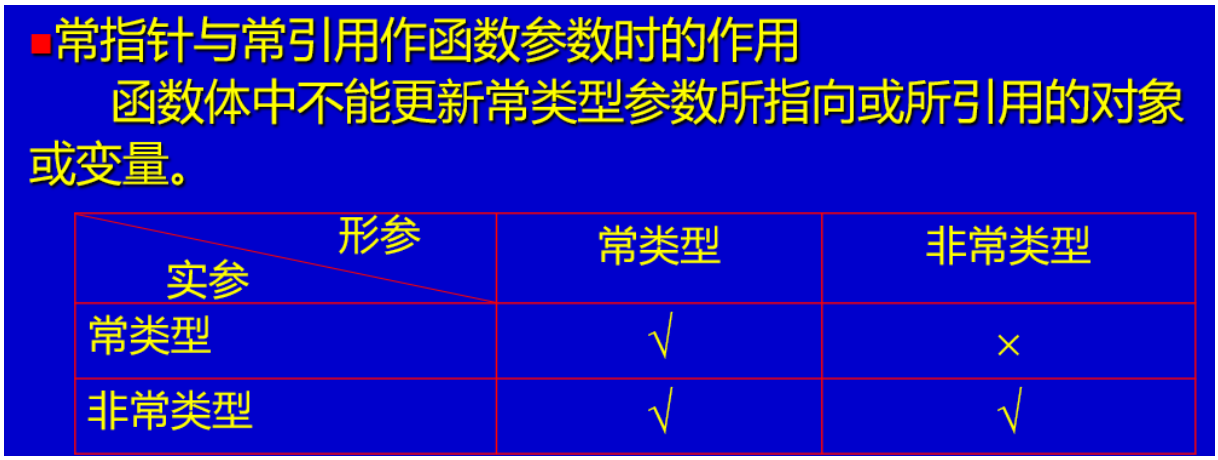
  例如：P=fun(N);

  （3）当对象作为函数返回值时（数据值）；

  例如：return R;

- **static(选择)**

  - 静态成员在整个程序运行过程中只会生成一个实例
  - 对于公有的静态成员函数，可以通过类名或对象名来调用，而一般的非静态成员函数只能通过对象名来调用。静态成员函数可以由类名通过符号“::”直接调用。
  - 静态成员函数可以直接访问该类的静态数据成员和静态函数成员，不能直接访问非静态数据成员和非静态成员函数.

- **const(记住就近原则就ok)**
  - 这里记住就近原则就行了
    例如
    int fun() const{} 这里const离函数体近 所以是常函数
    char * const ps1 = s1; const离*近 所以是指针常量 指针是常量，指针的地址不可改变
    const cahr * ps2 = s2; const离变量类型近 所以是常量指针 指针指向的是常量，常量不可以改变
  - 

  - 上面的图的意思是传参的时候非const修饰的变量不可以传给const修饰的函数参数
- **对象数组(稍微了解)**
  - 构造顺序与析构顺序相反
  - 例

```
#include <iostream>
using namespace std;
class Date {
public:
    Date() {
        month=day=year=0;
        cout<<"Default constructor called."<<endl;
    }
    Date(int m,int d,int y) {
        month=m;    day=d;    year=y;
        cout<<"Constructor called."<<day<<endl;
```

```
    }
    ~ Date (){
        cout<<"Destructor called."<<day<<endl;
    }
    void Print(){
        cout<<"Month="<<month<<",Day="<<day
               <<",Year="<<year<<endl;
    }
private:
    int month, day ,year;
};
int main(){

    Date dates[5]={Date(7,22,1998),Date(7,23,1998),
                               Date(7,24,1998)};
    dates[3]= Date(7,25,1998);    dates[4]= Date(7,26,1998);
    for(int i=0;i<5;i++)  dates[i].Print();
    return 0;
}
/*
  Constructor called.22
  Constructor called.23
  Constructor called.24
  Default constructor called.
  Default constructor called.
  Constructor called.25
  Destructor called.25
  Constructor called.26
  Destructor called.26
  Month=7,Day=22,Year=1998
  Month=7,Day=23,Year=1998
  Month=7,Day=24,Year=1998
  Month=7,Day=25,Year=1998
  Month=7,Day=26,Year=1998
  Destructor called.26
  Destructor called.25
  Destructor called.24
  Destructor called.23
  Destructor called.22
*/
```

