C++第六讲

一、多态

面向对象三大特征：封装、继承、多态
多态：实现“一物多用”，一种形式多种状态，是实现泛型编程的一种
泛型编程：试图以不变的代码，实现可变的功能

1.1 函数重写

在子类中定义与父类原型相同的函数，这个过程就叫函数重写
原型相同：函数名、参数（参数个数、参数类型）必须一致，但是，函数体不同
注意：函数重写发生在父子类直接
辨析：重载、重写
重载：函数名相同、形参列表必须不同，发生在同一个类中
重写：函数名相同、形参列表必须相同，发生在不同类中

1.2 虚函数

1> C++中规定，在定义成员函数时，在前面加关键字virtual，那么此函数便是虚函数
2> 虚函数需要在子类中进行重写之后使用
3> 声明虚函数的情况
i> 先看一下函数所在的类是否被作为基类，如果作为基类，再看一下子类中是否对该函数进行了重写，如果没有重写，则无需定义成虚函数
ii> 如果子类中重写了基类的函数，还需再看一下，是否有父类的指针或者引用指向子类对象，如果没有，则也无需定义成虚函数
iii> 如果父类指针或引用指向子类对象了，再看一下，是否通过父类指针或引用调用子类重写的该函数，并且使用的是子类的功能，如果没有，也无需定义成虚函数
iV> 当父类指针或者引用指向子类对象，并且想要通过父类指针或引用调用子类中的函数时，那么需要将该函数在父类中进行定义，并且设置为虚函数。
V> 在父类设置虚函数时，如果只是单纯为了让子类进行重写，那么父类的函数体内可以为空。
4> 只要有一个父类中设置虚函数，那么，该类的所有子类中的该函数全部为虚函数，即使子类中没有加关键字virtual，子类中的该函数依然是虚函数

```

#include <iostream>

```

```cpp
using namespace std;

class A
{
protected:
    int age;

public:
    A() {}

    virtual void show()
    {
        cout<<"I am A"<<endl;
    }
};

class B:public A
{
private:
    int num;
public:
    B(){}
    void show()
    {
        cout<<"I am B"<<endl;
    }
};


int main()
{
    cout<<sizeof(A)<<endl;         //4
    cout<<sizeof(B)<<endl;         //8

    B b1;
    A &a1 = b1;

    a1.A::show();       //?

    return 0;
}
```

1.3 多态

通过父类的指针或者引用指向子类的对象，可以调用子类中重写的父类的虚函数

1.4 实现多态的必要条件

1> 函数重写
2> 虚函数
3> 父类指针指向子类对象

```
#include <iostream>

using namespace std;

class Father
{
protected:
    string name;
    int age;
public:
    Father(){}
    Father(string n, int a):name(n), age(a){}

   virtual void show()
    {
        cout<<name<<endl;
        cout<<age<<endl;
        cout<<"I am Father"<<endl;
    }
};

class Son : public Father
{
private:
    string toy;

public:
    Son() {}
    Son(string n, int a, string t):Father(n,a), toy(t) {}

    void show()
    {
        cout<<toy<<endl;
```

```
    }

};

class Son2 : public Father
{

public:
    Son2() {}
    virtual ~Son2() {}

    void show()
    {
        cout<<"I am second"<<endl;
    }
};


int main()
{
    Son s("zhangs", 18, "car");
    s.show();
    Son2 s2;

    cout<<"****************************\n";
    Father &f = s;          //定义父类的引用目标为子类的对象
    f.show();           //?

    cout<<"****************************\n";
    Father *p = &s;
    p->show();              //car

    cout<<"****************************\n";
    Father *p2 = &s2;
    p2->show();              //我是老二

    cout<<"****************************\n";
    Father f1("zhangpp", 20);
    f1.show();
    return 0;
}
```

