C Plus Plus

1. 面向对象基础知识

- 日常生活中，我们都习惯于将事物**分类。**

1.1 面向对象基本概念

面向对象的**意义**在于

- 将日常生活中**习惯的思维方式引入程序设计中**
- 将需求中的概念**直观的映射**到解决方案中
- 以**模块为中心**构建可复用的软件系统
- 提高软件产品的**可维护性和可扩展性**

类和**对象**是面向对象中的两个基本概念

- 类：指的是**一类事物**，是一个**抽象的概念**
- 对象：指的是属于某个类的**具体实体**
- 类是**一个模型**，**这种模型可以创建出不同的对象实体**
- 对象实体是类模型的一个**具体实例**

一个类可以有很多对象，而一个对象必然属于某个类。

- 自然语言：中文、英文；
- 老虎：华南虎、东北虎、孟加拉虎；

类和对象的意义

- 类用于**抽象的描述**一类事物所特有的属性和行为；
    - 如：电脑拥有CPU，内存和硬盘，并且可以开机和运行程序；
- 对象是**具体的事物**，**拥有所属类中描述的一切属性和行为**
    - 如：每一只老虎都有不同的体重，不同食量以及不同的性情；

一些有趣的问题：

- 类一定存在实际的对象吗？
    - 不一定，比如恐龙；
- 类的对象数目是确定的吗？
    - 不确定的；
- 类一定都来源于现实生活中吗？
    - 不一定
- 类都是独立的吗？类之间存在关系吗？
    - 不
- 对象实例一定只属于一个类吗？
    - 不一定
- 对象实例可能完全相同吗？
    - 不可能

类之间的基本关系

- 继承

    - 从已存在类**细分出来的类**和原类之间具有继承关系(is-a)——就是的关系，是单向的关系；
    - 继承的类（ 子类） 拥有原类（父类） 的所有属性和行为
- 组合
    - 一些类的存在**必须依赖**于其它的类， 这种关系叫组合；
    - 组合的类在某一个局部上由其它的类组成；同生死，共存亡；整体与局部的关系；

例如将人类、动物、生物进行分类

```
#include <iostream>

using namespace std;

struct Biology {
bool living;
};

struct Animal : Biology {
bool movable;
void findFood() {}
};

struct Plant : Biology {
bool growable;
};

struct Beast: Animal
{
    /* data */
    void sleep();
};

struct Human : Animal {
void sleep() {}
};

int main(){
    printf("hello\n");
}
```

1.2 类和封装的概念

前面我们主要是巩固了分类的思想，那么还有什么可以挖掘的概念？那就是封装的思想；

- 类通常分为以下两个部分
    - **类的实现细节**
    - **类的使用方式**
- 当**使用类**时， 不需要关心其实现细节
- 当**创建类**时， 才需要考虑其内部实现细节

封装的概念

- 根据经验： 并不是类的每个属性都是对外公开的
    - 如： 女孩子不希望外人知道自己的体重和年龄
    - 如： 男孩子不希望别人知道自己的身高和收入
- 而一些类的属性是对外公开的

    - 如： 人的姓名， 学历， 国籍， 等
- 必须在类的表示法中定义属性和行为的公开级别
    - 类似文件系统中文件的权限

有些东西不希望别人知道，也就是封装的概念。

C + + 中类的封装

- 成员变量： C + + 中用于表示类属性的变量
- 成员函数： C + + 中用于表示类行为的函数
- C + + 中可以给成员变量和成员函数定义**访问级别**
    - public
        - 成员变量和成员函数可以在类的内部和外界访问和调用
    - private
        - 成员变量和成员函数只能在类的内部被访问和调用

```
#include <iostream>

using namespace std;

struct Biology {
bool living;
};

struct Animal : Biology {
bool movable;
void findFood() {}
};

struct Plant : Biology {
bool growable;
};

struct Beast: Animal
{
    /* data */
    void sleep();
};

struct Human : Animal {
void sleep() {}
};

struct Girl : Human {
    private:
        int age;
        int weight;
    public:
        void print(){
            age = 22;
            weight = 40;
            printf("I am a girl %d, weight %d \n", age, weight);
        }
};
```

```
int main(){
    Girl g;
    printf("hello\n");
    //printf("g.age = %d", g);
    g.print();
    return 0;
}
```

类成员的作用域

- 类成员的作用域都只在类的内部， 外部无法直接访问；
- 成员函数可以直接访问成员变量和调用成员函数；
- 类的外部可以通过类变量访问public 成员；
- **类成员的作用域与访问级别没有关系**——访问级别只是针对外部的；
    - C++中用`struct`定义的类中所有成员默认为public

要访问类的成员变量和成员函数必须通过类的对象进行，是否访问成功需要看访问级别。

```
#include <stdio.h>

int i = 1;

struct Test
{
    /* data */
    private:
        int i;

    public:
        int j;
        int geti()
        {
            i = 3;

            return i;
        }
};

int main(){

    int i = 2;
    Test test;

    test.j = 4;

    printf("i = %d\n", i); //i =2
    printf("::i = %d\n", ::i); //::i = 1;
    printf("test.i = %d\n", test.i); //erro
    printf("test.j = %d\n", test.j); //test.j = 4
    printf("test.geti = %d\n", test.geti); //test.geti() =3
}
```

小结

- 类通常可以分为使用方式和内部细节两部分

- 类的封装机制使得**使用方式和内部细节相分离**
- **C+ +中通过定义类成员的访问级别实现封装机制**
- public 成员可以在类的内部和外界访问和调用
- private 成员只能在类的内部被访问和调用

1.3 类的真正形态

经过不停的改进， 结构体`struct`变得越来越不像它在C 语言中的样子了！ ！！

`struct`在C 语言中已经有了自己的含义, 必须继续兼容。

在C+ +中提供了新的关键字`class`用于类定义。

`class`和`struct`的用法是完全相同的。具体的不同：

- 在用`struct`定义类时， 所有成员的默认访问级别为public
- ```
  struct A{
      //default to public
      int i;
      //default to public
      int getI(){
          return i;
      }
  }
  ```
- 在用`class`定义类时， 所有成员的默认访问级别为private
- ```
  class B{
      //default to private
      int i;
      //default to private
      int getI(){
          return i;
      }
  }
  ```

  小实例

  需求： 开发一个用于四则运算的类

  - 提供`setOperator`函数设置运算类型
  - 提供`setParameter`函数设置运算参数
  - 提供`result`函数进行运算
    - 其返回值表示运算的合法性
    - 通过引用参数返回结果

思考：类分为两部分，类的实现和类的使用，那么我们应该在代码中如何实现呢?

C+ +中的类支持**声明和实现**的分离

将类的实现和定义分开

- `.h`头文件中只有类的声明
  - 成员变量和成员函数的声明
- `.cpp`源文件中完成类的其它实现
  - 成员函数的具体实现

`Operator.h`:(用户的需求)

```
#ifndef _OPERATOR_H_
#define _OPERATOR_H_

class Operator
{
private:
    /* data */
    char mpO;
    double mp1;
    double mp2;
public:
    bool setOperator(char op);
    void setParameter(double p1, double p2);
    bool result(double &r);
};

#endif
```

`Operator.c`:（工程师的需求)

```
#include "Operator.h"

bool Operator::setOperator(char op){
    bool ret = false;

    if((op == '+') || (op == '-') || (op == '*') || (op == '/')   ){
        ret = true;
        mpO = op;
    }
    else{
        mpO = '\O';
    }
    return ret;
}

void Operator::setParameter(double p1, double p2){
    mp1 = p1;
    mp2 = p2;
}

bool Operator::result(double &r){
    bool ret = true;

    switch (mpO)
    {
    case '/':
        /* code */
        balal;
        break;
    case '+':
        balabal;
        break;
    default:
        ret = false;
```

```
        break;
    }
    return ret;
}
```

`mian.c`：（最后的使用案例）

```
#include <stdio.h>
#include "Operator.h"

int main(){
    Operator op;
    op.setOperator('/');
    op.setParameter(9,3);
    ...
}
```

小结：

- C++引进了新的关键字`class`用于定义类；
- `struct`和`class`的区别在于**默认访问级别的不同**
- C++中的类支持声明和实现的分离
    - **在头文件中声明类**
    - **在源文件中实现类**

1.4 对象的构造

对象中成员变量的初始值是多少？C++在定义类的时候不能够支持给成员变量赋值。

下面的类定义中成员变量`i`和`j`的初始值为什么？

```
class test
{
private:
    /* data */
    int i;
    int j;
public:
    int  getI(){
        return i;
    }
    int getJ(){
        return j;
    }
};
```

上面的结果为随机值；

当我们定义全局对象时，如下：

```
class test
{
private:
    /* data */
```

```
    int i;
    int j;
public:
    int  getI(){
        return i;
    }
    int getJ(){
        return j;
    }
};

test test;
int main(){
    //test对象的i,j初始为0
    test * pt = new test
    pt->i;
    pt->getI();
    //这里的test是分配在堆空间的，初始值应该为随机值。
    //同样需要释放内存。
    delete pt;
    return 0;
}
```

同样，我们可以定义一个指针`test * pt = new test`，类得到的是一个数据类型。

对象的初始化

- 从程序设计的角度， 对象只是变量，因此：
  - 在**栈**上创建对象时， 成员变量初始为随机值
  - 在**堆**上创建对象时， 成员变量初始为随机值
  - 在**静态存储区**创建对象时， 成员变量初始为0 值
- 生活中的对象都是在初始化后上市的
- 初始状态（ 出厂设置）是对象普遍存在的一个状态。

问题：**程序中如何对一个对象进行初始化?**

1. 我们可以通过在类中添加初始化函数：

```
class test{

public:
    void initial(){
        i = 1;
        j = 1;
    }
};

test test;
test.initial();
```

但是这种方式不是在创建对象的时候立马调用，而且`i,j`的值都不能更改。

一般而言，对象都需要一个确定的初始状态。

上面存在的问题：

- initialize 只是一个普通函数， 必须显示调用；

- 如果未调用initialize 函数， 运行结果是不确定的

2. 使用构造函数

C+ +中可以定义与类名相同的**特殊成员函数**

- 这种特殊的成员函数叫做构造函数
    - 构造没有任何返回类型的声明
    - 构造函数在对象定义时自动被调用

```
class test
{
private:
    /* data */
    int i;
    int j;
public:
    int  getI(){
        return i;
    }
    int getJ(){
        return j;
    }
    //构造函数
    test(){
        i = 1;
        j = 2;
    }
};
```

小结

- 每个对象在使用之前都应该**初始化**;
- 类的**构造函数**用于对象的**初始化**;
- 构造函数**与类同名并且没有返回值**;
- **构造函数在对象定义时自动被调用;**

带有参数的构造函数——每个类可以构造不同的初始化状态的对象。

- 构造函数可以根据需要定义参数;
- 一个类中可以存在多个重载的构造函数;
- 构造函数的重载遵循C+ +重载的规则

```
class Test
{
public:
    Test(int v)
    {
    // use v to initialize member
    }
};
```

友情提示

- **对象定义** - 申请对象的空间并调用构造函数;
- **对象声明** - 告诉编译器存在这样一个对象

- ```
  Test t; //定义对象并调用构造函数
  int main()
  {
      //告诉编译器存在名为t 的Test 对象,链接器会去寻找对象。
      extern Test t;
      return 0 ;
  }
  ```

构造函数的自动调用

```
class Test
{
public:
    Test () { }
    Test(int v) { }
};
int main()
{
    Test t; //调用Test ()
    //定义对象，初始化
    Test t1(1); // 调用Test(int v)
    Test t2 = 1 ; //调用Test(int v)
    //赋值和初始化是不一样的
    int i = 1; //初始化
    i = 1; //赋值
    return 0;
}
```

构造函数的调用

- 一般情况下，构造函数在**对象定义时被自动调用**
- 一些特殊情况下， **需要手工调用构造函数**

问题：如何创建一个对象数组?

```
Test ta[3] = {Test(), Test(1), Test(2)};
```

还有一种手工调用构造函数初始化方式：

```
Test t = Test(100);
```

小实例：

需求： 开发一个**数组类**解决**原生数组的安全性问题**

- 提供函数获取数组长度
- 提供函数获取数组元素
- 提供函数设置数组元素

`Intarray.cpp`：
```

```
    IntArray(int len){

    }
    int length(){

    }
    bool get(int index, int & value){

    }
    bool set(int index, int value){

    }
```

`Intarray.h`:

```
#ifndef _OPERATOR_H_
#define _OPERATOR_H_

class Operator
{
private:
    /* data */
    char mp0;
    double mp1;
    double mp2;
public:
    IntArray(int len);
    int length();
    bool get(int index, int & value);
    bool set(int index, int value);
};

#endif
```

小结

- 构造函数可以根据需要定义参数
- 构造函数之间可以存在重载关系
- 构造函数遵循C+ +中重载函数的规则
- 对象定义时会触发构造函数的调用
- 在一些情况下可以手动调用构造函数
- 无参构造函数用于定义对象的默认初始状态
- 拷贝构造函数在创建对象时拷贝对象的状态
- 对象的拷贝有浅拷贝和深拷贝两种方式
    - 浅拷贝使得对象的物理状态相同
    - 深拷贝使得对象的逻辑状态相同

两个特殊的构造函数

- **无参构造函数**
    - **没有参数**的构造函数
        - 当类中没有定义构造函数时，**编译器默认提供一个无参构造函数**，并且其函数体为空。

- **拷贝构造函数**
  - **参数为`const class_name &`的构造函数**
    - 当类中没有定义拷贝构造函数时，**编译器默认提供一个拷贝构造函数，简单的进行成员变量的值复制。**

当里面没有构造函数时，编译器会自动为其添加无参构造函数。

面向对象里面如何实现用一个对象初始化另外的对象呢?

```
Test t1;
Test t2 = t1;
//两个对象的结果一致
```

会自动添加一个拷贝构造函数。

```
Test(){

}
```

拷贝构造函数的意义

- 兼容C语言的初始化方式
- 初始化行为能够符合预期的逻辑——两个对象的状态一模一样
- 浅拷贝
  - 浅拷贝使得对象的物理状态相同——对象占用的空间不一样
- 深拷贝
  - 深拷贝使得对象的逻辑状态相同——对象指针指向的内容是一致的。

编译器提供的拷贝构造函数只提供浅拷贝。

深拷贝：——深的意思是要深入到对象里面的指考虑。

```
Test(const Test& t){
    i = t.i;
    j = t.j;
    p = new int;
    //自己去申请一段内存空间
    *p = *t.p;
    //然后放入原对象相同的值
}
```

什么时候需要进行深拷贝?

- 对象中有**成员指代**了系统中的资源
  - 成员指向了动态内存空间
  - 成员打开了外存中的文件
  - 成员使用了系统中的网络端口
  - .....
- 一般性原则
  - 自定义拷贝构造函数， **必然需要实现深拷贝**！ !!

问题：类中是否可以定义`const`成员?

小实验：

下面的类定义是否合法?

如果合法 `ci` 的值是什么，存储在哪里?

```
class Test
{
private:
const int ci;
public :
    //编译器报错，只读变量，不能出现在等号左边
    Test(){
        ci = 10;
    }
int getCI() { return ci;}
} ;

int main(){
    Test t;
    //如何初始化一个const类成员
    printf("t.ci = %d \n", t.getCI());
    //编译器报错，没有初始化
}
```

1.5 初始化列表

类成员的初始化

- C+ +中提供了初始化列表对成员变量进行初始化
- 语法规则
    - ```
      class sName :: ClassName () :
      m1 (v1), m2(v1,v2), m3(v3)
      {
          // some other initialize operation
      }
      ```
    - 上面初始化列表的解释，`::`前为类名，后为构造函数，`:`后分别为构造函数的参数为类成员初始化。
- 注意事项
    - 成员的初始化顺序与成员的声明顺序相同
    - 成员的初始化顺序与初始化列表中的位置无关
    - 初始化列表先于构造函数的函数体执行

```
#include <stdio.h>

class value{
    private:
    //erro,初始化只能用初始化列表
    int mi = 0;
    public:
    value(int i){
        printf("i  = %d\n", i);
```
```

```
        mi = i;
    }
    int getI(){
        return mi;
    }
}
class Test
{
private:
    value m1;
    value m2;
    value m3;
public :
    //变量只能初始化
    Test(){
        ci = 10;
    }
int getCI() { return ci;}
} ;
```

类中的 `const` 成员

- 类中的 `const 成员` 会被分配空间——与当前的对象一起
- 类中的 `const` 成员的本质是只读变量
- 类中的 `const` 成员只能在初始化列表中指定初始值

编译器无法直接获取到 `const` 成员的初始值此无法进入符号表成为真正意义上的常量。

```
class Test
{
private:
    const int i;
public :
    //变量只能初始化
    Test(): i(10){
        ci = 10;
    }
    int getCI() { return ci;}
    //此例子说明类中的const变量不是常量，不过是只读变量，下面的代码会改变其值。
    int setCI(int v){
        int* p = const_cast<int*>(&ci);
        *p = v;
    }
} ;
```

小插曲:

初始化与赋值不同

- 初始化 **正在创建的对象**进行初值设置
- 赋值 **已经存在的对象**进行值设置

1.6 对象的构造顺序

不同类的构造函数可能会相互依赖，会导致对象的构造顺序变得复杂。

问题：C+ +中的类可以定义多个对象，那么对象构造的顺序是怎样的?