- 构建指针数组的时候不会让类实例化 ppt第二章67页开始

this指针(了解)

- 类的每个非静态成员函数都有一个隐含的this指针参数，不需要显示说明。this指针指向调用该函数的对象
- 举例

```
class A{
public:
    int fun(){}
};

int main(){
    A a;
    a.fun(); // 这里调用的时候看似没有传参
    // 但是fun内部其实
    // fun(A *this){}
    // 是这样的
}
```

友元(了解)

- 友元提供了在不同类的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制，友元分为友元函数和友元类。
- 友元函数可以访问类内私有成员
- 友元类 非特别重点 可以看一下ppt第二章86页

第三章

继承方式(选择)

- 公有继承
- 保护继承
- 私有继承
-

继承方式 / 基类成员	public	private	protected
public	public	private	protected
private	不可访问	不可访问	不可访问
protected	protected	private	protected

    - **关于基类对象的访问权限请看上图**

- **派生类的构造过程与析构过程(选择填空)**
    - 调用基类的构造函数
    - 对派生类的成员初始化列表中数据成员（包括子对象、常量、引用等必须初始化的成员）按照**声明顺序(上到下)(注意与初始化表顺序无关)**，依次初始化
    - 派生类构造函数函数体
    - **析构与构造完全相反(包括第二步)**
    - 注：常数据成员、引用成员、子对象，只能在成员初始化列表中进行初始化。
- **多继承**
    - 构造与析构：按照声明顺序(从左至右)进行
    - **菱形继承(虚继承)(代码结果分析)**
        - 多继承的二义性问题
        - 虚基类子对象由**最远派生类**(因为虚基类的构造函数只会被调用一次)的构造函数通过**调用虚基类的构造函数**进行初始化
        - 例

```
#include <iostream>
using namespace std;
class CBase0 {
protected:
```

```
     int b0;
public:
     CBase0(int x=0){ b0=x; }
     int GetB0(){ return b0; }
};
class CBase1 : virtual public CBase0 {
public:
    CBase1(int x=0) : CBase0(x){}
};
class CBase2 : virtual public CBase0 {
public:
     CBase2(int x=0) : CBase0(x){}
};
class CDerived : public CBase1,public CBase2 {
public:
    CDerived(int x,int y,int z):CBase0(x),CBase1(y),CBase2(z)
{}
};
int main(){
    CDerived d1(10,15,20);
    cout << d1.GetB0() <<endl;
    cout << d1.CBase1::GetB0() <<endl;
    cout << d1.CBase2::GetB0() <<endl;
    return 0;
}
```

第四章(代码大题概率较大)

- ## 多态
  - 多态类型(了解)
    - 静态多态：编译时即可确定名称
    - 动态多态：执行时才能确定名称
- ## 运算符重载(大题)
  - 前缀(++i, --i之类的)operator()
  - 后缀(i++, i--之类的)operator(**int**)
  - 注意一下赋值(=)运算符重载

- **静态联编与动态联编**
  - 如果指针或引用以及虚函数缺任何一个就是静态联编

```
#include <iostream>
using namespace std;
class A{
public:
    A() {a=0;}
    A(int i) {a=i;}
    void Print() {cout<<a<<endl;}
    int Geta() {return a;}
private:
    int a;
};
class B:public A{
public:
    B() {b=0;}
    B(int i,int j):A(i),b(j) {}
    void Print() {A::Print();cout<<b<<endl;}
private:
    int b;
};
void fun(A& d){
    cout<<d.Geta()*10<<endl;
}
int main(){
    B bb(9,5);
    A aa(5);
    aa=bb; // 静态联编
    aa.Print(); // 调用的就是A里的方法
    A *pa=new A(8);
    B *pb=new B(1,2);
    pa=pb; // 静态联编
    pa->Print(); // 调用的还是A里的方法
    fun(bb); // 依旧是A的方法
    return 0;
}
/*
9
```