1.5 多态应用的举例

```
#include <iostream>

using namespace std;

class Animal
{
protected:
    string name;
    string food;      //食物
public:
    Animal() {}
    Animal(string n, string f):name(n), food(f) {}

    virtual void voice()
    {

    }
};

class Sheep : public Animal
{
private:
    int leg;          //腿的个数

public:
    Sheep() {}
    Sheep(string n, string f, int l):Animal(n, f), leg(l) {}

    void voice() override
    {
        cout<<name<<"   "<<food<<"   "<<leg;
        cout<<"   mie mie mie..."<<endl;
    }
};

class Wolf : public Animal
{
private:
    int tail;         //尾巴的个数

public:
    Wolf() {}
```

```
    Wolf(string n, string f, int t):Animal(n, f), tail(t) {}

    void voice() override
    {
        cout<<name<<"   "<<food<<"   "<<tail;
        cout<<"   wu wuu wuuu..."<<endl;
    }
};


void print(Animal & a)          //定义全局函数，实现叫声
{
    a.voice();
}


int main()
{
    Sheep s("xiyy", "gress", 4);
    s.voice();

    Wolf w("xiaohh", "meat", 1);
    w.voice();

    cout<<"*******************************"<<endl;
    print(s);       //羊的叫声
    print(w);         //狼的叫声

    return 0;
}

```

二、虚析构函数

1> 在四个特殊成员函数中，只有析构函数能设置成虚函数

2> 功能：争取指引delete关键字释放子类空间的

3> 定义格式：在定义析构函数之前加关键字virtual即可

4> 即使子类的析构函数没有设置成虚析构函数，也是虚析构函数

5> 在以后开发过程中，如果定义的类作为基类使用，那么要将其析构函数设置成虚析构函数

```
#include <iostream>

using namespace std;

```

```
4
5  class Person
6  {
7  protected:
8      string name;
9      int age;
10 public:
11     Person() {}
12     Person(string n, int a):name(n), age(a)
13     {cout<<"Person::construct"<<endl;}
14     //将父类的析构函数设置成虚析构函数
15     //能争取指引delete关键字将子类的空间一起析构
16     virtual ~Person() {cout<<"Person::xigou"<<endl;}
17     virtual void show() = 0;           //纯虚函数
18 };
19
20 class Yellow : public Person
21 {
22 private:
23     string skin;       //肤色
24 public:
25     Yellow() {}
26     Yellow(string n, int a, string s):Person(n,a), skin(s)
27     {cout<<"Yellow::construct"<<endl;}
28      ~Yellow() {cout<<"Yellow::xigou"<<endl;}
29
30     void show()
31     {
32         cout<<name<<"  "<<age<<"  "<<skin<<endl;
33     }
34 };
35
36
37 int main()
38 {
39     Person *p = new Yellow("zhangpp", 18, "yellow");
40
41     p->show();        //?
42
43     delete p;
44
45     return 0;
46 }
47
```

三、纯虚函数（抽象）

3.1 引入纯虚函数的目的

在某些情况下，在类中的虚函数，我们不想使用父类的实例来调用，或者说使用父类的实例调用没有意义，主要是为了让子类进行重写的，那么此时，我们可以将该函数设置成纯虚函数

3.2 声明纯虚函数

在类中，只有声明，没有定义
virtual 返回值类型 函数名（参数列表） = 0;