- 对于局部对象
  - 当**程序执行流**到达对象的定义语句时进行构造
- 下面程序中的对象构造顺序是什么?

```
int i= 0;
Test a1 = i;   // 0
while (i < 3 )
    Test a2 = ++i; // 1,2,3
if (i < 4 )
    Test a = a1; //4 拷贝构造函数
else
    Test a(100);
```

- 对于堆对象
  - 当**程序执行流**到达 `new` 语句时创建对象
  - 使用 `new` 创建对象将自动触发构造函数的调用
- 下面程序中的对象构造顺序是什么?

```
//看到new就行
int i= 0 ;
Test* a1= new Test(i); //0
while (++i < 10 )
    if(i % 2 )
        new Test(i);  // 1,3,5,7,9
if(i < 4 )
    new Test(*a1);
else
    new Test( 100 ); //100
```

- 对于全局对象
  - 对象的构造顺序是**不确定的**！！！！——Attention
  - 不同的编译器**使用不同的规则确定构造顺序**
  - 我们在开发的时候应避免全局对象的相互依赖，避免灾难性错误

小结:

- 局部对象的构造顺序依赖于程序的执行流
- 堆对象的构造顺序依赖于 `new` 的使用顺序
- 全局对象的构造顺序是不确定的

1.7 对象的销毁

- 生活中的对象都是被初始化后才上市的
- 生活中的对象被销毁前会做一些清理工作：比如回收旧手机

  问题：C++ 中如何清理需要销毁的对象

  一般而言 需要销毁的对象都应该做清理

  解决方案

- 为每个类都提供一个 `public free` 函数
- 对象不再需要时立即调用 `free` 函数进行清理
    - ```
      class Test
      {
          int* p;
      public :
          Test() { p = new int; }
          void free () { delete p; };
      } ;
      ```
    - 这样的方式的问题是很有可能忘记了`free`，从而产生内存泄漏，导致程序非常不稳定。用户只能频繁重启。
- 存在的问题
    - `free` 只是一个普通的函数 必须显示的调用
    - 对象销毁前没有做清理 很可能造成资源泄漏
        - C++编译器是否能够**自动调用某个特殊的函数进行对象的清理?**

析构函数

- C++的类中可以定义一个特殊的清理函数
    - 这个特殊的清理函数叫做析构函数
    - 析构函数的功能与构造函数相反
- 定义：`~ClassName()`
    - 析构函数没有参数也没有返回值类型声明（PS：说明不能被重载）
    - 析构函数在对象销毁时自动被调用
- ```
  class test{
      public:
          test(){

          }
          ~test(){

          }
  }
  int main(){
      //局部变量t被销毁的时候析构函数被调用
      test t;
      //在堆空间创建对象
      test* pt = new test();
      //此次销毁堆空间的对象
      delete pt;
      //局部对象在return前销毁
      return 0;
  }
  ```
- 析构函数的定义准则
    - 当类中 自定义了构造函数 并且构造函数使用了系统资源 （如： 内存申请 文件打开 ），需要自定义析构函数

小结：

- 析构函数是对象销毁时进行清理的特殊函数

- 析构函数在对象销毁时自动被调用
- 析构函数是对象释放系统资源的保障

1.8 神秘的临时对象

有趣的问题：

下面的程序输出什么？ 为什么？

```
class Test {
    int mi;
    //正确的用法：直接用一个普通的函数
    void init(int i){
        mi = i;
    }
public:
    Test ( int i) {
        mi = i;
    }
    Test(){
        Test(0); //直接调用构造函数只生成了临时对象，此条语句相当于没有
    }
    void print(){
        printf("mi = %d\n",mi);
    }
}

int main(){
    Test t; //希望的结果是mi = 0
    //编译器的结果是随机值
    t.print();
    return 0;
}
```

发生了什么

- 程序意图：
  - `Test()` 中以 0作为参数调用 `Test(int i)`
  - 将成员变量 mi 的初始值设置为0
- 运行结果：
  - 成员变量 `mi` 的值为随机值

究竟哪个地方出了问题？

思考

- 构造函数是一个特殊的函数
  - 是否可以直接调用？
  - 是否可以在构造函数中调用构造函数？
  - 直接调用构造函数的行为是什么？

答案

- 直接调用构造函数将产生一个临时对象
- 临时对象的生命周期只有一条语句的时间
- 临时对象的作用域只在一条语句中
- 临时对象是 C++ 中**值得警惕的灰色地带**

在C++里面除了需要注意继承C语言的野实际工程开发中需要人为的避开临时对象指针问题，还要注意临时对象的影响。

```
Test();
Test(10);
//直接调用构造函数，产生两个临时对象
```

编译器的行为

- 现代 C++ 编译器在不影响最终执行结果的前提下，**会尽力减少临时对象的产生！！**

```
Test t = Test(10); //1.生成临时对象；2.用临时对象初始化t对象；3.调用拷贝构造函数。C++为了杜绝性能问题，直接优化为 Test t = 10;将第三步优化掉。
```

小结

- 直接调用构造函数将产生一个**临时对象**
- 临时对象是**性能的瓶颈**，也是 bug 的来原之一
- 现代 C++ 编译器会尽力避开临时对象
- 实际工程开发中需要人为的**避开临时对象**

1.9 经典问题解析

关于析构的疑问

- 当程序中存在多个对象的时候，如何确定这些对象的析构顺序?

单个对象创建时构造函数的调用顺序

- 调用**父类**的构造过程
- 调用**成员变量**的构造函数（调用顺序与声明顺序相同）
- 调用**类自身**的构造函数

析构函数与对应构造函数调用顺序相反！！！

多个对象析构时

- 析构顺序与构造顺序相反

```
#include <stdio.h>

class Member{
    const char* ms;
public:
    Member(const char* s){
        printf("Member(const char* s): %s\n",s);
        ms = s;
    }
    ~Member(){
        printf("~Member():%s\n",ms);
    }
}

class Test{
    Member mA;
    Member mB;

public:
```

```
    Test():mB("mB"), mA("mA"){ //成员变量的构造顺序与初始化列表无关
        printf("Test()\n");
    }
    ~Test(){
        printf("~Test()\n");
    }
}

Member gA("gA");

int main(){
    Test t; //gA
            //没有父类
            //mA
            //mB
            //Test()
    //析构函数：~Test(), mB,mA,gA
    return 0;
}
```

关于析构的答案

对于栈对象和全局对象，类似于入栈与出栈的顺序， **最后构造的对象被最先析构**!

堆对象的析构发生在使用 `delete` 的时候，与 `delete` 的使用顺序相关！

关于 `const` 对象的疑问

`const` 关键字能否修饰类的对象，如果可以， 有什么特性?

- `const` 关键字能够修饰对象
- `const` 修饰的对象为只读对象
- 只读对象的成员变量不允许被改变
- 只读对象是编译阶段的概念，运行时无效

`const` 成员函数的定义：

```
Type ClassName::function(Type p) const
//const成员函数在函数名之后，不是之前!
```

类中的函数声明与实际函数定义中都须带 `const` 关键字

```
const Test t(1); //const对象
t.mj = 1000; //编译出错

int Test::getMi(){
    return mi;
}

t.getMi;//编译出错，const对象只能调用const成员函数
//需要将上面的getMi()改为
int Test::getMi() const{
    return mi;
}
```

关于类成员的疑问

成员函数和成员变量都是隶属于具体对象的吗?

从面向对象的角度

- 对象由**属性（成员变量）**和**方法（成员函数）**构成

从程序运行的角度

- 对象由**数据**和**函数**构成
  - 数据可以位于**堆、栈和全局数据区**
  - 函数只能位于代码段
  - 成员函数可以直接访问类的成员变量，从而可以解释拷贝构造函数。

小结

- 对象的析构顺序与构造顺序相反
- `const` 关键字能够修饰对象，得到只读对象
- 只读对象只能调用 `const` 成员函数
- 所有对象共享类的成员函数
- 隐藏的 `this` 指针用于表示当前对象

2. 类的成员和关系

2.1 类的静态成员变量

成员变量的回顾

- 通过对象名能够访问 `public` 成员变量
- 每个对象的成员变量都是专属的
- 成员变量不能在对象之间共享

新的需求

- 统计在程序运行期间某个类的对象数目
- 保证程序的安全性（不能使用全局变量)
- 随时可以获取当前对象的数目

```
#include <stdio.h>
//可以尝试定义一个全局变量，但是不安全
int gCount = 0;

class Test{
    private:
        int mCount; //采用局部变量，每个对象都有独立的成员变量，不能满足需求
    public:
        Test()
        {
            mCount++;
        }
        ~Test(){
            --mCount;
        }
        int getCount(){
            return mCount;
        }
};

Test gTest;
```

```
int main(){
    Test t1;
    Test t2;
    printf("count = %d\n", gTest.getCount());
    printf("count = %d\n", t1.getCount());
    printf("count = %d\n", t2.getCount());
    return 0;
}
```

静态成员变量

在 C+ +中可以定义静态成员变量

- 静态成员变量**属于整个类所有**
- 静态成员变量的生命期不依赖于任何对象
- 可以**通过类名直接访问**公有静态成员变量
- 所有对象共享类的静态成员变量
- **可以通过对象名访问公有静态成员变量**

静态成员变量的特性

- 在定义时直接通过`static`关键字修饰
- 静态成员变量需要在类外单独分配空间
- 静态成员变量在程序内部位于全局数据区

语法规则：

```
Type ClassName::VarName = value;
```

```
#include <stdio.h>
//可以尝试定义一个全局变量，但是不安全
int gCount = 0;

class Test{
    private:
        static int mCount; //采用局部变量，每个对象都有独立的成员变量，不能满足需求
    public:
        Test()
        {
            mCount++;
        }
        ~Test(){
            --mCount;
        }
        int getCount(){
            return mCount;
        }
};

int Test::cCount = 0;

Test gTest;

int main(){
    Test t1;
    Test t2;
```

```
    printf("count = %d\n", gTest.getCount());
    printf("count = %d\n", t1.getCount());
    printf("count = %d\n", t2.getCount());
    Test* pt = new Test();//此时会增加一个count=4
    delete pt; //对象销毁，count = 3
    return 0;
}
```

小结

- 类中可以通过 `static` 关键定义静态成员变量
- 静态成员变量隶属于类所有
- 每一个对象都可以访问静态成员变量
- 静态成员变量在**全局数据区**分配空间
- 静态成员变量的生命期为程序运行期

2.2 类的静态成员函数

上面的需求满足了吗？没有。

- 随时可以获取当前对象的数目(**Failure**)

如果当前没有一个对象时，怎么打印出没有对象的时候呢？因为上面的代码需要一个对象去调用，所以未能满足需求。

```
//直接通过类名来访问静态成员变量
printf("count = %d\n", Test::ccount);
//但是可能会修改了其中的值
Test::count = 1000;
//此时的结果变成了1000，破坏了安全性
printf("count = %d\n", Test::ccount);
```

问题分析

我们需要什么？

- 不依赖对象就可以访问静态成员变量
- 必须保证静态成员变量的安全性
- 方便快捷的获取静态成员变量的值

静态成员函数

在 C+ +中可以定义静态成员函数

- 静态成员函数是类中特殊的成员函数
- 静态成员函数 整个类所有
- 可以通过类名直接访问公有静态成员函数
- 可以通过对象名访问公有静态成员函数

静态成员函数的定义

- 直接通过`static`关键字修饰成员函数

- ```
  class Test
  {
      public :
          static void Func1() { }
          static int Func2();
  } ;
      int Test: :Func2() {
          return 0 ;
      }
  }
  ```

- ```
  #include <stdio.h>

  class demo{
      private:
          int i;
      public:
          int getI();
          static void StaticFunc(const char* s);
          static void StaticSetI(demo& d, int v);
  };

  int demo::getI(){
      return i;
  }

  void demo::StaticFunc(const char* s){
      printf("StaticFunc:%s \n" ,s );
  }

  void demo::StaticSetI(demo& d, int v){
      //静态成员函数不能直接访问类对象，但是可以通过类对象访问
      d.i = v;
  }

  int main(){
      demo d;
      //通过类名直接调用成员函数
      demo::StaticFunc("start!\n");
      demo::StaticSetI(d, 10);
      printf("d.i= %d\n", d.getI);
      return 0;
  }
  ```

静态成员函数 vs 普通成员函数

	静态成员函数	普通成员函数
所有对象共享	yes	yes
隐含this指针	**no**	yes
访问普通成员变量/函数	**no**	yes
访问静态成员变量/函数	yes	yes
通过类名直接调用	yes	**no**
通过对象名直接调用	yes	yes

满足用户的所有需求：

```
#include <stdio.h>
//可以尝试定义一个全局变量，但是不安全
int gCount = 0;

class Test{
    private:
        static int mCount; //采用局部变量，每个对象都有独立的成员变量，不能满足需求
    public:
        Test()
        {
            mCount++;
        }
        ~Test(){
            --mCount;
        }
        static int getCount(){
            return mCount;
        }
};

int Test::cCount = 0;

Test gTest;

int main(){
    Test t1;
    Test t2;
    printf("count = %d\n", Test::GetCount()); //0
    printf("count = %d\n", t1.getCount());
    printf("count = %d\n", t2.getCount());
    Test* pt = new Test();//此时会增加一个count=4
    delete pt; //对象销毁，count = 3
    printf("count = %d\n", Test::GetCount()); //2
    return 0;
}
```

小结

- 静态成员函数是类中特殊的成员函数
- 静态成员函数没有隐蔽的`this`参数
- 静态成员函数可以通过**类名**直接访问

- 静态成员函数只能直接访问**静态成员变量（函数）**

2.3 二阶构造模式

模式的本质是**方法**，软件设计里面有门课叫设计模式，其实就是经典的设计方法，被前人们总结提炼出来的。

关于构造函数的回顾

- 类的构造函数用于对象的初始化
- 构造函数与类同名并且没有返回值
- 构造函数在对象定义时自动被调用

问题：

1. 如何判断构造函数的执行结果？——没有办法
2. 在构造函数中执行 `return` 语句会发生什么？——不意味着对象的构造成果
3. 构造函数执行结束是否意味着对象构造成功？

```
#include <stdio.h>

class Test{

    private:
        int mi;
        int mj;
    public:
        Test(int i, int j){
            mi = i;
            //当执行到此次的时候直接返回，mj没有被初始化
            return ;
            mj = j;
        }
        int getI(){
            return mi;
        }
        int getJ(){
            return mj;
        }
};

int main(){
    //对象的诞生与构造函数的执行结果没有关系的，目前没有办法捕捉到创建的对象是否能用
    Test t1(1, 2);

    printf("t1.mi = %d\n",t1.getI());
    printf("t1.mj = %d\n",t1.getJ());
}
```

解决的办法可以添加一个`status`。

```
#include <stdio.h>

class Test{

    private:
        int mi;
```

```
        int mj;
        bool mStatus;
    public:
        Test(int i, int j): mStatus(false){
            mi = i;
            //当执行到此次的时候直接返回，mj没有被初始化
            return ;
            mj = j;
            mStatus = true;
        }
        int getI(){
            return mi;
        }
        int getJ(){
            return mj;
        }
};

int main(){
    //对象的诞生与构造函数的执行结果没有关系的，目前没有办法捕捉到创建的对象是否能用
    Test t1(1, 2);
    if(t1.mStatus){
    printf("t1.mi = %d\n",t1.getI());
    printf("t1.mj = %d\n",t1.getJ());
    }
}
```

构造函数

- 只提供自动初始化成员变量的机会
- 不能保证初始化逻辑一定成功
- 执行 `return` 语句后构造函数立即结束

构造函数能决定的只是对象的初始状态 而不是对象的诞生！

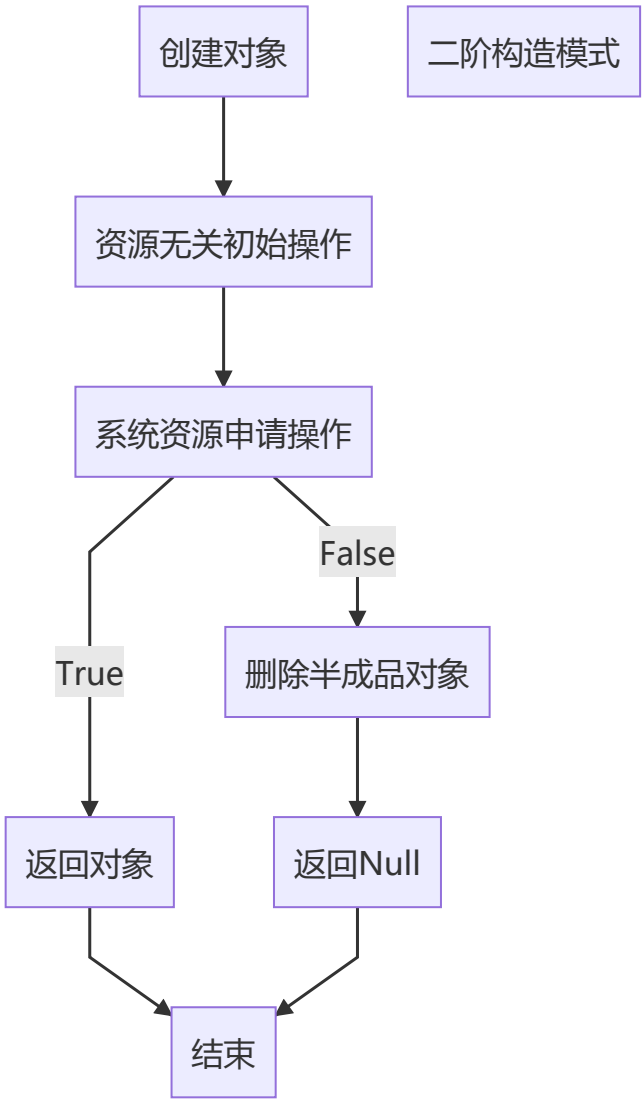
半成品对象

- 半成品对象的概念
  - 初始化操作不能按照预期完成而得到的对象
  - 半成品对象是合法的 **C+ +对象**， 也是 Bug 的重要来源

二阶构造

工程开发中的构造过程可分为

- **资源无关**的初始化操作
  - 不可能出现异常情况的操作
- 需要使用**系统资源**的操作
  - 可能出现异常情况如： 内存申请、访问文件



二阶构造示例一

```
class TwoPhaseCons {
private :
    TwoPhaseCons() { // 第一阶段构造函数
    }
    bool construct() { //第二阶段构造函数
        return true;
    }
public :
        static TwoPhaseCons* NewInstance(); //对象创建函数
};
```

二阶构造示例二

```
TwoPhaseCons* TwoPhaseCons::New nstance() {
    TwoPhaseCons* ret = new TwoPhaseCons();
    //若第二阶段构造失败，返回NULL，ret->construct()执行失败为False
    if(! (ret && ret->construct())) {
        delete ret;
        ret = NULL;
    }
```

```
    return ret;
}

int main(){
    TwoPhaseCons* obj =new TwoPhaseCons::NewInstance();
    printf("obj = %p\n", obj);
}
```

二阶构造创建的对象是放在堆上面的，正好符合工程上的大对象。

小结

- 构造函数只能决定对象的初始化状态
- 构造函数中初始化操作的失败不影响对象的诞生
- 初始化不完全的半成品对象是 Bug 的重要来源
- 二阶构造人为的将初始化过程分为两部分
- 二阶构造能够确保创建的对象都是完整初始化的

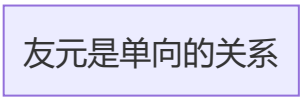
2.4 友元的尴尬能力

友元这种看似超魔法的能力如今在工程开发中逐渐遗弃，为什么呢?