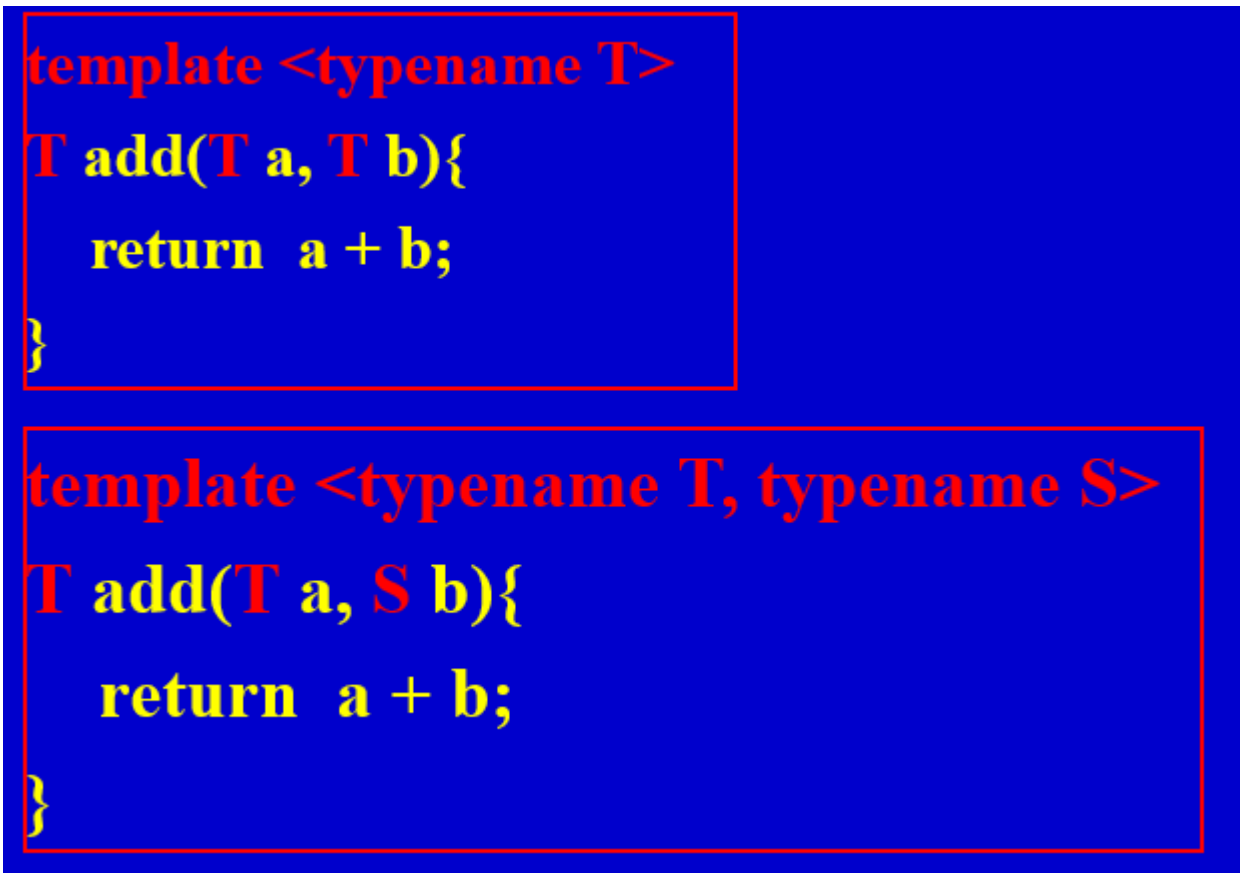
```
1
90
*/
```

- **动态联编(大题)**
  - 派生类继承方式必须为公有继承
  - 虚函数机制----------虚函数表(看一下sizeof那个题)
- **虚析构函数(代码结果填空)**
  - 一般将虚构函数定义为虚析构函数(但是构造函数不可以定义为虚函数)
  - 例

```
class B:public A{
public:
    B(int i) {buf=new char[i];}
    virtual ~B(){
        delete [] buf;
        cout<<"B::~B() called. "<<endl;
    }
private:
    char *buf;
};
int main(){
    A *a=new B(15);
    delete a;
    return 0;
}
/*
    B::~B() called.
    A::~A() called.
*/
```

第五章(代码结果填空)

- **函数模板**
-



- **类模板**
- 类似上面(但是是类的模板)
- **注：模板用替换就可以了**