3.3 纯虚函数使用要求

在父类中进行声明，在子类中必须进行重写，如果子类不重写，那么子类也用不了

3.4 抽象类

只有类中包含至少一个纯虚函数，那么该类便是抽象类

3.5 抽象类的特点

不能进行实例化对象，但是可以定义抽象类的引用或者指针的

```
#include <iostream>

using namespace std;

class Animal    //如果类中有纯虚函数，那么该类便是抽象类
                //抽象类不允许实例化对象，但是可以用抽象类的指针或引用
{
protected:
    string name;
    string food;       //食物
public:
    Animal() {cout<<"Animal::construct"<<endl;}
    Animal(string n, string f):name(n), food(f) {}

    virtual void voice() = 0;    //纯虚函数
                    //所在类，无需通过实例调用该函数，只是为了让子类进行重写的函数
```

```
17
18  };
19
20  class Sheep : public Animal
21  {
22  private:
23      int leg;          //腿的个数
24
25  public:
26      Sheep() {}
27      Sheep(string n, string f, int l):Animal(n, f), leg(l) {}
28
29      void voice() override
30      {
31          cout<<name<<"   "<<food<<"   "<<leg;
32          cout<<"   mie mie mie..."<<endl;
33      }
34  };
35
36  class Wolf : public Animal
37  {
38  private:
39      int tail;          //尾巴的个数
40
41  public:
42      Wolf() {}
43      Wolf(string n, string f, int t):Animal(n, f), tail(t) {}
44
45      void voice() override
46      {
47          cout<<name<<"   "<<food<<"   "<<tail;
48          cout<<"   wu wuu wuuu..."<<endl;
49      }
50  };
51
52
53  void print(Animal & a)
54  {
55      a.voice();
56  }
57
58
59
60  int main()
61  {
```

```
    Sheep s("xiyy", "gress", 4);
    s.voice();

    Wolf w("xiaohh", "meat", 1);
    w.voice();

    cout<<"*******************************"<<endl;
    print(s);       //羊的叫声
    print(w);         //狼的叫声

    cout<<"*******************************"<<endl;
    //Animal a;         //报错，抽象类是不允许实例化对象的
    //a.voice();

    return 0;
}

```

四、异常处理机制

1> C++提供了异常处理机制
2> 在代码执行过程中，可以抛出对应的异常，然后进行捕获处理
3> 格式

```
被调函数中使用关键字throw抛出对应的异常
在主调函数中使用
try{}catch(){}来进行捕获和处理异常
```

```
#include <iostream>

using namespace std;

double my_div(double m, double n)
{
    if(n == 0)
    {
        throw -1.1;
    }
    return m/n;
}


int main()
{
```

```
    try{
        double res = my_div(1,0);
        cout<<res<<endl;
    }catch(int e)        //捕获所有的整形异常
    {
        if(e == -1)
        {
            cout<<"分母不能为0"<<endl;
        }
    }catch(double e)    //捕获所有的小数异常
    {
        if(e == -1.1)
        {
            cout<<"分母不能为0"<<endl;
        }
    }

    return 0;
}

```

五、Larmda表达式

```
    1>  可以认为是一个轻量版的函数，是仿函数的一种
    2>  格式：[捕获列表]（参数列表）mutable ->返回值{函数体};
    3>  []：是捕获列标，可以将表达式外部的变量供我使用
    4>  [=]: 表示将外界的所有变量都以值捕获形式使用
    5>  [&]: 表示将外界的所有变量都以引用捕获形式使用
    6>  [=,&a]:表名除了a是引用捕获，其余都是值捕获
    7>  [&,a]:表名除了a是值捕获，其余都是引用捕获
    8>  (参数列标)：形式参数，用来传值的
    9>  mutable:该关键字修饰的函数，可以在表达式中进行值捕获的变量的修改
    10> ->返回值：返回值类型，可以省略
    11> {}：中是函数体内容
    12> 因为这是一个表达式，别忘了分号结尾
```

```
#include <iostream>

using namespace std;

int main()
{

```

```
    int a = 520;
    int b = 1314;
    int c = 999;

    cout<<&a<<"   "<<&b<<"   "<<&c<<endl;

    //auto larmda = [&, c](int m, int n)->void      //除了c之外，其他都是引用捕获
    //auto larmda = [a](int m, int n)->void      //表明通过值捕获了a但b和c不能使用
    //auto larmda = [&a](int m, int n)->void      //表明通过引用捕获了a，但是b和c不能使用
    //auto larmda = [&a, b](int m, int n)->void      //表明通过引用捕获了a，值捕获了b，但是c不能使用
   // auto larmda = [a,b,c](int m, int n)->void      //表明三个变量全部都是值捕获
    //auto larmda = [=](int m, int n)->void      //表明所有变量全部都是值捕获
    auto larmda = [=, &a](int m, int n)mutable    //表名除了a之外，其余都是值捕获
    {
        b = 3399;

        cout<<&a<<"   "<<&b<<"   "<<&c<<endl;

        cout<<a<<endl;
        cout<<b<<endl;       //3399
        cout<<m+n<<endl;
    };

    larmda(10, 20);

    cout<<b<<endl;

    return 0;
}
```