友元的概念

什么是友元?

- 友元是 C+ + 中的 一种关系
- 友元关系发生在函数与类之间或者类与类之间
- 友元关系是单项的，不能传递



友元的用法

- 在类中以 `friend` 关键字声明友元
- 类的友元可以是**其它类或者具体函数**
- 友元**不是**类的一部分
- 友元**不受类中访问级别的限制**
- 友元**可以直接访问具体类的所有成员**

在类中用 `friend` 关键字对函数或类进行声明

```
class Point
{
    double x;
    double y;
```

```
public:
    Point(double x, double y){
        this->x = x;
        this->y = y;
    }
    int getX(){
        return x;
    }
      int getY(){
        return y;
    }
    //此处表明func是class Point 的友元，那么func函数可以无限制对class Point进行访问
    friend void func(Point& p);
} ;

void func(Point& p)
{
    double ret = 0;
    //此处如果不用友元，只有用p1.getI()，但需要调用八次get*()函数
    ret = (p2.y - p1.y) * (p2.y - p1.y)+
        (p2.x - p1.x) * (p2.x - p1.x)+
    ret = sqrt(ret);
}

int main(){

   Point p1(1,2);
   Point p2(10,20);

   printf("p1(%d, %d)\n", p1.getX(), p1.getY());

}
```

友元的尴尬

- 友元是为了**兼顾C语言**的**高效**而诞生的
- 友元**直接破坏**了面向对象的封装性
- 友元在实际产品中的**高效得不偿失的**
- 友元在现代软件工程中**已经逐渐被遗弃**

注意事项

- 友元关系不具备传递性
- 类的友元可以是其它类的成员函数
- 类的友元可以是某个完整的类
    - 所有的成员函数都是友元

```
class Point
{
    double x;
    double y;
public:
    Point(double x, double y){
        this->x = x;
        this->y = y;
    }
```

```
    //Point2是Point的友元
    friend class Point2;
} ;
class Point2
{
    double x;
    double y;
public:
    Point(double x, double y){
        this->x = x;
        this->y = y;
    }
    friend class Point3;
} ;
class Point3
{
    double x;
    double y;
public:
    Point(double x, double y){
        this->x = x;
        this->y = y;
    }
    friend void func(Point& p);
} ;
```

2.5 类中的函数重载

函数重载回顾

- 函数重载的**本质为相互独立的不同函数**
- C+ + 中通过**函数名**和**函数参数**确定函数调用
- 无法直接通过**函数名**得到重载函数的入口地址
- 函数重载**必然发生在同一个作用域中**——特别重要

类中的重载

- 类中的成员函数可以进行重载
    - 构造函数的重载
    - 普通成员函数的重载
    - 静态成员函数的重载

问题

全局函数、普通成员函数以及静态成员函数之间是否可以构成重载?

1. 重载函数的本质为多个不同的函数
2. 函数名和参数列表是唯一的标识
3. 函数重载必须发生在同一个作用域中

```
#include <stdio.h>

void func(){
    printf("void func()\n");
}

void func(int i){
```

```
    printf("void func(int i), i = %d\n",i);
}

class Test{
    int i;
public:
    Test(){
         printf("Test::Test\n");
        this->i = 0;
    }
    Test(int i){
        printf("Test::Test(int i)\n");
        this->i = i;
    }
    Test(const Test& obj){
        printf("Test::Test(Test& obj)\n");
        this->i = obj.i;
    }
    static void func(){
        printf("Test::func()\n");
    }
    void func(int i){
    printf("void TEst::func(int i), i = %d\n",i);
    }
    int getI(){
        return i;
    }
};

int main{
    //全局函数的重载
    func();
    func(1);
    Test t; //Test::Test
    Test t1(1); //Test::Test(int i)
    Test t2(t1);// Test::Test(Test& obj)
    func();  //void func()
    Test::func();//Test::func()
    func(2);
    //类的成员函数是可以构成重载的
    t1.func(2);
    t1.func();
    return 0;
}
```

重载的意义

- 通过函数名对函数功能进行提示
- 通过**参数列表**对函数用法进行提示
- 扩展系统中已经存在的函数功能

```
#include <stdio.h>

//C++里面直接采用重载
char* strcpy(char* buf, const char* str, unsigned int n){
    return strnpy(buf, str, n);
}
```

```
int main(){
    const char* s = "LIu";
    char buf[10] ={0};
    //在C语言里面需要记忆一个新的函数名
    strncpy(buf, s, sizeof(buf) -1);
    //容易越界
    strcpy(buf, s);

    printf("%s\n", buf);
}
```

重载能够扩展系统中已经存在的函数功能，那么重载是否也能够扩展其它更多的功能？使其更加强大。

思考

- 下面的复数解决方案是否可行？——后面讲述如何解决

```
class Complex
{
public :
    int a;
    int b;
};

int main()
{
    //想通过重载扩展+法功能，但是暂时没法编译
    Complex c1= { 1 , 2 };
    Complex c2 = { 3 , 4 };
    Complex c3 =c1 + c2;
    return 0 ;
}
```

小结

- 类的成员函数之间可以进行重载
- 重载必须发生在同一个作用域
- 全局函数 和成员函数不能构成重载关系
- 重载的意义在于扩展已经存在的功能

2.6 操作符重载的概念

初探

```
class Complex
{
public :
    int a;
    int b;
};
//此次可以a,b被直接暴露，我们可以通过友元来实现，但也不完美
Complex Add(const Complex& p1. const Complex& p2){
    Complex ret;
    ret.a = p1.a + p2.b;
    ret.b = p1.b + p2.b;
}
```

```
int main()
{
    Complex c1= { 1 , 2 };
    Complex c2 = { 3 , 4 };
    Complex c3 =c1 + c2;
    return 0 ;
}
```

`Add` 函数可以解决 `Complex` 对象相加的问题，但是 `Complex` 是现实世界中确实存在的复数并且复数在数学中的地位和普通的实数相同。

为什么不能让 `+` 操作符也支持复数相加呢?

操作符重载

- C+ +中的重载能够**扩展操作符的功能**
- 操作符的重载以**函数的方式进行**
- 本质:
  - 用特殊形式的函数扩展操作符的功能
- 通过 `operator` 关键字可以定义特殊的函数
- `operator` 的本质是通过函数重载操作符
- 语法:
  - ```
    Type operator Sign( const Type& pl, const Type& p2)
    {
        Type ret;
        return ret;
    }
    //Sign 为系统预定义的操作符，如:+-*/等
    ```

```
class Complex
{
private :
    int a;
    int b;
public:
    friend Complex operator + (const Complex& p1. const Complex& p2);
};
Complex operator +(const Complex& p1. const Complex& p2){
    Complex ret;
    ret.a = p1.a + p2.b;
    ret.b = p1.b + p2.b;
    return ret;
}
int main()
{
    Complex c1= { 1 , 2 };
    Complex c2 = { 3 , 4 };
    Complex c3 =c1 + c2; //operator +(c1, c2)
    return 0 ;
}
```

可以将操作符重载函数定义为类的成员函数

- 比全局操作符重载函数少一个参数 （左操作数）
```

- **不需要依赖友元**就可以完成操作符重载
- **编译器优先在成员函数中寻找操作符重载函数**

```
class Type
{
public :
    /*
    Type operator Sign( const Type& p1, const Type& p2)
    {
        Type ret;
        return ret;
    }
    对比下面的版本
    */
    Type operator Sign(const Type & p)
    {
        Type ret;
        return ret;
    }
} ;
```

```
class Complex
{
private :
    int a;
    int b;
public:
   Complex operator +(const Complex& p){
    //此时的左操作数变成成员函数
    Complex ret;
    ret.a = this->a + p.b;
    ret.b = this->b + p.b;
    return ret;
    }
};

int main()
{
    Complex c1= { 1 , 2 };
    Complex c2 = { 3 , 4 };
    Complex c3 = c1 + c2; //c1.operator+(c2)
    return 0 ;
}
```

小结

- 操作符重载 是C+ +的强大特性之一
- 操作符重载的**本质** 通过函数扩展操作符的功能
- `operator` 关键字是实现操作符重载的关键
- 操作符重载遵循相同的函数重载规则
- 全局函数和成员函数都可以实现对操作符的重载

2.7 完善的复数类

复数类应该具有的操作

- 运算：+-*/
- 比较：== !=
- 赋值：=
- 求模：modulus

利用操作符重载

- 统一复数与实数的运算方式
- 统一复数与实数的比较方式
- ```
  Complex operator + (const Complex& c);
  Complex opera or - (const Complex& c);
  Complex operator * (const Complex& c);
  Complex operator / (const Complex& c);
  bool operator == (const Complex& c);
  bool operator != (const Complex& c);
  Complex& operator = (const Complex& c);
  ```

`Complex.h`:

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

class Complex{
    double a;
    double b;
public:
        Complex (double a = 0, double b = 1);
        double getA();
        double getB();
        double getModulus();

        Complex operator + (const Complex& c);
        Complex opera or - (const Complex& c);
        Complex operator * (const Complex& c);
        Complex operator / (const Complex& c);
        bool operator == (const Complex& c);
        bool operator != (const Complex& c);
        Complex& operator = (const Complex& c);
    }
};
```

`Complex.cpp`:

```
#include "Complex.h"
#include "math.h"
Complex::Complex(double a = 0, double b = 1){
    this->a = a;
    this->b = b;
}
double Complex::getA(){
    return a;
```
```

```
}
double Complex::getB(){
    return b;
}
double Complex::getModulus(){
    return sqrt(a * a + b * b);
}


Complex Complex::operator + (const Complex& c){
    double na = a + c.a;
    double nb = b + c.b;
    Complex ret(na, nb);
    return ret;
}
Complex Complex::operator - (const Complex& c){
    double na = a - c.a;
    double nb = b - c.b;
    Complex ret(na, nb);
    return ret;
}
Complex Complex::operator * (const Complex& c){
    //Complex Multiple : (a+bi)(c+di) = (ac - bd) + (bc + ad)i
    double na = a * a.c - b* c.b;
    double nb = b * a.c - a* c.d;
    Complex ret(na, nb);
    return ret;
}
Complex Complex::operator / (const Complex& c){
    //Complex div :  (a+bi)/(c+di) = (ac+bd)/(c^2+ d^2)+ (bc-ad)/(c^2+d^2)i
    double cm = c.a * c.a + c.b * c.b
    double na = (a * a.c + b* c.b) / cm;
    double nb = (b * a.c - a* c.d) /cm;
    Complex ret(na, nb);
    return ret;
}
bool Complex::operator == (const Complex& c){
    return (a == c.a ) && (b == c.b);
}
bool Complex::operator != (const Complex& c){
    return !(*this ==c);
}
Complex& Complex::operator = (const Complex& c){
    if( this != &c){
        a = c.a;
        b = c.b;
    }
    //为什么返回的是左操作数呢？因为为了连续赋值
    return *this;
}
```

`main.cpp`:

```
#include "Complex.h"
#include "math.h"
#include <stdio.h>

int main(){
    Complex c1(1,2);
    Complex c1(3,5);
    //演算是否正确
    Complex c3 = c2 - c1;
    return 0;
}
```

注意事项

- C+ +规定**赋值操作符** `(=)` 只能**重载为成员函数**
- 操作符重载不能改变原操作符的**优先级**
- 操作符重载不能改变操作数的**个数**
- 操作符重载不应改变操作符的**原有语义**
- 操作符重载本质为**函数定义**

2.8 初探C++标准库

操作符 `<<` 的原生意义是按位左移，例:`1<<2`，其意义是将整数 1按位左移2位，即:

`0000 0001 》 00000100`

重载左移操作符， 将变量或常量左移到一个对象中!

```
#include <stdio.h>

const char endl ='\n';

class Console{
 public:
    Console& operator << (int i){
        printf("%d", i);
        return *this;
    }
    Console& operator << (char c){
        printf("%c", c);
        return *this;
    }
    Console& operator << (const char* c){
        printf("%s", c);
        return *this;
    }
    Console& operator << (double d){
        printf("%f", d);
        return *this;
    }
};

Console cout;

int main(){
    cout << 1;
    cout << endl;
```

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

重复发明轮子并不是一件有创造性的事，站在巨人的肩膀上解决问题会更加有效！

C+ +标准库

- C+ +标准库并不是 C+ +语言的一部分
- C+ +标准库是由类库和函数库组成的集合
- C+ +标准库中定义的类和对象都位于 `std` 命名空间中
- C+ +标准库的头文件都不带 `.h` 后缀
- C+ +标准库涵盖了C库的功能

C++ **编译环境的组成**

- C++标准库
- **C语言兼容库**
- 编译器扩展库
- C++标准语法库
- C++扩展语法库

C++标准库预定义了多数常用的数据结构

`<bitset><deque><list><queue><set><stack><vector><map><cstdio><cstring><cstdlib><cmath>`

C++的输入输出

```
#include <iostream>

using namespace std;

int main(){
    double a;
    cout << "Hello world\n" << endl;
    cout << "Input a:";
    cin >> a;
}
```

小结

- C+ +标准库是由类库和函数库组成的集合
- C+ +标准库包含经典算法和数据洁构的实现
- C+ +标准库涵盖了C库的功能
- C+ +标准库位于`std`命名空间中

2.9 C++中的字符串类

在C语言中，没有字符串的数据类型，使用的是**字符数组**来模拟字符串。

历史遗留问题

- C语言 不支持真正意义上的字符串
- C语言用字符数组 和一组函数实现字符串操作
- C语言不支持自定义类型， 因此无法获得字符串类型

解决方案

- 从C到C++的进化过程引入了自定义数据类型
- 在C++中可以通过类完成字符串类型的定义

问题：

C++中的原生类型系统是否包含字符串类型？

标准库中的字符串类

- C++语言直接支持C语言的所有概念
- C++语言中没有原生 的字符串类型
- C++标准库提供了 `string` 类型
  - `string` 直接支持字符串连接
  - `string` 直接支持字符串的大小比较
  - `string` 直接支持子串查找和提取
  - `string` 直接支持字符串的插入和替换

```
#include <iostream>
#include <string>

using namespace std;

void string_sort(string a[], int len){
    //select sort
    for(int = 0; i < len, i++){
        for(int j = i; j < len; j++){
            if(a[i] > a[j]){
                swap(a[i], a[j]);
            }
        }
    }
}

void string_add(string a[], int len){
    string ret = "";
    for(int i = 0; i <len; i++){
        ret += a[i] +";";
    }
    return ret;
}
int main(){
    string sa[7] = {
        "Hello",
        "LIU",
        "Java",
        "C++",
        "Python",
        "Typescript"
    };
    string_sort(sa, 7);
    for(int i=0; i < 7; i++){
        cout << sa[i] << endl;
    }
    return 0;
}
```

字符串和数字的转换

- 标准库中提供了相关的类对字符串和数字进行转换
- 字符串流类 `sstream` 用于 `string` 的转换
  - `<sstream>` —相关头文件
  - `istringstream` —字符串输入流
  - `ostringstream` —字符串输出流

使用方法：

- `string` -> 数字
  ```
  istringstream iss("123.45");
  double num;
  iss >> num;
  ```
- 数字 -> `string`
  ```
  ostringstream oss;
  oss << 543.21;
  string s = oss.str();
  ```

```
bool to_number(const string&, int& n){
    istringstream iss(s);
    return iss >> n;
}

bool to_number(const string&, double& n){
    istringstream iss(s);
    return iss >> n;
}
...
//针对不同类型，可以采用后面学习的模板技术
```

问题：**字符串循环右移**

- `abcdefg` 循环右移 3位后得到 `efgabcd`
- ```
  #include <iostream>
  #include <string>

  using namespace std;

  string right_func(const string& s, unsigned int n){
      string ret = "";
      unsigned int pos = 0;
      //不论右移多少位都是合法
      n = n % s.length();
      pos = s.length() - n;

      ret = s.substr(pos);
      ret += s.substr(0, pos);
  }

  //操作符重载
  ```
  ```

```
string operator >> (const string& s, unsigned int n){
    string ret = "";
    unsigned int pos = 0;
    //不论右移多少位都是合法
    n = n % s.length();
    pos = s.length() - n;

    ret = s.substr(pos);
    ret += s.substr(0, pos);
}

int main(){
    string r = right_func("abcdefg", 8);
    string r = (s >> 3);
    return 0;
}
```

小结

- 应用开发中大多数的清况都在进行字符串处理
- C+ +中 没有直接支持原生的字符串类型
- 标准库中通过 `string` 类支持字符串的概念
- `string` 类支持字符串数字的相互转换
- `string` 类的应用使得问题的求解变得简单

课后练习

- 字符串反转
  - 要求:
    - 使用`string` 类完成
  - 示例:
    - `"we;tonight;you" ➔ "ew;thginot;uoy"`
  - 提示:
    - `string` 类中提供了成员函数可以查找目标字符的位置
  - ```
    #include <iostream>
    #include <string>
    #include <algorithm>

    using namespace std;
    string Reverse(const string& s,const char c)
    {
        unsigned int head = 0;
        unsigned int pos = 0;
        string ret, ssub;
        while((pos = s.find(c,head))!=string::npos)
        {
            ssub = s.substr(head,pos-head+1);
            reverse(ssub.begin(),ssub.end()-1);
            ret += ssub ;
            head = pos+1;
        }
        if(head<s.length())
        {
            ssub = s.substr(head,s.length()-head+1);
            reverse(ssub.begin(),ssub.end());
    ```
    ```

```
            ret += ssub;
        }
        return ret;
    }

    int main(){
        cout << Reverse("", ';') << endl; //null
        cout << Reverse(";", ';') << endl; //;
        cout << Reverse("abcd;", ';') << endl; //dcab;
        cout << Reverse("we;tonight;you", ';') << endl; //ew;thginot;uoy
        return 0;
    }
```

2.10 数组操作符的重载

问题：

`string` 类对象还具备方式字符串的灵活性吗？ **还能直接访问单个字符吗?**

字符串类的兼容性

- `string`类最大限度的考虑了C字符串的兼容性；
- 可以按照使用 C字符串的方式使用 `string` 对象
- ```
  string s= "alb2c3d4e" ;
  int n = 0;

  for (int i = 0; i < s.length(); i++)
  {
      if(isdigit(s[i] ))
      {
          n++;
      }
  }
  ```

问题

类的对象怎么支持数组的下标访问?