六、模板

6.1 模板函数

1> 有时程序员定义函数时，由于函数参数类型不同，或者参数的返回值类型不同，会导致定义多个功能性质相同的函数，造成代码的冗余，此时我们可以考虑定义函数模板来实现

2> 所谓函数模板，就是在定义函数时，函数参数和返回值的类型都不给定，而是由函数调用时实参进行传递过来使用

3> 定义函数模板时跟普通函数有所区别，但是调用函数时，隐式调用跟普通函数调用没有区别，系统会根据传递参数的类型，自动推导函数的类型。

4> 也可以进行显式调用，在调用函数名后加个尖括号，写上要传递的类型：尖找尖，圆找圆

5> 一个模板只能对应下面一个函数，如果要再定义目标函数，需要重新定义模板参数

```
#include <iostream>

using namespace std;

//定义求和函数
//int my_sum(int m, int n)
//{
//    return m+n;
//}

//函数重载
//double my_sum(double m, double n)
//{
//    return m+n;
//}

//函数模板
template <class T>
T my_sum(T m, T n)
{
    return m+n;
}

template <typename T>
T my_add(T a, T b)
{
    return a+b;
}
/*
 * template <typename T>
```

```
 * template：定义模板的关键字，说明要开始定义模板了
 * <>:里面时类型形参名，里面可以有多个类型的参数，中间用逗号隔开
 *      template <typename T1，typename T2，。。。，Tn >
 * typename:声明类型的形参，也可以用关键字class
 * */

int main()
{
    int a = 520;
    int b = 1314;

    cout<<a+b<<endl;   //实现方式一
    cout<<my_sum<int>(a, b)<<endl;    //实现方式二
    cout<<my_sum<double>(1.2, 2.3)<<endl;   //实现方式三
    cout<<my_sum<string>(string("hello"), string("world"))
<<endl;       //实现方式四   函数模板

    return 0;
}
```

函数模板的特化

1> 如果基础模板和特化模板同时出现时，如果隐式调用，则调用的时基础模板，

2> 如果显示调用，优先调用特化模板，如果没有特化模板，则去调用基础模板

3> 在实际开发中，要求调用函数模板时，使用显式调用

```
#include <iostream>

using namespace std;

//函数模板
template <class T>
T my_sum(T m, T n)
{
    cout<<"AAAAAAAAAA"<<endl;
    return m+n ;
}

//函数模板
template <class T>
T my_sum(T m, int n)    //函数模板的特化
{
    cout<<"BBBBBBBBBBBBBBB"<<endl;
    return m+n;
```

```
}

int main()
{
    int res;

    res = my_sum<double>(520, 1314);      //特化模板
    res = my_sum(520, 1314);            //调用基础模板
    cout<<res<<endl;

    return 0;
}

```

6.2 模板类

```
#include <iostream>

using namespace std;

template<typename T>
class Node
{
private:
    T data;         //数据域
    Node *next;         //指针域
public:
    Node() {}
    Node(T d):data(d), next(NULL) {}

    void show();

};

//后面只要用到Node，就要加<T>,并且在前面要声明模板
template<typename T>
void Node<T>::show()
{
    cout<<data<<endl;
}


int main()
{

```

```
29      //必须显性调用
30      Node<int> p1(10);
31      p1.show();
32
33      Node<char> p2('a');
34      p2.show();
35
36      Node<string> p3("hello");
37      p3.show();
38
39      return 0;
40  }
```

6.3 模板函数和模板类的实现机制(笔试面试题)

核心机制：延时编译、二次编译

当编译器第一次遇到函数模板或者类模板时，由于类型不明确，所以只是对除了类型之外的其他的相关语法的检查，如果检查无误，则会生产模板的内部实现机制。

当编译器第二次遇到函数模板或者类模板时，会根据传过来的实参的类型，确定模板类型，再一次进行语法检查，如果无误，则会生产函数或者类的原型。

七、STL标准模板库

C++ 标准模板库(STL)

C++ STL (Standard Template Library标准模板库) 是通用类模板和算法的集合，它提供给程序员一些标准的数据结构的实现如 queues(队列), lists(链表), 和 stacks(栈)等.