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;
class test{

};
int main(){
    test t[0]; //erro
    return 0;
}
```

被忽略的事实。。。

- 数组访问符是 C/C+ +中的内置操作符
```

- 数组访问符的原生意义是**数组访问**和**指针运算**
  - `a[n]<->*(a+n)<->*(n+a)<->n[a]`

数组访问操作符（ [] ）

- 只能通过类的成员函数重载
- 重载函数能仅能使用一个参数
- 可以定义不同参数的多个重载函数
- ```
  #include <iostream>
  #include <string>
  #include <algorithm>

  using namespace std;
  class test{
      int a[5];
      public:
          int& operator [] (int i){
              return a[i];
          }
          int& operator [] (const char* s){
              if( s == "lst")
                  return a[0];
              else if
              ...
          }
      int length(){
          return 5;
      }
  };

  int main(){
      test t;
      cout << t[1] << endl;
      for(int i = 0; i < t.length(); i++){
          t[i] = i;
      }
      for(int i =0; i < t.length(); i++){
          cout << t[i] << endl;
      }
      return 0;
  }
  ```

- 数组类重写
- `IntArray.h`:
- ```
  #ifndef _INTARRAY_H_
  #define _INTARRAY_H_

  class IntAarry{
      int m_length;
      int* m_pointer;

      IntArray(int len);
      IntArray(const IntArray& obj);
      bool construct();
  ```

```
public:
    static IntArray* NewInstance(int length);
    int length();
    bool get(int index, int& value);
    bool set(int index, int value);
    int& operator [] (int index);
    IntArray& self();
    ~IntArray();
}

#endif
```

- `IntAarry.cpp`:
- ```
  #include <iostream>

  using namespace std;
  IntArray::IntArray(int len){
      m_length = len;
  }

  bool IntArray::construct(){
      bool ret = true;
      m_pointer = new int[m_length];

      if(m_pointer){
          for(int i = 0; i < m_length; i++){
              m_pointer[i] = 0;
          }
      }
      else{
          ret = false;
      }
      return ret;
  }
  IntArray* IntArray::NewInstance(int length){
      IntArray* ret = new IntArray(length);

      if(!(ret && ret->construct() )){
          delete ret;
      }
      return ret;
  }

  int IntArray::length(){
      return m_length;
  }

  bool IntArray::get(int index, int& value){
      bool ret = (0 <= index) && (index < length());

      if(ret){
          value = m_pointer[index];
      }
      return ret;
  }
  ```
  ```

```
bool IntArray::set(int index, int value){
     bool ret = (0 <= index) && (index < length());
     if(ret){
         m_pointer[index] = value;
     }
}

int& IntArray::operator [] (int index){

}
IntArray::~IntArray(){
    delete [] m_pointer;
}
IntArray::IntArray& self(){
    return *this;
}

int main(){
    //有什么办法替代指针呢？后面的智能指针可以替代
    IntArray* a = IntArray::NewInstance(5);
    if( a != NULL){
        IntArray& array = a->self();
         //(*a)[0] = 1;
        array[0] = 1;
        for(int i = 0; i < a -> length(); i++){
            cout <<array[i] <<endl;
        }
    }

    delete a;
    return 0;
}
```

小结

- 数组访问符的重载能够使得对象模拟数组的行为
- 只能通过类的成员函数重载数组访问符
- 重载函数能且仅能使用一个参数

2.11 函数对象分析

客户需求

- 编写一个函数
  - 函数可以获得斐皮那契数列每项的值
  - 每调用一次返回一个值
  - 函数可根据需要重复使用
- ```
  for(int i = 0; i<10 ; i++)
  {
      cout << fib() << endl;  //带状态函数
  }
  ```

```
#include <iostream>
#include <string>
#include <algorithm>
```
```

```
using namespace std;
int fib(){
    static int a0 = 0;
    static int a1 = 1;
    int ret = a1;
    a1 = a0 + a1;
    a0 = ret;
    return ret;
}
int main(){
    for(int i = 0; i<10 ; i++) {
        cout << fib() << endl;  //带状态函数
    }
    //但是不能重来，一旦开始不能重启
    return 0;
}
```

存在的问题

- 函数一旦开始调用就无去重来
    - 静态局部变量处于函数内部外界无法改变
    - 函数为全局函数是唯一的，无法多次独立使用
    - 无法指定某个具体的数列项作为初始值

函数对象

- **使用具体的类对象取代函数**
- 该类的对象具备**函数调用**的行为
- 构造函数指定**具体数列项的起始位置**
- **多个对象相互独立的求解数列项**

函数调用符(())

- 只能通过类的成员函数重载
- 可以定义不同参数的多个重载函数

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;
class Fib{
        int a0;
        int a1;
public:
    int operator()(){
        int ret = a1;
        a1 = a0 + a1;
        a0 = ret;
        return ret;
    }
    Fib(int n){
        a0 = 0;
        a1 = 1;
        //从第n项开始
        for(int i = 2 ; i<= n; i++){
            int t = a1;
```

```
            a1 = a0 + a1;
            a0 = t;
        }
    }
}

int main(){
    Fib fib;
    for(int i = 0; i<10 ; i++) {
        cout << fib() << endl;  //带状态函数
    }
    Fib fib2;
    for(int i = 0; i<10 ; i++) {
        cout << fib2() << endl;  //此时可以重新开始
    }
    return 0;
}
```

小结

- 函数调用操作符（ () ）是可重载的
- 函数调用操作符只能通过类的成员函数重载
- 函数调用操作符可以定义不同参数的多个重载函数
- 函数对象用于在工程中取代函数指针

2.12 经典问题解析

关于赋值操作符的疑问

- 什么时候需要重载赋值操作符，编译器是否提供默认的赋值操作?
- 编译器为每个类默认重载了赋值操作符
- 默认的赋值操作符仅完成浅拷贝
- 当需要进行深拷贝时必须重载赋值操作符
- **赋值操作符和拷贝构造函数**有相同的存在意义

```
//返回引用和this，参数为const Test& ; 将指针指向的内容全部复制到新空间
Test& operator = (const Test& obj){
    //处理自赋值的情况
    if (this != &obj){
        delete m_pointer;
        m_pointer = new int (*obj.m_pointer);
    }
    return *this;
}
```

编译器提供的函数

```
class Test{

};
//与下面的等价
class Test{
    public:
        Test();
        Test(const Test&);
        Test& operator = (const Test&);
        ~Test();
};
```

关于`string`的疑问

下面的代码输出什么？为什么？

```
string s= " 12345" ;
const char* p = s.c_str() ;  //12345
cout << p << endl; //p成为野指针
s.append (" abced" ) ; //12345
cout << p << endl ;
```

3. 类的关系

3.1 智能指针分析

永恒的话题

- 内存泄漏（臭名昭著的Bug）
  - 动态申请堆空间，用完后不归还
  - C++语言中没有垃圾回收的机制
  - 指针无法控制所指堆空间的生命周期

```
#include <iostream>
#include <string>

using namespace std;
class Test{
    int i;
public:
    Test(int i){
        this-> = i;
    }
    int value(){
        return i;
    }
    ~Test(){

    }
}
int main(){
    //当i到上万次的时候，可能就会导致内存泄漏
    //p 是局部变量，程序结束后，p销毁了，但堆空间还在
    for(int i = 0; i < 5; i++){
        Test* p = new Test(i);
        cout << p->value() << endl;
    }
    return 0;
}
```

深度的思考

我们需要什么

- 需要一个特殊的指针
- 指针生命周期结束时主动释放堆空间
- 一片堆空间最多只能由一个指针标识
- 杜绝指针运算和指针比较

解决方案

- 重载指针特征操作符（->和*）
- 只能通过类的成员函数重载
- 重载函数**不能**使用参数
- 只能定义**一个重载函数**

使用对象替代指针——智能指针

```
#include <iostream>
#include <string>

using namespace std;
class Test{
    int i;
public:
    Test(int i){
        this-> = i;
    }
    int value(){
        return i;
    }
    ~Test(){

    }
}

class pointer{
    Test* mp;
public:
    pointer(Test* p = NULL){
        mp = p;
    }

    Test* operator ->(){
        return mp;
    }

    pointer(const pointer& obj){
        mp = obj.mp;
        const_cast<pointer&>(obj).mp = NULL;
    }

    pointer& operater = (const pointer& obj){
        if (this != &obj){
        mp = obj.mp;
        obj.mp = NULL;
    }
        return *this;
    }

    Test& operator *(){
        return *mp;
    }

    ~pointer(){
        delete mp;
    }
}


int main(){
    for(int i = 0; i < 5; i++){
        pointer p = new Test(i);
```

```
        cout << p->value() << endl;
    }
    return 0;
}
```

智能指针的使用军规

- **只能用来指向堆空间中的对象或者变量!!**

小结

- 重载指针特征符能够使用对象代替指针
- 智能指针只能用于指向堆空间 的内存
- 智能指针的意义在于最大程度的避免内存问题

3.2 逻辑操作符的陷阱

逻辑运算符的原生语义

- 操作数只有两种值 `true` 和 `false`
- 逻辑表达式不用完全计算就能确定最终值
- 最终结果只能是 `true` 或者 `false`

```
#include <iostream>
#include <string>

using namespace std;

int func(int i){
    cout << "int i" << i <<endl;
    return i;
}
int main(){
    int a = 0;
    int b = 1;
    //func(b)不用被调用
    if (func(a) && func(b)){
        cout << "result = T" << endl;
    }
    else
        cout << "result = F" << endl;
    //都会被调用
    if (func(a) || func(b)){
        cout << "result = T" << endl;
    }
    else
        cout << "result = F" << endl;
    return 0;
}
```

重载逻辑操作符

逻辑操作符可以重载吗？重载逻辑操作符 什么意义？

```
#include <iostream>
#include <string>
```

```
using namespace std;

class Test{
    int mvalue;
public:
    Test(int v){
        mvalue = v;
    }
    int value() const{
        return mvalue;
    }
};

bool operator && (const Test& l, const Test& r){
    return l.value() && r.value;
}

bool operator || (const Test& l, const Test& r){
    return l.value() || r.value;
}

int main(){
        int a = 0;
    int b = 1;
    //重载后两个都会被调用
    if (func(a) && func(b)){
        cout << "result = T" << endl;
    }
    else
        cout << "result = F" << endl;
    //而且调用顺序是相反的
    //operator || (func(a), func(b))
    //操作符的本质是函数调用
    if (func(a) || func(b)){
        cout << "result = T" << endl;
    }
    else
        cout << "result = F" << endl;
    return 0;
}
```

问题的本质分析

1. C+ +通过**函数调用扩展操作符**的功能
2. 进入**函数体前必须完成所有参数的计算**
3. 函数参数的计算次序是**不定的**
4. **短路法则完全失效**

逻辑操作符重载后无法完全实现原生的语义

一些有用的建议

- 实际工程开发中避免重载逻辑操作符
- 通过重载比较操作符代替逻辑操作符重载
- 直接使用成员函数代替逻辑操作符重载
- 使用**全局函数**对逻辑操作符进行重载

小结

- C+ +从语法上支持逻辑操作符重载
- 重载后的逻辑操作符不满足短路法则
- 工程开发中不要重载逻辑操作符
- 通过重载比较操作符替换逻辑操作符重载
- 通过专用成员函数替换逻辑操作符重载

3.3 逗号运算符的分析

工程中不要重载逗号操作符!

3.4 前置操作符和后置操作符

值得思考的问题

- 下面的代码有没有区别？为什么？
  - ```
    i++; //i的值作为返回值，i自增1
    ++i; //i自增1，i的值作为返回值
    ```
  - ```
    //i++
    eax,dword ptr[i]
    eax,1
    dword ptr[i]
    //++i
    ecx,dword ptr[i]
    ecx,1
    dword ptr[i]
    //工程编译器实现没有区别
    ```

意想不到的事实

- 现代编译器产品会对代码进行优化
- 优化使得最终的二进制程序更加高效
- 优化后的二进制程序丢失了 C/C+ +的原生语义
- 不可能从编译后的二进制程序还原 C/C+ +程序

+ +操作符重载

- 全局函数和成员函数均可进行重载
- 重载前置+ +操作符不需要额外的参数
- 重载后置+ +操作符需要一个 `int` 类型的占位参数

```
#include <iostream>
#include <string>

using namespace std;

class test{
    int mValue;
public:
    test(int i){
        mValue = i;
    }
    int value(){
        return mValue;
```

```
    }
    test& operator ++(){
        ++mValue;

        return *this;
    }

    test operator ++(int){
        //后置需要保存当前的值
        test ret(mValue);
        mValue++;

        return ret;
    }
};
int main(){
    test t(0);
    test tt = ++t;
    //前置的++不用再栈上生成对象，不用调用构造函数和析构函数
    ++t;
    return 0;
}
```

`complex.h`:

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H_

class Complex{
    double a;
    double b;
public:
        Complex (double a = 0, double b = 1);
        double getA();
        double getB();
        double getModulus();

        Complex operator + (const Complex& c);
        Complex opera or - (const Complex& c);
        Complex operator * (const Complex& c);
        Complex operator / (const Complex& c);
        bool operator == (const Complex& c);
        bool operator != (const Complex& c);
        Complex& operator = (const Complex& c);
        Complex& operator ++();
        Complex operator ++(int i);
    }
};
```

`complex.cpp`:

```
#include "Complex.h"
#include "math.h"
Complex::Complex(double a = 0, double b = 1){
    this->a = a;
    this->b = b;
```

```
}
double Complex::getA(){
    return a;
}
double Complex::getB(){
    return b;
}
double Complex::getModulus(){
    return sqrt(a * a + b * b);
}


Complex Complex::operator + (const Complex& c){
    double na = a + c.a;
    double nb = b + c.b;
    Complex ret(na, nb);
    return ret;
}
Complex Complex::operator - (const Complex& c){
    double na = a - c.a;
    double nb = b - c.b;
    Complex ret(na, nb);
    return ret;
}
Complex Complex::operator * (const Complex& c){
    //Complex Multiple : (a+bi)(c+di) = (ac - bd) + (bc + ad)i
    double na = a * a.c - b* c.b;
    double nb = b * a.c - a* c.d;
    Complex ret(na, nb);
    return ret;
}
Complex Complex::operator / (const Complex& c){
    //Complex div :  (a+bi)/(c+di) = (ac+bd)/(c^2+ d^2)+ (bc-ad)/(c^2+d^2)i
    double cm = c.a * c.a + c.b * c.b
    double na = (a * a.c + b* c.b) / cm;
    double nb = (b * a.c - a* c.d) /cm;
    Complex ret(na, nb);
    return ret;
}
bool Complex::operator == (const Complex& c){
    return (a == c.a ) && (b == c.b);
}
bool Complex::operator != (const Complex& c){
    return !(*this ==c);
}
Complex& Complex::operator = (const Complex& c){
    if( this != &c){
        a = c.a;
        b = c.b;
    }
    //为什么返回的是左操作数呢？因为为了连续赋值
    return *this;
}

Complex& operator ++(){
    a = a + 1;
    b = b + 1;
    return *this;
```

```
}
Complex operator ++(int i){
    Complex ret(a, b);
    a = a + 1;
    b = b + 1;
    return ret;
}
```

真正的区别

- 对于基础类型的变量
  - 前置++的效率与后置++的效率基本相同
  - 根据项目组骗码规范进行选择
- 对于类类型的对象
  - 前置++的效率高于后置++
  - 尽量使用前置++操作符提高程序效率

小结

- 编译优化使得的可执行程序 高效
- 前置++操作符和后置++操作符都可以被重载
- ++操作符的重载必须符合其原生语义
- 对于基础类型 前置++与后置++的效率几乎相同
- 对于类类型 前置++的效率高于后置++

3.5 类型转换函数

再论类型转换

- 标准数据类型之间会进行隐式的类型安全转换
- 转换规则由小字节到大字节是安全的
- `char->(short -> int ) ->unsigned int -> long -> unsigned long -> float -> double`

```
#include <iostream>
#include <string>

using namespace std;

int main(){
    short s = 'a';
    unsigned int ui = 1000; //默认为int
    int i = -2000;
    double d = i; //现在i为double
    //(unsigne int) i + unsigned ui
    //尽量避免隐士类型转换，容易出错
    if(ui + i > 0){
        cout << "positive" << endl;
    }
    else{
        cout << "Negetive" << endl;
    }
    //下面的输出为4字节，为int + int，为了高效计算，将short 和 char都->int，编译器的隐士类型转换非常容易导致程序出错
    cout << "sizeof(s + 'b') = " << sizeof(s + 'b') << endl;
```

```
    return 0;
}
```

问题

- 普通类型与类类 之间能否进行类型转换？类类 之间能否进行类型转换？

```
#include <iostream>
#include <string>

using namespace std;
class Test{

    public:
        Test(){

        }
        Test(int i){

        }
};
int main(){
    Test t;
    t = Test(5);
    return 0;
}
```

再论构造函数

- 构造函数可以定义不同类型的参数
- 参数满足下列条件时称为转换构造函数(参数只要不是当前类型就行)
    - 有且仅有一个参数
    - 参数是基本类型
    - 参数是其它类类型

另一个视角

- 旧式的C方式强制类型转换
- ```
  int i;
  Test t;
  i = int(1.5) ;
  t = Test (100);
  ```

编译器的行为

- 编译器会尽力尝试让源码通过编译
- ```
  Test t;
  t = 100 ;
  ```
- 100这个立即数默认为`int`类型，怎么可能赋值给`t`对象呢！现在就报错吗？不急，我看看有没有转换构造函数！OK，发现`Test`类中定义了`Test(int i)`，可以进行转换，默认等价于：`t = Test(100);`。
- 编译器尽力尝试的绪果是隐式类型转换

```
#include <iostream>
#include <string>

using namespace std;
class Test{
    int mvalue;
    public:
        Test(){

        }
        Test(int i){

        }
        Test operator + (const Test& p){
            Test ret(mvalue + p.mvalue);
            return ret;
        }
      int value(){
          return mvalue;
      }
};

int main(){
    Test t;
    t = Test(5);
    Test t;
    //编译通过，bug存在，编译器隐式类型转换
    r = t + 10; //r = t + Test(10);

    cout << r.value() << endl;
    return 0;
}
```

- 隐式类型转换
    - 会让程序以意想不到的方式进行工作
    - 是工程中 bug 的重要来源
- 工程中通过 `explicit` 关键字杜绝编译器的转换尝试
- 转换构造函数被 `explicit` 修饰时只能进行显示转换
    - 转换方式
    - ```
      static_cast <ClassName >(value);
      ClassName(value);
      (ClassName)value;//不推荐
      ```

```
#include <iostream>
#include <string>

using namespace std;
class Test{
    int mvalue;
    public:
        Test(){

        }
        explicit Test(int i){
```
```

```
        }
        Test operator + (const Test& p){
            Test ret(mvalue + p.mvalue);
            return ret;
        }
      int value(){
          return mvalue;
      }
};

int main(){
    Test t;
    //t = Test(5);
    t = static_cast<Test>(5);
    Test r;
    //explicit 后就需要我们自己强制类型转换
    r = t + static_cast<Test>10; //r = t + Test(10);

    cout << r.value() << endl;
    return 0;
}
```

小结

- 转换构造函数只有一个参数
- 转换构造函数的参数类型是其它类型
- 转换构造函数在类型转换时被调用
- 隐式类型转换是工程中 bug 的重要来源
- `explicit` 关键字用于杜绝隐式类型转换

问题

- 类类型是否能够类型转换到普通类型?