C++ STL 提供给程序员以下三类数据结构的实现:

- 顺序结构
  - C++ Vectors
  - C++ Lists
  - C++ Double-Ended Queues
- 容器适配器
  - C++ Stacks
  - C++ Queues
  - C++ Priority Queues
- 联合容器
  - C++ Bitsets
  - C++ Maps

- C++ Multimaps
- C++ Sets
- C++ Multisets

程序员使用复杂数据结构的最困难的部分已经由STL完成. 如果程序员想使用包含int数据的stack, 他只要写出如下的代码:

stack myStack;

接下来, 他只要简单的调用 push() 和 pop() 函数来操作栈. 借助 C++ 模板的威力，他可以指定任何的数据类型，不仅仅是int类型. STL stack实现了栈的功能，而不管容纳的是什么数据类型.

7.1 vectors

1> 介绍

Vectors 包含着一系列连续存储的元素,其行为和数组类似。访问Vector中的任意元素或从末尾添加元素都可以在常量级时间复杂度内完成，而查找特定值的元素所处的位置或是在Vector中插入元素则是线性时间复杂度。

2> 常用函数

```
1、构造函数
  vector();
  vector( size_type num, const TYPE &val );
  vector( input_iterator start, input_iterator end );
2、判空函数： bool empty();如果当前vector没有容纳任何元素,则empty()函数返回true,否则返回false
3、容量：size_type capacity(); 返回当前vector在重新进行内存分配以前所能容纳的元素数量
4、大小： size_type size();函数返回当前vector所容纳元素的数目
5、尾插： void push_back( const TYPE &val );添加值为val的元素到当前vector末尾
6、尾删： void pop_back();函数删除当前vector最末的一个元素
7、元素访问： TYPE at( size_type loc );返回当前Vector指定位置loc的元素的引用. at() 函数 比 [] 运算符更加安全, 因为它不会让你去访问到Vector内越界的元素
8、找第一个元素：TYPE front();返回当前vector起始元素的引用
9、找最后一个元素：TYPE back();返回当前vector最末一个元素的引用
10、返回第一个元素的迭代器：iterator begin();
11、最后一个元素迭代器： iterator end();返回一个指向当前vector末尾元素的下一位置的迭代器
12、清空容器： void clear();删除当前vector中的所有元素.

```

```cpp
#include <iostream>
#include<vector>

using namespace std;

int main()
{
    vector<int> v1;         //无参构造一个vector对象

    //判断容器内是否为空
    if(v1.empty())
    {
        cout<<"empty"<<endl;
    }else
    {
        cout<<"not empty"<<endl;
    }

    //求容器大小
    cout<<"size of v1:"<<v1.size()<<endl;

    //求容器当前最大容量
    cout<<"capacity of v1:"<<v1.capacity()<<endl;

    //进行尾插
    for(int i=0; i<5; i++)
    {
        v1.push_back(i+10);
        cout<<"capacity of v1:"<<v1.capacity()<<endl;
    }
    cout<<"size of v1:"<<v1.size()<<endl;

    //尾删
    v1.pop_back();
    cout<<"size of v1:"<<v1.size()<<endl;

    for(int i=0; i<v1.size(); i++)
    {
        //cout<<v1.at(i)<<" ";
        cout<<v1[i]<<" ";
    }
    cout<<endl;

    //找到第一个和最后一个
    cout<<"the first one:"<<v1.front()<<endl;
```

```
    cout<<"the last one:"<<v1.back()<<endl;

    //使用迭代器遍历容器

    for( auto it=v1.begin(); it!=v1.end(); it++ )
    //for( vector<int>::iterator it=v1.begin(); it!=v1.end(); it++ )
    {
        cout<<*it<<" ";
    }
    cout<<endl;

    //使用枚举for循环输出容器信息
    for(int val:v1)
    {
        cout<<val<<" ";
    }
    cout<<endl;

    //清空所有元素
    v1.clear();
    cout<<"size of v1:"<<v1.size()<<endl;


cout<<"**********************************************************"<<endl;
    vector<char> v2(5, 'K');
    for(char val:v2)
    {
        cout<<val<<" ";
    }
    cout<<endl;

cout<<"**********************************************************"<<endl;

    int arr[10] = {2,3,6,8,4,5,1,9,37,7};
    vector<int> v3(arr, arr+6);
    for(auto val:v3)
    {
        cout<<val<<" ";

    }
    cout<<endl;
    return 0;
```

```
86 }
87
```

7.2 迭代器

C++ Iterators(迭代器)

迭代器可被用来访问一个容器类的所包函的全部元素，其行为像一个指针。举一个例子，你可用一个迭代器来实现对vector容器中所含元素的遍历。有这么几种迭代器如下：

迭代器	描述
input_iterator	提供读功能的向前移动迭代器，它们可被进行增加(++)，比较与解引用（*）。
output_iterator	提供写功能的向前移动迭代器，它们可被进行增加(++)，比较与解引用（*）。
forward_iterator	可向前移动的，同时具有读写功能的迭代器。同时具有input和output迭代器的功能，并可对迭代器的值进行储存。
bidirectional_iterator	双向迭代器，同时提供读写功能，同forward迭代器，但可用来进行增加(++)或减少(--)操作。
random_iterator	随机迭代器，提供随机读写功能.是功能最强大的迭代器， 具有双向迭代器的全部功能，同时实现指针般的算术与比较运算。
reverse_iterator	如同随机迭代器或双向迭代器，但其移动是反向的。（Either a random iterator or a bidirectional iterator that moves in reverse direction.）（我不太理解它的行为）

第种容器类都联系于一种类型的迭代器。第个STL算法的实现使用某一类型的迭代器。举个例子，vector容器类就有一个**random-access**随机迭代器，这也意味着其可以使用随机读写的算法。既然随机迭代器具有全部其它迭代器的特性，这也就是说为其它迭代器设计的算法也可被用在vector容器上。

如下代码对vector容器对象生成和使用了迭代器：

```
  vector<int> the_vector;
  vector<int>::iterator the_iterator;

  for( int i=0; i < 10; i++ )
    the_vector.push_back(i);

  int total = 0;
  the_iterator = the_vector.begin();
  while( the_iterator != the_vector.end() ) {
    total += *the_iterator;
    the_iterator++;
  }
  cout << "Total=" << total << endl;
```

提示：通过对一个迭代器的解引用操作（*），可以访问到容器所包含的元素。

7.3 list

C++ Lists（链表）

Lists将元素按顺序储存在链表中. 与 向量(vectors)相比，它允许快速的插入和删除，但是随机访问却比较慢.

```
常用方法
1、构造函数
    list();
    list( size_type count,const T& value);
    list( size_type count );
2、获取第一个元素：reference front();
3、获取最后一个元素：reference back();
4、判空函数： bool empty();如果当前list没有容纳任何元素,则empty()函数返回true,否则返回false
5、容量：size_type capacity(); 返回当前vlist在重新进行内存分配以前所能容纳的元素数量
6、大小： size_type size();函数返回当前list所容纳元素的数目
7、尾插： void push_back( const TYPE &val );添加值为val的元素到当前list末尾
8、尾删： void pop_back();函数删除当前list最末的一个元素
9、头插： void push_front( const TYPE &val );添加值为val的元素到当前list第一个位置
10、头删： void pop_front();函数删除当前list第一个元素
11、排序
    void sort();    //升序
    void sort( Comp compfunction ); //带有策略的排序
```

作业

仿照系统的vector，手动实现一个my_vector