```
#include <iostream>
#include <string>

using namespace std;
class Test{
      Test(){

      }
      int value(){
          return mvalue;
      }
};

int main(){
    Test t;
    //编译不通过
    int i= (int)t;
    return 0;
}
```

类型转换函数

- C+ +类中可以定义类型转换函数
- 类型转换函数用于将类对象转换为其它类型
- 语法规则：
- ```
  operator Type ()
  {
      Type ret;
      //
      return ret;
  }
  ```

```
#include <iostream>
#include <string>

using namespace std;
class value{
  public :
    value(){

    }
};
class Test{
    int mvalue;
    public:
      Test(int i){
            mvalue = i;
      }
      int value(){
         return mvalue;
      }
    operator int (){
        return mvalue;
    }

    operator value(){
        value ret;
        return ret;
    }
};

int main(){
    Test t(100);
    //隐士类型转换
    //int i = t.operator int()
    int i = t;
    //隐式类型转换
    value v = t;
    //此时i为100
    return 0;
}
```

类型转换函数

- 与转换构造函数具有同等的地位
```

- 使得编译器有能力将对象转化为其它类型
- 编译器能够隐式的使用类型转换函数
- 无法抑制隐式的类型转换函数调用
- 类型转换函数可能与转换构造函数冲突
- 工程中以 `Type toType ()` 的公有成员代替类型转换函数
    - ```
      //Qt里面的例子
      #include <QDebug>
      #include <QString>

      int main(){
          QString str = "";
          int i = 0;
          double d = 0;
          short s = 0;

          str = "-255";
          //Type toType()
          i = str.toInt();
          d = str.toDouble();
          s = str.toShort();

          qDebug() << "i = " << i <<endl;
          qDebug() << "d = " << d <<endl;
          qDebug() << "s = " << s <<endl;
          return 0;
      }
      ```

3.6 继承的概念和意义

思考

- 类之间是否存在直接的关联关系?

组合关系：**整体与部分的关系**

比如：电脑：CPU+硬盘+内存+主板

```
#include <iostream>
#include <string>

using namespace std;
class cpu{
  public :
    cpu(){

    }
};

class Memory{
  public :
    Memory(){

    }
```
```

```
};

class mainBoard{
  public :
    mainBoard(){

    }
};

class Disk{
  public :
    Disk(){

    }
};

class computer{
  mainBoard mbd;
  cpu cpu1;
  Memory mem;
  Disk disk;
  string stros;
  public:
  computer(){

  }
  void install(string os){
      stros = os;
  }
};
```

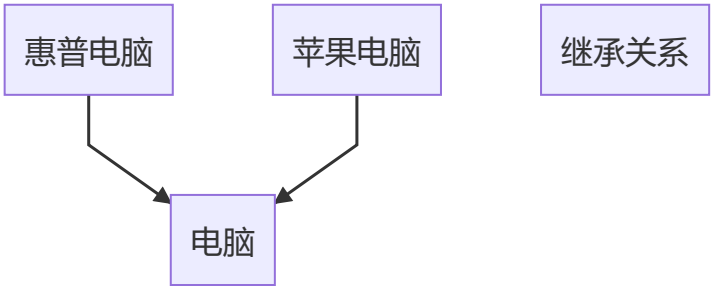
类之间的组合关系

- 组合关系的特点
  - 将其它**类的对象**作为当前类的成员使用
  - 当前**类的对象与成员对象的生命期**相同
  - 成员对象在用法上与普通对象**完全一致**

组合关系是最简单的关系，在实际开发中能用组合关系的就不要用其他更复杂的关系。

继承关系：**父子关系**

- 继承关系：父子关系



父类：电脑；子类（派生类）：惠普电脑和苹果电脑

惊艳的继承

- 面向对象中的**继承**指类之间的父子关系
    - 子类拥有父类的**所有属性和行为**
    - 子类就是一种特殊的父类
    - **子类对象可以当作父类对象使用**
    - 子类中可以**添加父类没有的**方法和属性

C+ +中通过下面的方式描述继承关系

```
#include <iostream>
#include <string>

using namespace std;

class Parent{
    int mv;
public:
    Parent(){
        cout << "Parent()" <<endl;
        mv = 10;
    }
    void method(){
        cout << "mv = " << mv <<endl;
    };
};
class Child: public Parent{ //描述继承关系
public:
    void hello(){
        cout << "I am a child" << endl;
    }
};

int main(){
    //继承代码复用
    Child c;
    c.method();
    return 0;
}
```

重要的规则:

- 子类是一个特殊的父类
- 子类对象可以直接初始化父类对象
- 子类对象可以直接赋值给父类对象

```
Child c;
Parent p1 = c;
Parent p2;
p2 = c;
```

继承的意义:

- 继承是 C+ +中代码复用的重要手段。通过继承可以获得父类的所有功能 并且可以在子类中重写已有功能，或者添加新功能。

小结

- 继承是面向对象中类之间的一种关系

- 子类用于父类的所有属性和行为
- 子类对象可以当作父类对象使用
- 子类中可以添加父类没有的方法和属性
- 继承是面向对象中代码复用的重要手段

3.7 继承的访问级别

值得思考的问题

- 子类是否可以直接访问父类的私有成员?

根据C++语法:

- 外界不能直接访问类的`private`成员，那么子类应该不能直接访问父类的私有成员。

继承中的访问级别

- 面向对象中的访问级别不只是 `public` `private`
- 可以定义`protected` 访问级别
- 关键字`protected` 的意义
    - 修饰的成员不能被外界直接访问
    - 修饰的成员可以被子类直接访问

```
class Test{
    protected a;
};
class a : public Test{
    //
    //可以访问a
    a = 10;
};

int main(){
    Test a;
    //main相对于类是外部，访问失败
    a.a = 100;
}
```

思考

- 为什么面向对象中需要 `protected` ?

```
//实现点线功能
#include <iostream>
#include <string>

using namespace std;

class object{
 protected:
    string mName;
    string mInfo;
 public:
    object(){
        mName = "object";
        mInfo = "";
```

```
    }
    string name(){
        return mName;
    }
    string info(){
        return mInfo;
    }
};

class point : public object{
 private:
    int mX;
    int mY;
 public:
    point(int x = 0, int y = 0){
        ostringstream s;
        mX = x;
        mY = Y;
        mName = "point";

        s<< "P(" << mX << "," << mY << ")";

        mInfo = s.str();
    }
    int x(){
        return mX;
    }
    int y(){
        return mY;
    }
};

class line : public object{
private:
    point p1;
    point p2;
public:
    line(point p1, point p2){
        mp1 = p1;
        mp2 = p2;
        mName = "line";

        s << "line from " << mp1.info() << "to" << mp2.info();
        mInfo  = s.str();
    }
};
int main(){
    object o;
    point p(1,3);
    //下面自己打印各种信息
    return 0;
}
```

小结

- 面向对象中的访问级别不只是 `public private`
- `protected` 修饰的成员不能被外界所访问

- `protected` 使得子类能够访问父类的成员
- `protected` 关键字是为了继承而专门设计的
- 没有 `protected` 就无法完成真正意义上的代码复用

3.8 不同的继承方式

被忽视的细节

冒号`:`表示继承关系， `Parent` 表示被继承的类，`public` 的意义是什么？

```
class Parent{
};
class Child : public Parent{ //public什么意思
};
```

有趣的问题

- 是否可以将继承语句中的 `public` 换成 `protected private` ？如果可以，与 `public` 继承有什么区别？

```
#include <iostream>
#include <string>

using namespace std;
class parent{

};
//三种继承有什么区别？ 编译通过，说明C++不只一种继承方式
class childA : public parent{};

class childB : protected parent{};

class childC : private parent{};

int main(){

    return 0;
}
```

不同的继承方式

- C+ +中支持三种不同的继承方式
    - `public` 继承
        - 父类成员在子类中保持原有访问级别
    - `private` 继承
        - 父类成员在子类中变为私有成员
    - `protected` 继承
        - 父类中的公有成员变为保护成员，其它成员保持不变

	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

继承成员的访问属性 = max{继承方式，父类成员访问属性}

C++中的默认**继承方式为`private`!**

```
#include <iostream>
#include <string>

using namespace std;
class parent{
    private :
    int m_a;
    protected:
    int m_b;
    public:
    int m_c;
    void set(int a, int b, int c){
        m_a = a;
        m_b = b;
        m_c = c;
    }
};
//三种继承有什么区别？ 编译通过，说明C++不只一种继承方式
class childA : public parent{
    public:
    void print(){
        cout << "m_a" << m_a << endl;
    }
};

class childB : protected parent{
    public:
    void print(){
        cout << "m_a" << m_a << endl;
    }
};

class childC : private parent{
    public:
    void print(){
        cout << "m_a" << m_a << endl;
    }
};

int main(){
    //定义对象进行访问实验
    childA a;
    a.print();
    return 0;
}
```

遗憾的事实

- 一般而言C ++工程项目中**只使用`public`继承**
- C++的派生语言只支持一种继承方式 `public` 继承
- `protected private` 继承带来的复杂性远大于实用性

D语言的继承：

```
//将下面的代码定义为模块
module D_Demo;
//C++中的#include
import std.stdio;
import std.string;

class obj{
protected:
    string mName;
    string mInfo;
public:
    //D语言的构造函数
    this(){
        mName = "object";
        mInfo = "";
    }
    string name(){
        return mName;
    }
    string info(){
        return mInfo;
    }
};
//D语言只有公有继承
class point : obj{
private:
    int mX;
    int mY;
public:
    this(int x, int y ){
        mX = x;
        mY = Y;
        mName = "point";
        mInfo = format("P(%d,%d)",mX,My);
    }
    int x(){
        return mX;
    }
    int y(){
        return mY;
    }
};

void main(string[] args){
    writefln("D Demo");
    point p = new point(1,2);
    writefln(p.name());
    writefln(p.info());
}
```

C#的继承

```
class obj{
    protected string mName;
    protected string minfo;
    //构造函数
    public obj(){
        mName = "object";
        mInfo = "";
    }
    public string name(){
        return mName;
    }
    public string info(){
        return mInfo;
    }
}

class point : obj{
    private int mX;
    private int mY;

    public point(int x, int y){
        mX = x;
        mY = y;
        mName = "point";
        mInfo = "P" + mX + "," + mY + ")";
    }

    public int x(){
        return mX;
    }

    public int y(){
        return mY;
    }
}

class program{
    public static void Main(String[] args){
        System.Console.WriteLine("C# Demo");
        point p = new point(1,2);
        System.Console.WriteLine(p.name());
        System.Console.WriteLine(p.info());
    }
}
```

Java的继承：

```
class obj{
    protected string mName;
    protected string minfo;
    //构造函数
    public obj(){
        mName = "object";
        mInfo = "";
    }
```

```
    public string name(){
        return mName;
    }
    public string info(){
        return mInfo;
    }
}


class point extends obj{
    private int mX;
    private int mY;

    public point(int x, int y){
        mX = x;
        mY = y;
        mName = "point";
        mInfo = "P" + mX + "," + mY + ")";
    }

    public int x(){
        return mX;
    }

    public int y(){
        return mY;
    }
}
class program{
    public static void Main(String[] args){
        System.out.println("C# Demo");
        point p = new point(1,2);
        System.out.println(p.name());
        System.out.println(p.info());
    }
}
```

小结

- C+ +中支持3种不同的继承方式
- 继承方式直接影响父类成员在子类中的访问属性
- 一般而言 工程中只使用 `public` 的继承方式
- C+ +的派生语言中只支持 `public` 继承方式

3.9 继承中的构造与析构

思考

- 如何初始化父类成员?
- 父类构造函数和子类构造函数有什么关系?

子类对象的构造

- 子类中可以定义构造函数
- 子类构造函数

- 必须对继承而来的成员进行初始化
  - 直接通过初始化列表或者赋值的方式进行初始
  - 调用父类构造函数进行初始化

父类构造函数在子类中的调用方式

- **默认调用**
  - 适用于无参构造函数和使用默认参数的构造函数
- **显示调用**
  - 通过初始化列表进行调用
  - 适用于**所有**父类构造函数

父类构造函数的调用

```
class Child :public Parent
{
public :
    Child()//隐式调用
    {
        cout << " Child () " << endl;
    }
    Child(string s) : Parent( " Parameter to Parent" )//初始化列表显示调用父类构造函数
    {
        cout << " Child () " << s << endl;
    }
} ;
```

```
#include <iostream>
#include <string>

using namespace std;

class parent{
public:
    parent(){
        cout << "parent" << endl;
    }
    parent(string s){
        cout << "parent(string s)" << s <<endl;
    }
};

class child : public parent{
    public:
        child(){
            cout << "child" << endl;
        }
        child(string s) : parent(s){
            cout << "child(string s)" << endl;
        }
};

int main(){
    child c; // parent() child
    child cc("cc"); //parent(string s) child(string s)
    return 0;
```

```
}
```

构造规则

- 子类对象在创建时会首先调用父类的构造函数
- 先执行父类构造函数再执行子类的构造函数
- 父类构造函数可以被**隐式调用**或者**显示调用**（显示调用必须通过初始化列表完成）

对象创建时构造函数的调用顺序

1. 调用父类的构造函数
2. 调用成员变量的构造函数
3. 调用类自身的构造函数

口诀心法：

- 先父母，后客人，再自己

```
#include <iostream>
#include <string>

using namespace std;

class object{
   public:
    object(string s){
        cout << "parent" << endl;
    }
};
class parent: public object{
public:
    parent(): object("default"){
        cout << "parent" << endl;
    }
    parent(string s): object(s){
        cout << "parent(string s)" << s <<endl;
    }
};

class child : public parent{
    object m1;
    object m2;
    public:
    //此次为重点
        child(): m1("default"), m2("default"){
            cout << "child" << endl;
        }
        child(string s) : parent(s), m1(s), m2(s){
            cout << "child(string s)" << endl;
        }
};

int main(){
    child c; // parent() child
    child cc("cc"); //parent(string s) child(string s)
    return 0;
}
```

子类对象的析构

析构函数的调用顺序与构造函数相反

1. 执行自身的析构函数
2. 执行成员变量的析构函数
3. 执行父类的析构函数

```
#include <iostream>
#include <string>

using namespace std;

class object{
    string ms;
   public:
    object(string s){
        cout << "object" << endl;
    }
    ~object(){
        cout << "~object" << endl;
    }
};
class parent: public object{
public:
    parent(): object("default"){
        cout << "parent" << endl;
    }
    parent(string s): object(s){
        cout << "parent(string s)" << s <<endl;
    }
    ~parent(){
        cout << "~parent" << endl;
    }
};

class child : public parent{
    object m1;
    object m2;
    public:
    //此次为重点
        child(): m1("default"), m2("default"){
            cout << "child" << endl;
        }
        child(string s) : parent(s), m1(s), m2(s){
            cout << "child(string s)" << endl;
        }
   ~child(){
       cout << "~child" << endl;
   }
};

int main(){
    child c; // parent() child
    child cc("cc"); //parent(string s) child(string s)
    return 0;
}
```

小结

- 子类对象在创建时需要调用父类构造函数进行初始化
- 先执行父类构造函数然后执行成员的构造函数
- 父类构造函数显示调用需要在初始化列表中进行——隐式调用只能是无参构造函数
- 子类对象在销毁时需要调用父类析构函数进行清理
- 析构顺序与构造顺序对称相反

3.10 父子间的冲突

思考

- 子类中是否可以定义父类中的同名成员？如果可以， 如何区分？如果不可以， 为什么？

```
#include <iostream>
#include <string>

using namespace std;
//命名空间跟后面的类是一致的
namespace A{
    int g_i = 1;
}

namespace B{
    int g_i = 1;
}


class parent{
    public:
    int mi;
    parent(){
        cout << "parent():" << &mi << endl;
    }
};

class child : public parent{
  public :
    int mi;
    child(){
        cout << "child():" << &mi << endl;
    }
};
int main(){
    child c;
    c.mi = 100; //mi究竟是子类自定义的，还是父类继承来的。
    c.parent::mi = 1000;
    return 0;
}
```

答案

- 子类**可以定义**父类中的同名成员
- 子类中的成员将**隐藏**父类中的同名成员
- 父类中的**同名成员依然存在于子类中**

- 通过**作用域分辨符**`::`访问父类中的同名成员

访问父类中的同名成员

```
child c;
c.mi = 100; //子类中的mi
c.parent::mi = 1000; //父类中的mi
```

再论重载

类中的成员函数可以进行重载

1. 重载函数的本质为多个不同的函数
2. 函数名和参数列表是唯一的标识
3. 函数重载必须发生在同一个作用域中

```
#include <iostream>
#include <string>

using namespace std;

class test {
    public:
    void add(int x, int y){

    }
};

class add : public test{
  public:
    //重载父类的成员函数，error
    //两个add函数处在了不同的作用域，只是发生了同名覆盖
    void add(int x, int y ,int z){

    }
};

int main(){
    add t;
    //发生同名覆盖，使用作用域访问父类的同名函数，不可能发生函数重载
    t.test::add(1,3);
    return 0;
}
```

问题

子类中定义的函数是否能重载父类中的同名函数?

- 子类中的函数将隐截父类的同名函数
- 子类无法重载父类中的成员函数
- 使用作用域分辨符访问父类中的同名函数
- 子类可以定义父类中完全相同的成员函数——重写
- 使用作用域分辨符访问父类中的同名成员

3.11 同名覆盖引发的问题

父子间的赋值兼容

- 子类对象可以当作父类对象使用(兼容性)
  - 子类对象可以直接赋值给父类对象
  - 子类对象可以直接初始化父类对象
  - 父类指针可以直接指向子类对象
  - 父类引用可以直接引用子类对象

```
#include <iostream>
#include <string>

using namespace std;

class parent{
public:
    int mi;

    int add(int i){
        mi += i;
    }

    int add(int a, int b){
        mi += (a + b);
    }
};

class chidl : public parent{
    public:
        int mv;
    //同名覆盖
        int add(int x, int y, int z){
            mv += (x + y + z);
        }
};
int main(){
    parent p;
    child c;
    p = c;

    parent p1(c);

    parent& rp = c;
    parent* pp = &c;
    rp,mi = 100;
    //使用引用和指针，只能访问父类，本质是子类对象退化为父类对象
    rp.add(5); //没有发生同名覆盖
    rp.add(10,19); //没有发生同名覆盖

    pp -> mv = 1000; //error not find mv
    pp -> add(1,2,3); //error not find add
    return 0;
}
```

当使用父类指针（引用）指向子类对象时

- 子类对象退化为父类对象
- 只能访问父类中定义的成员
- 可以直接访问被子类覆盖的同名成员

特殊的同名函数

- 子类中可以重定义父类中已经存在的成员函数
- 这种重定义发生在继承中，叫做函数重写
- 函数重写是同名覆盖的一种特殊情况

思考

- 当函数重写遇上赋值兼容会发生什么?

```
#include <iostream>
#include <string>

using namespace std;

class parent{
public:
    int mi;

    int add(int i){
        mi += i;
    }

    int add(int a, int b){
        mi += (a + b);
    }
};

class chidl : public parent{
    public:
        int mv;
    //同名覆盖
        int add(int x, int y, int z){
            mv += (x + y + z);
        }
};

void print(parent* p)
{
   p->print();
}
int main(){
    parent p;
    child c;
    p = c;

    parent p1(c);

    parent& rp = c;
    parent* pp = &c;
    return 0;
}
```

问题分析

- 编译期间，编译器只能根据**指针的类型**判断指向的对象
- 根据**赋值兼容**，编译器认为父类指针指向的是父类对象
- 因此，编译结果只可能是**调用父类中定义的同名函数**

```
void print(parent* p)
{
   p->print();
}
```

在编译这个函数的时候，编译器不可能知道指针`p`究竟指向了什么。

但是编译器没有理由报错。于是编译器认为最安全的做法是调用父类`print`函数，因为父类和子类肯定都有相同的`print`函数。

问题

编译器的处理方法是合理的吗？是期望的吗?

小结

- 子类对象可以当作父类对象使用（赋值兼容）
- 父类指针可以正确的指向子类对象
- 父类引用可以正确的代表子类对象
- 子类中可以重写父类中的成员函数

3.12 多态的概念和意义

函数重写回顾

- 父类中被重写的函数依然会继承给子类
- 子类中重写的函数将覆盖父类 的函数
- 通过作用域分辨符`::`可以访问到父类中的函数

面向对象中期望的行为

- 根据实际的对象类型判断如何调用重写函数
- 父类指针（引用）指向
  - 父类对象则调用父类中定义的函数
  - 子类对象则调用子类中定义的重写函数

面向对象中的多态的概念

- 根据实际的**对象类型决定函数调用**的具体目标
- 同样的**调用语句**在实际运行时有多种不同的表现形态
  - $p->print();\begin{cases} p\text{指向父类对象} & \text{I am parent} \\ p\text{指向子类对象} & \text{I am child} \end{cases}$

C++ **语言直接支持多态的概念**

- 通过使用`virtual`关键字对多态进行支持
- 被`virtual`声明的函数被重写后具有多态特性
- 被`virtual`声明的函数叫做虚函数

```
#include <iostream>
#include <string>

using namespace std;
```

```
class parent{
public:
    virtual void print(){
        cout << "I am parent" << endl;
    }
};

class chidl : public parent{
    public:
    //现在的print()已经具有多态性
    void print(){
        cout << "I am child" << endl;
    }
};

void how_to_print(parent* p)
{
   p->print();
}
int main(){
    parent p;
    child c;
    //指针指向的对象调用对应的函数
    how_to_print(&p);
    how_to_print(&c);
    return 0;
}
```

多态的意义

- 在程序运行过程中展现出动态的特性
- **函数重写必须多态实现**，否则没有意义
- 多态是面向对象组件化程序设计的基础特性——后续的语言默认都为虚函数

理论中的概念

- 静态联编
    - 在程序的编译期间就能确定具体的函数调用——函数重载
- 动态联编
    - 在程序实际运行后才能确定具体的函数调用——函数重写

```
#include <iostream>
#include <string>

using namespace std;

class parent{
public:
    virtual void func(){
        cout << "void func" << endl;
    }

    virtual void func(int i){
        cout << "void func(int i)" << i << endl;
    }
    virtual void func(int i, int j){
```

```
        cout << "void func(int i, int j)" << i << "j:" << j <<endl;
    }
};

class child: public parent{
public:
    void func(int i, int j){
        cout << "void func(int i, int j):" << i + j << endl;
    }
    //同名覆盖
    void func(int i, int j, int k){
        cout << "void func(int i, int j, int k):" << i + j + k<< endl;
    }
};

void run( parent* p){
     p->func(1,3); //展现多态特性
                   //动态联编
}
int main(){
    //以下都是静态联编
   parent p;
   p.func();
   p.func(1);
   p.func(1,2);
   child c;
   c.func(1,2);
   run(&p);
   run(&c);
    return 0;
}
```

```
//少庄主替老庄主报仇版C++
#include <iostream>
#include <string>

using namespace std;

class boss{
public:
    int fight(){
        int ret = 10;
        cout << "boss::fighting():" << ret<< endl;

        return ret;
    }
};

class Master{
public:
    virtual int eightSwordKill(){
        int ret = 8;

        cout << "Master::eightSwordKill():" << ret << endl;
        return ret;
    }
};
```

```
//赋值兼容性
void field_pk(Master* master, Boss* boss){
    int k = master->eightSwordKill();
    int b = boss->fight();

    if(k < b){
        cout << "Master is killed.." << endl;
    }
    else
        cout << "Boss is killed.."<< endl;
}

class NewMaster : public Master{
public:
      int eightSwordKill(){
            int ret = Master::eightSwordKill() * 2;

            cout << "Master::eightSwordKill():" << ret << endl;
        return ret;
    }
};

int main(){
    Master master;
    Boss boss;

    cout << "Master vs BOSS" << endl;

    field_pk(&master, &boss);

    cout << "NewMaster vs BOSS" << endl;

    NewMaster newMater;

    field_pk(&newMaster, &boss);
    return 0;
}
```

小结

- 函数重写只可能发生在父类 和子类之间
- 根据实际对象的类型确定调用的具体函数
- `virtual`关键字是 C+ +中支持多态的唯一方式
- 被重写的虚函数可表现出多态的特性

3.13 C++对象模型分析

回归本质

- `class` 是一种特殊的`struct`
    - 在内存中`class` 依旧可以看作变量的集合
    - `class struct` 遵循相同的内存对齐规则
    - `class` 中的成员函数与成员变量是分开存放的
        - **每个对象有独立的成员变量**

- **所有对象共享类中的成员函数**

值得思考的问题

```
class A{
  int i;
  int j;
  char c;
  double d;
};
//sizeof(A)?

struct B{
  int i;
  int j;
  char c;
  double d;
};
//sizeof(B)?
```

```
#include <iostream>
#include <string>

using namespace std;
class A{
  int i;
  int j;
  char c;
  double d;
public:
    //虽然类的成员函数和类的成员函数语法上是写在一起的，但是实际内存分配是不同地方的
    void print(){
        cout << "i = " << i << ","
            << "j = " << j << ","
            << "c = " << c << ","
            << "d = " << d << ","
            <<endl;
    }
};

struct B{
  int i; //
  int j; //
  char c;
  double d;
};
int main(){
    //数据类型就是固定内存空间的别名
    A a;
    //对象仅仅是成员变量的集合
    cout << "sizeof (a) = " << sizeof(a) << endl; // 20 bytes
    cout << "sizeof (B) = " << sizeof(B) << endl; // 20 bytes
    cout << "sizeof (A) = " << sizeof(A) << endl; // 20 bytes

    a.print();
    //访问权限关键字失效在运行时，仅仅只是二进制代码
    B* p = reinterpret_cast<B*>(&a);
```

```
    //指针修改私有成员
    p->i = 1;
    p->j = 2;
    p->c = 3;
    p->d = 4; // i,j,c,d =1,2,3,4
    return 0;
}
```

运行时的对象退化为结构体的形式

- 所有成员变量在内存中依次排布
- 成员变量间**可能存在内存空隙**
- 可以**通过内存地址**直接访问成员变量
- **访问权限关键字在运行时失效**

C++ 对象模型分析

- 类中的**成员函数**位于代码段中
- 调用成员函数时对象地**址作为参数隐式传递**
- 成员函数**通过对象地址访问成员变量**
- C++ 语法规则隐藏了对象地址的传递过程

对象本质分析

`test.cpp`

```
#include <iostream>
#include <string>

using namespace std;

class Demo{
    int mi;
    int mj;
public:
    Demo(int i, int j){
        mi = i;
        mj = j;
    }
    int getI(){
        return mi;
    }

    int getJ(){
        return mj;
    }

    int add(int value){
        return mi + mj + value;
    }
};
int main(){

    Demo d(1,2);

    cout << "sizeof(d) =" << sizeof(d) << endl; //8bytes
    cout << "d.getI() = " << d.getI() << endl; //1
    cout << "d.getJ() = " << d.getJ() << endl; //2
```

```
    cout << "d.add(3) = " << d.add(3) << endl; //6
    return 0;
}
```

用C语言实现上述程序

`a.h`:

```
#ifndef _A_H_
#define _A_H_
//void实现信息隐藏
typedef void Demo;

Demo* Demo_Create(int i, int j);
int Demo_GetI(Demo* pThis);
int Demo_GetJ(Demo* pThis);
int Demo_Add(Demo* pThis,int value);

void Demo_Free(Demo* pThis);
#endif
```

`a.c`:

```
#include "a.h"

struct ClassDemo{
  int mi;
  int mj;
};
Demo* Demo_Create(int i, int j){
    struct ClassDemo* ret = (struct ClassDemo*)malloc(sizeof(ClassDemo));

    if(ret != NULL){
        ret-> mi = i;
        ret-> mj = j;
    }

    return ret;
}
int Demo_GetI(Demo* pThis){
    struct ClassDemo* obj = (struct ClassDemo*)pThis;

    return obj->mi;
}
int Demo_GetJ(Demo* pThis){
    struct ClassDemo* obj = (struct ClassDemo*)pThis;

    return obj->mj;
}
int Demo_Add(Demo* pThis,int value){
   struct ClassDemo* obj = (struct ClassDemo*)pThis;

    return obj->mi + obj->mj + value;
}
void Demo_Free(Demo* pThis){
    free(pThis);
```

```
}
```

`main.c`:

```
#include <stdio.h>
#include "a.h"

int main(){
    Demo* d = Demo_Create(1, 2); //Demo* d = new Demo(1, 2);

    printf("d.mi = %d\n", Demo_GetI(d)); // d-> getI()
    printf("d.mj = %d\n", Demo_GetJ(d)); // d-> getI()
    d->mi = 100; //error mi是private，类外部无法访问，面向对象信息隐藏
    Demo_Free(d);
    return 0;
}
```

小结

- C+ +中的类对象在内存布局上与结构体相同
- 成员变量 和成员函数在内存中分开存放
- 访问权限关键字在运行时失效
- 调用成员函数时对象地址作为参数隐式传递

继承对象模型

- 在 C+ +编译器的内部类可以理解结构体
- 子类是由父类成员叠加子类新成员得加到的
    - ```
      class Derived : public Demo{
        int k;
      };
      ```

```
#include <iostream>
#include <string>

using namespace std;
class Demo{
  int mi;
  int mj;
};
class Derived : public Demo{
  int k;
public:
    Derived(int i, int j, int k){
        mi = i;
        mj = j;
        mk = k;
    }

    void print(){
        cout << "mi = " << mi << endl; //余下的类似
    }
};
```
```

```
struct Test{
    int mi;
    int mj;
    int mk;
};
int main(){
    cout << "sizeof(Demo) = " << sizeof(Demo) << endl; //8
    cout << "sizeof(Derived) = " << sizeof(Derived) << endl; //12
    Derived d(1,2,3);
    Test* p = reinterpret_cast<Test*>(&d);

    d.print();

    p->mi = 10;
    p->mj = 20;
    p->mk = 30;

    d.print(); //d的对象内存分布跟结构体Test是一致的。
    return 0;
}
```

C++**多态的实现原理**

- 当类中声明虚函数时，编译器会在类中生成一个**虚函数表**
- 虚函数表是一个**存储成员函数地址**的数据结构
- 虚函数表是由**编译器**自动生成与维护的
- `virtual` 成员函数会被编译器放入虚函数表`VTABLE`中
- 存在虚函数时，每个对象中都有一**个指向虚函数表的指针**`VPTR`

```
void run(Demo* p , int v){ //实现多态的函数
    p-add(v); //p->具体对象(VPTR)->VTABLE(void(*pAdd)(int value)) ->Offxxxxx
}
```

编译器确认`add()`是否为虚函数?

1. Yes->编译器在对象`VPTR`所指向的虚函数表中查找`add()`的地址
2. No->编译器直接可以确定被调成员函数的地址

```
#include <iostream>
#include <string>

using namespace std;

class Demo{
  int mi;
  int mj;
  virtual void print(){

  }
};
class Derived : public Demo{
  int k;
public:
    Derived(int i, int j, int k){
        mi = i;
        mj = j;
```

```
        mk = k;
    }
    void print(){
        cout << "mi = " << mi << endl; //余下的类似
    }
};

struct Test{
    void* p;
    int mi;
    int mj;
    int mk;
};
int main(){
    cout << "sizeof(Demo) = " << sizeof(Demo) << endl; //8
    cout << "sizeof(Derived) = " << sizeof(Derived) << endl; //12
    Derived d(1,2,3);
    Test* p = reinterpret_cast<Test*>(&d);

    d.print();

    p->mi = 10;
    p->mj = 20;
    p->mk = 30;

    d.print(); //d的对象内存分布跟结构体Test是一致的。
    return 0;
}
```

用C语言实现多态

`a.h`:

```
#ifndef _A_H_
#define _A_H_
//void实现信息隐藏
typedef void Demo;
typedef void Derived;
Demo* Demo_Create(int i, int j);
int Demo_GetI(Demo* pThis);
int Demo_GetJ(Demo* pThis);
int Demo_Add(Demo* pThis,int value);
void Demo_Free(Demo* pThis);

Derived* Derived_Create(int i, int j);
int Derived_GetK(Derived* pThis);
int Derived_Add(Derived* pThis,int value);
#endif
```

`a.c`:

```
#include "a.h"
#include "malloc.h"
static int  Demo_Virtual_Add(Demo* pThis,int value);
static int  Derived_Virtual_Add(Demo* pThis,int value);
struct VTable{ //2.定义虚函数表数据结果
```

```
    //函数指针
    int (*pAdd)(Derived* ,int); //3.虚函数表存储指针
};
struct ClassDemo{
    //VTBL
  struct VTable* vptr; //1.定义虚函数表指针
  int mi;
  int mj;
};

struct ClassDerived{
    struct ClassDemo d;
    int mk;
};
//对外隐藏
static struct VTable g_Demo_vtble = {
    Demo_Virtual_Add
};
//子类虚函数表
static struct VTable g_Derived_vtble = {
    Derived_Virtual_Add
};
Demo* Demo_Create(int i, int j){
    struct ClassDemo* ret = (struct ClassDemo*)malloc(sizeof(ClassDemo));

    if(ret != NULL){
        ret->vptr = &g_Demo_vbtl; //4.关联对象和虚函数表
        ret-> mi = i;
        ret-> mj = j;
    }
    return ret;
}
int Demo_GetI(Demo* pThis){
    struct ClassDemo* obj = (struct ClassDemo*)pThis;

    return obj->mi;
}
int Demo_GetJ(Demo* pThis){
    struct ClassDemo* obj = (struct ClassDemo*)pThis;

    return obj->mj;
}
6.定义虚函数表中指针指向的具体函数
static int  Demo_Virtual_Add(Demo* pThis,int value){
    struct ClassDemo* obj = (struct ClassDemo*)pThis;

    return obj->mi + obj->mj + value;
}
//5.分析具体的虚函数！！！
int Demo_Add(Demo* pThis,int value){
   struct ClassDemo* obj = (struct ClassDemo*)pThis;

    return obj->vptr->pAdd(pThis, value);
}
void Demo_Free(Demo* pThis){
    free(pThis);
}
static int  Derived_Virtual_Add(Demo* pThis,int value){
```

```
    struct ClassDerived* obj = (struct ClassDerived*)pThis;

    return obj->mk + value;
}
Derived* Derived_Create(int i, int j){
     struct ClassDerived* ret = (struct
ClassDerived*)malloc(sizeof(ClassDerived));

    if(ret != NULL){
        ret->d.vptr = &g_Derived_vtbl;
        ret-> d.mi = i;
        ret-> d.mj = j;
        ret-> mk = k;
    }
    return ret;
}
int Derived_GetK(Derived* pThis){
    struct ClassDerived* obj = (struct ClassDerived*)pThis;

    return obj->mk;
}
int Derived_Add(Derived* pThis,int value){
    struct ClassDerived* obj = (struct ClassDerived*)pThis;

    return obj->d.vptr->pAdd(pThis, value);
}
```

`main.c`:

```
#include <stdio.h>
#include "a.h"
void run(Demo* p ,int v){
    int r = Demo_Add(p,3);
    printf("r = %d\n", r);
}
int main(){
    Demo* d = Demo_Create(1, 2); //Demo* d = new Demo(1, 2);
    Derived* pd = Derived_Create(1, 2, 3);
    printf("d.mi = %d\n", Demo_GetI(d)); // d-> getI()
    printf("d.mj = %d\n", Demo_GetJ(d)); // d-> getJ()
    rintf("pb->add(3) = %d\n", Derived_Add(pd, 3)); // d-> getJ()
    d->mi = 100; //error mi是private，类外部无法访问，面向对象信息隐藏
    run(pd, 3);
    Demo_Free(d);
    Demo_Free(pd);
    return 0;
}
```

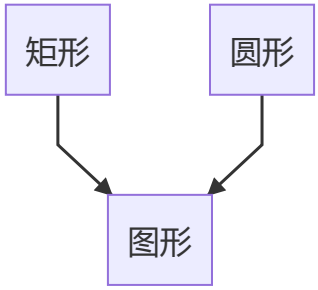
小结

- 继承的本质就是父子间成员变量的叠加;
- C+ +中的多态是通过虚函数表实现的
- 虚函数表是由编译器自动生成与维护的
- 虚函数的调用效率低于普通成员函数

3.14 C++中的抽象类和接口

首先结论是：C++中没有直接可以使用的抽象类和接口，但是并不是不可以使用。

什么是抽象类?

- 面向对象中的抽象概念
  - 在进行面向对象分析，会发现一些抽象的概念
    - 图形的面积如何计算？——单纯计算图像面积不可能，但是图形都有面积。
    - 

在现实中需要知道具体的图形类型才能求面积，所以对概念上的“图形“ 求面积是没有意义的!

```
class shape{
  public:
        double area(){
            return 0;
        }
};
```

`shape`只是一个概念上的类型，没有具体的对象！上面的代码不能保证不能创建对象，从而容易导致运行时Bug。

问题

`shape` 类有必要存在吗?

面向对象中的抽象类

- 可用于表示现实世界中的抽象概念
- 是一种只能定义类型 ， 不能产生对象的类
- 只能被继承并重写相关函数
- 直接特征是相关函数没有完整的实现

在设计的时候就考虑是否将其设计为抽象类，当它不能产生对象时。

`shape`是现实世界中各种图形的抽象概念。

因此:

- 程序中必须能够反映抽象的图形
- 程序中通过抽象类表示图形的概念
- 抽象类不能创建对象，只能用于继承

抽象类与纯虚函数

- C+ +语言中没有抽象类的概念
- C+ +中通过**纯虚函数**实现抽象类

- 纯虚函数是指**只定义原型的成员函数**
- 一个 C+ +类中存在**纯虚函数**就成为了抽象类

纯虚函数的语法规则

```
class Shape
{
    public :
    virtual double area() = 0;
};
```

”= 0“ 用于告诉编译器当前是声明纯虚函数，因此不需要定义函数体。

```
#include <iostream>
#include <string>

using namespace std;

class shape{
  public:
        virtual double area() = 0;
};

class Rect : public shape{
    int ma;
    int mb;
public:
    Rect(int a, int b){
        ma = a;
        mb = b;
    }
    double area(){
        return ma * mb;
    }
};

class Circle : public shape{
    int mr;
public:
    Circle(int r){
        mr = r;
    }
    double area(){
        return 3.14 * mr * mr;
    }
};

void area(shape* p){
    double r = p->area();
    cout << "r = " << r << endl;
}
int main(){
    //erro，抽象类不能定义对象
    shape s;
    Rect rect(1,2);
    Circle circle(10);
```

```
    area(&rect);
    area(&circle);
    return 0;
}
```

- 抽象类只能用作父类被继承
- 子类必须实现纯虚函数的具体功能
- 纯虚函数被实现后成为虚函数
- 如果子类没有实现纯虚函数，则子类成为抽象类

满足下面条件的 C+ +类则称为**接口**

- 类中没有定义任何的成员变量
- 所有的成员函数都是公有的
- 所有的成员函数都是纯虚函数
- 接口是一种特殊的抽象类

```
#include <iostream>
#include <string>

using namespace std;

class Channel{
public:
    //参考Qt里面的网络接口
    virtual bool open() = 0;
    virtual void close() = 0;
    virtual bool send(char* buf, int len) = 0;
    virtual int receive(char* buf, int len) = 0;
};

int main(){

    return 0;
}
```

小结

- 抽象类用于描述现实世界中的抽象概念;
- 抽象类只能被继承不能创建对象
- C+ +中没有抽象类的概念
- C+ +中通过纯虚函数实现抽象类
- 类中只存在纯虚函数的时成为接口
- 接口是是一种特殊的抽象类

3.15 被遗弃的多重继承

问题

- C+ +中是否允许一个类继承自多个父类?

C+ +**支持编写多重继承的代码**

- 一个子类可以拥有多个父类
- 子类拥有**所有父类的成员变量**

- 子类继承**所有父类的成员函数**
- 子类对象可以当作任意父类对象使用——赋值兼容原则

多重继承的语法规则

```
class Derived : public BaseA, public BaseB, public BaseC{
  //...
};
```

多重继承的本质与单继承相同!

```
#include <iostream>
#include <string>

using namespace std;

class BaseA{
  int ma;
};

class BaseB{
  int mbl
};

class Derived : public BaseA, public BaseB{
  int mc;
};

int main(){

    return 0;
}
```

工程开发中的"多重继承"**方式**

- 单继承某个类 + 实现（多个）接口

```
#include <iostream>
#include <string>

using namespace std;

class Base{
protected:
    int m1;
public:
    Base(int i){
        mi = i;
    }
    int getI(){
        return mi;
    }
    bool equal(Base* obj){
        return (this == obj);
    }
};
```

```
class Interface2
    virtual void add(int i) = 0;
    virtual void minus(int i) = 0;
};

class Interface1{

public:
    virtual void multiply(int i) = 0;
    virtual void devide(int i) = 0;
};

class Derived : public Base, public Interface1, public Interface2{
public:
    Derived(int i) : Base(i){

    }
    void add(int i){
        mi += i;
    }
    void minus(int i){
        mi -= i;
    }
    void multiply(int i){
        mi *= i;
    }
    void divide(int i){
        if( i != 0){
            mi /= i;
        }
    }
};
int main(){
    Derived d(100);
    Derived* p = &d;
    Interface1* pInt1 = &d;
    Interface2* pInt2 = &d;

    pInt1->add(10);
    pInt2->divide(11);
    pInt1->minus(5);
    pInt2->multiply(5);


    cout << "pInt1 == p" << p -> equal(dynamic_cast<Base*>pInt1)  <<endl;
    cout << "pInt2 == p" << p -> equal(dynamic_cast<Base*>pInt2)  <<endl;
    // 1 1  p1和p2是同一个
    return 0;
}
```

正确的使用多重继承

- 一些有用的工程建议
    - 先继承自一个父类，然后实现多个接口

- 父类中提供 `equal()` 成员函数
- `equal()` 成员函数用于判断指针是否指向当前对象
- 与多重继承相关的强制类型转换用 `dynamic_cast` 完成

小结

- 多继承中可能出现多个虚函数表指针
- 与多重继承相关的强制类型转换用 `dynamic_cast` 完成
- 工程开发中采用 `继承多接口` 的方式使用多继承
- 父类提供成员函数用于判断指针是否指向当前对象

3.16 经典问题分析

关于动态内存分配

在C语言中动态内存分配的关键是 `malloc` 和 `free`，在C++中为什么要引入新的关键字 `new` 和 `delete`。

`new` 关键字与 `malloc` 函数的区别

- `new` 关键字是 C++ 的一部分
- `malloc` 是由 C库提供的函数
- `new` 以具体类型为单位进行内存分配
- `malloc` 以字节为单位进行内存分配
- `new` 在申请内存空间时可进行初始化
- `malloc` 仅根据需要申请定量的内存空间

下面的代码输出什么？为什么？

```
class Test{
public:
    Test(){
        cout << "Test()" << endl;
        mp = new int(100);
    }
    ~Test(){
        cout << "~Test()" << endl;
        delete mp;
    }
};

int main(){
    Test* pn = new Test;
    //pm指向合法的对象吗？不是
    Test* pm = (Test*)malloc(sizeof(Test)); //构造函数只调用了一次

    delete pn;
    //free可以释放new的空间，但是不会调用析构函数
    free(pm);
    return 0;
}
```

`new` 和 `malloc` 的区别

- `new` 在所有C++编译器中都被支持
- `malloc` 在某些系统开发中是不能调用

- `new`能够触发构造函数的调用
- `malloc`仅分配需要的内存空间
- 对象的创建只能使用`new`
- `malloc`不适合面向对象开发

`delete`和`free`的区别

- `delete`在所有C++编译器都被支持
- `free`在某些系统开发中是不能调用
- `delete`能够触发析构函数的调用
- `free`仅归还之前分配的内存空间
- 对象的销毁只能使用`delete`
- `free`不适合面向对象开发

关于虚函数

- 构造函数是否可以成为虚函数?
- 析构函数是否可以成为虚函数?

构造函数不可能成为虚函数

- 在构造函数执行结束后， 虚函数表指针才会被正确的初始化

析构函数可以成为虚函数

- 建议在设计类时将**析构函数声明为虚函数**

```
#include <iostream>
#include <string>

using namespace std;

class Base{

public:
    //error构造函数不能成为虚函数
    virtual Base(){
        cout << "Base" <<endl;
    }

    virtual void func(){
        cout << "func" << endl;
    }
    virtual ~Base(){
        cout << "~Base()" << endl;
    }
};

class Derived : public Base{
public:
    Derived(){

    }
    ~Derived(){
        cout << "~Derived()" << endl;
    }
};
int main(){
```

```
    Base* p = new Derived();
    //析构函数添加virtual后，指针p析构根据实际的对象释放空间
    delete p;
    return 0;
}
```

- 构造函数中是否可以发生多态?
- 析构函数中是否可以发生多态?

答案

- 构造函数中不可能发生多态行为
    - 在构造函数执行时，虚函数表指针未被正确初始化
- 析构函数中不可能发生多态行为
    - 在析构函数执行时，虚函数表指针已经被销毁
- 构造函数和析构函数中不能发生多态行为，只调用当前类 定义的函数版本

!

```
#include <iostream>
#include <string>

using namespace std;

class Base{

public:
    //error构造函数不能成为虚函数
    virtual Base(){
        cout << "Base" <<endl;
        //下面调用的是Base中的func()
        func();
    }

    virtual void func(){
        cout << "func" << endl;
    }
    virtual ~Base(){
        cout << "~Base()" << endl;
    }
};

class Derived : public Base{
public:
    Derived(){

    }
    ~Derived(){
        cout << "~Derived()" << endl;
    }
};
int main(){
    Base* p = new Derived();
    //析构函数添加virtual后，指针p析构根据实际的对象释放空间
    delete p;
    return 0;
}
```

关于继承中的强制类型转换

- 继承中如何正确的使用强制类型转换?

`dynamic_cast` 是与继承相关的类型转换关键字

`dynamic_cast` 要求相关的类中必须有虚函数

用于有直接或者间接继承关系 指针（引用）之间

- 指针:
  - 转换成功：得到目标类型的指针
  - 转换失败： 得到一个空指针
- 引用:
  - 转换成功： 得到目标类型的引用
  - 转换失败： 得到一个异常操作信息
- 编译器会检查 `dynamic_cast` 的使用是否正确
- 类型转换的结果只可能在运行阶段才能得到

```
#include <iostream>
#include <string>

using namespace std;

class Base{

public:
    //error构造函数不能成为虚函数
    virtual Base(){
        cout << "Base" <<endl;
        //下面调用的是Base中的func()
        func();
    }

    virtual void func(){
        cout << "func" << endl;
    }
    virtual ~Base(){
        cout << "~Base()" << endl;
    }
};

class Derived : public Base{
public:
    Derived(){

    }
    ~Derived(){
        cout << "~Derived()" << endl;
    }
};
int main(){
    Base* p = new Derived();
    //析构函数添加virtual后，指针p析构根据实际的对象释放空间
    //error 父类初始化子类指针出错
    Derived* pd = p;
```

```
    //error 没有虚函数
    Derived* pd = dynamic_cast<Derived*> p;
    //给父类的析构函数添加virtual后成功
     Derived* pd = dynamic_cast<Derived*> p;
    if(pd != NULL){
        cout << "pd = " << pd << endl;
    }
    else{
        cout << "error" <<endl;
    }
    delete p;
    return 0;
}
```

小结

- new / delete 会触发构造函数或者析构函数的调用
- 构造函数不能成为虚函数
- 析构函数可以成为虚函数
- 构造函数和析构函数中都无法产生多态行为
- `dynamic_cast` 是与继承相关的专用转换关键字

4. 类模板和异常

4.1 函数模板的概念和意义

发散性问题

- C+ + 中有几种交换变量的方法?

交换变量的方法

- 定义宏代码块 VS 定义函数

```
#include <iostream>
#include <string>

using namespace std;

#define SWAP(t,a,b) \
do                  \
{                    \
    t c = a;         \
    a = b;           \
    b = c;           \
}while(0)

void Swap(int& a, int& b){
    int c = a;
    a = b;
    b = c;
}
void Swap(double& a, double& b){
    double c = a;
    a = b;
    b = c;
```

```
}

int main(){
    int a = 0;
    int b = 1;

    SWAP(int, a, b);

    double m = 2;
    double n = 3;

    SWAP(double, m, n);
    cout << "m = " << m << endl;
    cout << "n = " << n << endl;
    return 0;
}
```

- 定义宏代码块
  - 优点： 代码复用 ， 适合所有的类型
  - 缺点： 编译器不知道宏的存在，缺少类型检查
- 定义函数
  - 优点： 真正的函数调用，编译器对类型进行检查
  - 缺点： 根据类型重复定义函数，无法代码复用
- 新的需求
  - C+ +中有没有解决方案集合两种方法的优点?

泛型编程

- 泛型编程的概念
  - 不考虑具体数据类型的编程方式
  - 对于`Swap`函数可以考虑下面的泛型写法
  - ```
    void Swap( T& a, T& b)
    {
        T t = a;
        a = b;
        b = t;
    }
    ```
  - `Swap`泛型写法中的`T`不是一个具体的数据类型，而是泛指任意的数据类型

函数模板

- C++中泛型编程
  - 函数模板
    - 一种特殊的函数可用不同类型进行调用
    - 看起来和普通函数很相似，区别是类型可被参数化
    ```

        - ```
          template<typename T>
          void Swap(T& a, T& b)
          {
              T t = a;
              a = b;
              b = t;
          }
          ```
    - 函数模板的语法规则
        - `template` 关键字用于声明开始进行泛型编程
        - `typename` 关键字用于声明泛指类型
    - 函数模板的使用
        - 自动类型推导调用
        - 具体类型显示调用
            - ```
              int a= 0;
              int b = 1;
              Swap(a , b); //自动推导
              float c = 2;
              float d = 3;
              Swap<float> (c, d) ; //显示调用
              ```

```
#include <iostream>
#include <string>

using namespace std;

#define SWAP(t,a,b) \
do                  \
{                    \
    t c = a;         \
    a = b;           \
    b = c;           \
}while(0)

template <typename T>
void Swap(T& a, T& b){
    T c = a;
    a = b;
    b = c;
}

template <typename T>
void Sort(T a[], int len){
    for(int i = 0; i < len; i++){
        for(int j = i; j < len; j++){
            if( a[i] > a[j]){
                Swap(a[i], a[j]);
            }
        }
    }
}
template <typename T>
```

```
void println(T a[], int len){
    for(int i = 0; i< len; i++){
        cout << a[i] << ",";
    }
    cout << endl;
}

void Swap(int& a, int& b){
    int c = a;
    a = b;
    b = c;
}
void Swap(double& a, double& b){
    double c = a;
    a = b;
    b = c;
}

int main(){
    int a = 0;
    int b = 1;

    SWAP(int, a, b);

    double m = 2;
    double n = 3;

    SWAP(double, m, n);
    cout << "m = " << m << endl;
    cout << "n = " << n << endl;
    return 0;
}
```

小结

- 函数模板是泛型编程 C+ + 中的应用方式之一
- 函数模板能够根据实参对参数类型进行推导
- 函数模板支持显示的指定参数类型
- 函数模板是 C+ + 中里要的代码复用方式

4.2 深入理解函数模板

函数模板深入理解

- 编译器从函数模板通过具体类型产生不同的函数
- 编译器会对函数模板进行两次编译
  - 对模板代码本身进行编译
  - 对参数替换后的代码进行编译
- **注意事项**:
  - 函数模板本身不允许隐式类型转换
    - 自动推导类型时，必须严格匹配
    - 显示类型指定时，能够进行隐式类型转换

```
#include <iostream>
#include <string>
```

```
using namespace std;

template <typename T>
void Swap(T& a, T& b){
    T c = a;
    a = b;
    b = c;
}

typedef void(FuncI)(int&, int&);
typedef void(FuncD)(double&, double&);
int main(){
    FuncI* pi = Swap; //编译器自动推导T为int
    FuncD* pd = Swap; //double

    cout << "pi = " << reinterpret_cast<void*> (pi) << endl;
    cout << "pd = " << reinterpret_cast<void*> (pd) << endl;
    return 0;
}
```

多参数函数模板

- 函数模板可以定义任意多个不同的类型参数

```
template <typename T1, typename T2, typename T3>
T1 Add(T2 a, T3 b){
    return static_cast<T1>(a + b);
}

int r = Add<int, float ,double>(0.5, 0.8);
```

- 对于多参数函数模板
  - 无法自动推导返回值类型
  - 可以从左向右部分指定类型参数
  - ```
    //T1 = int, T2 = double, T3 = double 手工指定返回值类型为int
    int r1 = Add<int>(0.5 , 0.8); //推荐
    //T1 = int, T2 = float, T3 = double
    int r2 = Add<int, float>(0.5, 0.8);
    //T1 = int, T2 = float, T3 = float
    int r3 = Add<int, float, float>(0.5.0.8);
    ```
  - 工程中将返回值参数作为第一个类型参数

```
#include <iostream>
#include <string>

using namespace std;
//返回值类型，第一位
template <typename T1, typename T2, typename T3>
T1 Add(T2 a, T3 b){
    return static_cast<T1>(a + b);
}
```
```

```cpp
int main(){
    //T1 = int, T2 = double, T3 = double 手工指定返回值类型为int
    int r1 = Add<int>(0.5 , 0.8); //推荐
    //T1 = int, T2 = float, T3 = double
    double r2 = Add<double, float>(0.5, 0.8);
    //T1 = int, T2 = float, T3 = float
    float r3 = Add<float, float, float>(0.5.0.8);
    cout << "r1 = " << r1 << endl; //r1 = 1
    cout << "r2 = " << r2 << endl; //r1 = 1.3
    cout << "r3 = " << r3 << endl; //r1 = 1.3
    return 0;
}
```

有趣的问题

- 当函数重载遇见函数模板会发生什么?

重载函数模板

- 函数模板可以像普通函数一样被重载
    - C+ +编译器优先考虑普通函数
    - 如果函数模板可以产生一个更好的匹配，那么选择模板
    - 可以通过空模板实参列表限定编译器只匹配模板

```cpp
int r1 = Max(1, 2);
double r2 = Max<>(0.5, 0.8); //限定编译器只匹配函数模板
```

```cpp
#include <iostream>
#include <string>

using namespace std;

template <typename T>
T Max(T a, T b){
    cout << "T Max(T a, T b)" << endl;
    return a > b ? a : b;
}
//重载函数模板
int Max(int a, int b){
    cout << "int Max(int a, int b)" << endl;
    return a > b ? a : b;
}
//三个参数的重载
template <typename T>
T Max(T a, T b, T c){
    cout << "T Max(int a, int b, int c)" << endl;
    return Max(Max(a, b), c)
}

int main(){
    int a = 1;
    int b = 2;

    cout << endl << Max(a, b) <<endl; //"int Max(int a, int b)" 普遍函数
    cout << endl << Max<>(a, b) <<endl; //强制函数模板 Max<int>(int, int)
    cout << endl << Max(3.0, 4.0) <<endl; //函数模板1
```

```
    cout << endl << Max(3.0, 4.0, 5.5) <<endl; //函数模板2
    cout << endl << Max('a', 10); //普通函数，隐式转换
    return 0;
}
```

小结

- 函数模板通过具体类型 **产生不同的函数**
- 函数模板可以定义**任意多个不同的类型参数**
- 函数模板中的**返回值类型必须显示指定**
- 函数模板可以像普通函数一样被重载

4.3 类模板的概念和意义

思考

- 在C++中是否能够将泛型的思想应用于类?

类模板

- 一些类主要用于存储和组织数据元素
- 类中数据组织的方式和数据元素的具体类型无关
- 如：数组类，链表类，Stack类，Queue类，等

C++中将模板的思想应用于类，使得类的实现不关注数据元素的具体类型

而只关注类所需要实现的功能。

C+ + **中的类模板**

- 以相同的方式处理不同的类型
- 在类声明前使用 `template` 进行标识
- `< typename T>` 用于说明类中使用的泛指类型 `T`
- ```
  template <typename T>
  class Operator{
  public:
      T op(T a, T b);
  }
  ```

类模板的应用

- 只能显示指定具体类型，无法自动推导
- 使用具体类型 `<Type>` 定义对象
- ```
  Operator<int> op1;
  Operator<string> op2;
  int i = op1.op(1, 2 );
  string s= op2.op( "" );
  ```
- 声明的泛指类型 `T` 可以出现在类模板的任意地方
- 编译器对类模板的处理方式和函数模板相同
  - 从类模板通过具体类型产生不同的类
  - **在声明的地方对类模板代码本身进行编译**
  - **在使用的地方对参数替换后的代码进行编译**

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
class operator{
public:
    T add(T a, T b){
        return a + b;
    }
    T minus(T a, T b){
        return a - b;
    }
    T multiply(T a, T b){
        return a * b;
    }
    T divide(T a, T b){
        return a / b;
    }
};

string operator - (string& l, string& r){
    return "Minus";
}
int main(){
    operator<int> op1;

    cout << op1.add(1, 2) <<endl;
    //第一次编译
    operator<string> op2;
    //第二次编译
    cout << op2.add("", "") <<endl;
    //编译error，重载-后成功编译
    cout << op2.minus("", "") <<endl;

    return 0;
}
```

类模板的工程应用

- 类模板必须在头文件中定义
- 类模板不能分开实现在不同的文件中
- 类模板外部定义的成员函数需要加上模板`<>`声明

以上只是工程上的建议，但非常有用。

`Operator.h`:

```
#ifndef _OPEROT_H
#define _OPEROT_H
template <typename T>
class operator{
public:
    T add(T a, T b);
    T minus(T a, T b);
    T multiply(T a, T b);
```

```
    T divide(T a, T b);
};

template <typename T>
T operator<T>::minus(T a, T b){
    return a - b;
}

template <typename T>
T operator<T>::add(T a, T b){
    return a + b;
}

template <typename T>
T operator<T>::multiply(T a, T b){
    return a * b;
}

template <typename T>
T operator<T>::divide(T a, T b){
    return a / b;
}
#endif
```

`test.cpp`:

```
int main(){
    operator<int> op1;

    cout << op1.add(1, 2) <<endl;
    cout << op2.minus(1, 2) <<endl;
    return 0;
    return 0;
}
```

小结

- 泛型编程的思想可以应用于类
- 类模板以**相同的方式处理不同类型的数据**
- 类模板非常适用于编写**数据结构**相关的代码
- 类模板在使用时**只能显示指定类型**

4.4 类模板深度剖析

多参数类模板

- 类模板可以定义任意多个不同的类型参数
- ```
  template
  < typenarne T1 , typename T2>
  class Test
  {
      public :
      void add(T2 a, T2 b);
  } ;

  Test<int, float> t;
  ```
  ```

类模板可以被特化

- 指定类模板的特定实现
- 部分类型参数必须显示指定
- 根据类型参数分开实现类模板

类模板的特化类型

- 部分特化 - 用特定规则约束类型参数
- 完全特化 - 完全显示指定类型参数

```
#include <iostream>
#include <string>

using namespace std;

template <typename T1, typename T2>
class Test{
public:
    void add(T1 a, T2 b){
        cout << "void add(T1 a, T2 b)" <<endl;
        cout << a + b << endl;
    }
};

template
<typename T> //部分特化
class Test<T ,T> { //当Test类模板的两个类型参数完全相同时，使用这个实现
      void add(T a, T b){
        cout << "void add(T1 a, T2 b)" <<endl;
        cout << a + b << endl;
    }
    void print(){
        cout << "class Test<T, T>" <<endl;
    }
};
template
<> //完全特化
class Test <void* a, void* b>{
    public:
     void add(void* a, void* b){
        cout << "error" <<endl;
    }
}
int main(){
    Test<int, float> t1;
    Test<long, long> t2;
    Test<void*, void*>t3;
    t1.add(1,2.5);
    t2.add(5, 5);
    t3.add(NULL, NULL);
    return 0;
}
```

类模板特化注意事项

- 特化只是模板的分开实现

    - 本质上是同一个类模板
- 特化类模板的使用方式是统一的
    - 必须显示指定每一个类型参数

问题

- 类模板特化与重定义有区别吗？函数模板可以特化吗？
- 重定义和特化的不同
    - 重定义
        - 一个类模板和一个新类（或者两个类模板）
        - 使用的时候需要考虑如何选择的问题
    - 特化
        - 以统一的方式使用类模板和特化类
        - 编译器自动优先选择特化类
- 函数模板只支待类型参数完全特化
- ```
  template
  < typename T > //函数模板定义
  bool Equal(T a, Tb)
  {
      return a== b;
  }
  template
  < > //函数模板完全特化
  bool Equal<void*>(void* a, void* b)
  {
      return a == b;
  }
  ```

```
template
< typename T>
bool Equal(T a ,T b){
    return a == b;
}

template
<>
bool Equal<double>(double a, double b){
    const double delta = 0.000001;
    double r = a - b;
    return (-delta < r) && (r < delata);
}

bool  Equal(double a, double b){
    const double delta = 0.000001;
    double r = a - b;
    return (-delta < r) && (r < delata);
}

int main(){
    //放弃重载的全局，使用函数特化版
    Equal<>(0.001, 0.001);
    Equal<double>(0.001, 0.001);
    return 0;
```
```

```
}
```

工程中的建议

- 当需要重载函数横板时，优先考虑使用模板特化
- 当模板特化无法满足需求，再使用函数重载！

小结

- 类模板可以定义任意多个不同的类型参数
- 类模板可以被部分特化和完全特化
- 特化的本质是模板的分开实现
- 函数横板只支持完全特化
- 工程中使用模板特化代替类（函数）重定义

4.5 数组类模板

4.6 智能指针类模板

- 智能指针的意义
  - 现代 C+ +开发库中最重要的类模板之一
  - C+ +中 自动内存管理的主要手段
  - 能够在很大程度上避开内存相关的问题
- `STL` 中的智能指针 `auto_ptr`
  - 生命周期结束时，销毁指向的内存空间
  - 不能指向堆数组，只能指向堆对象（变量）
  - **一片堆空间只属于一个智能指针对象**
  - 多个智能指针对象不能指向同一片堆空间

`auto_ptr` 对堆空间是独占的。

```
#include <iostream>
#include <string>
#include <memory>

using namespace std;

class test{
    string m_name;
public:
    test(const char* name){
        m_name = name;
    }
    void print(){
        cout << "I'm" << name << "," <<endl;
    }

    ~test(){
        cout << "goodbye" << m_name "," << endl;
    }
};
int main(){
    //智能指针本质是个类
```

```
    auto_ptr<test> pt(new test("Liu"));
    pt->print();

    cout << "pt = " << pt.get() << endl;
    cout << endl;
    //一篇堆空间只属于一个智能指针，这里堆空间的所属权发生了转移
    auto_ptr<test> pt1(pt);
    cout << "pt = " << pt.get() << endl; //pt = 0
    cout << "pt1 = " << pt1.get() << endl; //智能指针发生了转移
    pt1->print();
    return 0;
}
```

- `STL` 中的其它智能指针
  - `shared_ptr`
    - 带有引用计数机制，支持多个指针对象指向同一片内存
  - `weak_ptr`
    - 配合 `shared_ptr` 而引入的一种智能指针
  - `unique_ptr`
    - 一个指针对象指向一片内存空间，不能拷贝构造和赋值
- Qt 中的智能指针
  - `QPointer`
    - 当其指向的对象被销毁时，它会被自动置空（避免多次释放和野指针）
    - 析构时不会自动销毁所指向的对象（需要自己写 `delete`）
  - `QSharedPointer`
    - 引用计数型智能指针
    - 可以被自由地拷贝和赋值
    - 当引用计数为0时才删除指向的对象

```
#include <QPointer>
#include <QString>
#include <QDebug>
#include <QSharedPointer>
using namespace std;

class test : public QObject{
    QString m_name;
public:
    test(const char* name){
        m_name = name;
    }
    void print(){
        qDebug() << "I'm" << name << "," <<endl;
    }

    ~test(){
        qDebug()  << "goodbye" << m_name "," << endl;
    }
};
int main(){

    QPointer<test> pt(new test("liu"));
```

```
    QPointer<test> pt1(pt);
    QPointer<test> pt2(pt);
    delete pt;
    //下面三个都为空
    qDebug() << "pt = " << pt;
    qDebug() << "pt1 = " << pt1;
    qDebug() << "pt2 = " << pt2;

    QSharedPointer<test> spt(new test("liu"));
    //析构函数被调用，指向的引用对象减一 = 0
    QSharedPointer<test> spt1(spt1);
    QSharedPointer<test> spt1(spt2);

    spt -> print();
    spt1 -> print();
    spt2 -> print();
    //三次调用构造函数，一次析构函数，因为引用计数
    return 0;
}
```

创建智能指针类模板

`SmartPointer.h`

```
#ifndef _SMARTPOINTER_H
#define _SMARTPOINTER_H

template
<typename T>
class pointer{
    T* mp;
public:
    pointer(T* p = NULL){
        mp = p;
    }

    Test* operator ->(){
        return mp;
    }

    pointer(const pointer<T>& obj){
        mp = obj.mp;
        //转移智能指针后将其赋为空置
        const_cast<pointer<T>&>(obj).mp = NULL;
    }

    pointer& operater = (const pointer<T>& obj){
        if (this != &obj){
        mp = obj.mp;
        obj.mp = NULL;
    }
        return *this;
    }

    T& operator *(){
```

```
        return *mp;
    }

    T* get(){
        return mp;
    }
    ~pointer(){
        delete mp;
    }
}


int main(){
    for(int i = 0; i < 5; i++){
        pointer p = new Test(i);
        cout << p->value() << endl;
    }
    return 0;
}

#endif
```

`main.cpp`:

```
#include <iostream>
#include <string>
#include <memory>
#include "SmartPointer.h"
using namespace std;

class test{
    string m_name;
public:
    test(const char* name){
        m_name = name;
    }
    void print(){
        cout << "I'm" << name << "," <<endl;
    }

    ~test(){
        cout << "goodbye" << m_name "," << endl;
    }
};
int main(){
    //智能指针本质是个类
    //跟auto_ptr的结果一致
    pointer<test> pt(new test("Liu"));
    pt->print();

    cout << "pt = " << pt.get() << endl;
    cout << endl;
    //一篇堆空间只属于一个智能指针，这里堆空间的所属权发生了转移
    pointer<test> pt1(pt);
    cout << "pt = " << pt.get() << endl; //pt = 0
    cout << "pt1 = " << pt1.get() << endl; //智能指针发生了转移
    pt1->print();
```

```
    return 0;
}
```

小结

- 智能指针 C+ +中 自动内存管理的主要手段
- 智能指针在各种平台上都有不同的表现形式
- 智能指针在各种平台上都有不同的表现形式
- `STL` 和 `Qt` 都提供了对智能指针的支持

4.7 单例类模板

需求的提出

- 在架构设计时，某些类在整个系统生命期中最多只能有一个对象存在（`Single Instance`）。

问题

- 如何定义一个类，使得这个类最多只能创建一个对象？

单例模式

- 要控制类的对象数目， **必须对外隐藏构造函数**
- 思路
    - 将构造函数的访问属性设置为 `private`
    - 定义 `instance` 并初始化为 `NULL`
    - 当需要使用对象时，访问 `instance` 的值
        - 空值：创建对象 并用 `instance` 标记
        - 非空值：返回 `instance` 标记的对象

```
#include <iostream>
#include <string>

using namespace std;

class Sobject{
    //
    static Sobject* c_instance;
    Sobject(const Sobject&);
    Sobject& operator = (const Sobject&);
    Sobject(){

    }
public:
    static Sobject* get_instance();

};

Sobject* Sobject::c_instance = NULL;
Sobject* Sobject::get_instance(){
    if(c_instance == NULL){
        c_instance = new Sobject();
    }
    return c_instance;
}
```

```
int main(){
    //整个系统并不释放这个对象
    Sobject* s = Sobject::get_instance();
    Sobject* s1 = Sobject::get_instance();
    return 0;
}
```

存在的问题

- 需要使用单例模式时
    - 必须定义静态成员变量 `c_instance`
    - 必须定义静态成员函数 `GetInstance()`

解决方案

- 将单例模式相关的代码抽取出来开发单例类模板。当需要单例类时，直接使用单例类模板。

```
#include <iostream>
#include <string>
#include "Singleton.h"
using namespace std;

class Sobject{
    friend class Singleton<Sobject>;
    Sobject(const Sobject&);
    Sobject& operator = (const Sobject&);
    Sobject(){

    }
};

public:
   void print(){

   }

};
int main(){
    Sobject* s = Singleton<Sobject>::get_instance();
    Sobject* s1 = Singleton<Sobject>::get_instance();
    Sobject* s2 = Singleton<Sobject>::get_instance();

    s->print();
    s1->print();
    s2->print();
    return 0;
}
```

`Singleton.h`:

```
#ifndef _SINGLETON_H
#define _SINGLETON_H

template
<typename T>
class Singleton{
```

```
    static T* c_instance;
public:
    static T* get_instance();
};

template
<typename T>
T* Singleton<T>::c_instance = NULL;

template
<typename T>
T* Singleton<T>::get_instance(){
    if(c_instance == NULL){
        c_instance = new T();
    }
    return c_instance;
}
#endif
```

小结

- 单例模式是开发中最常用的设计模式之一
- 单例模式的应用使得一个类最多只有一个对象
- 可以将单例模式相关的代码抽象成类模板
- 需要使用单例模式的类直接使用单例类模板

4.8 C语言异常处理

C语言本身是没有异常处理的，但是可以实现。

异常的概念

- 程序在运行过程中可能产生异常
- 异常 `Exception` `Bug` 的区别
    - 异常是程序运行时可预料的执行分支
    - `Bug` 是程序中的错误，是不被预期的运行方式

异常处理

- 异常（`Exception`）与 `Bug` 的对比
    - 异常
        - 运行时产生除0的情况
        - 需要打开的外部文件不存在
        - 数组访问时越界
    - `Bug`
        - 使用野指针
        - 堆数组使用结束后未释放
        - 选择排序无法处理长度为0的数组

异常处理的方式

- C语言的处理方式:if...else...

- ```
  void func(...){
      if(判断是否产生异常){
          正常情况代码逻辑;
      }
      else
          异常情况代码逻辑;
  }
  ```

```
#include <stdio.h>
double divide(double a, double b, int* valid){
    const double delta = 0.000000000001;
    double ret = 0;
    if(!((-delta < b) && (b < delta))){
        ret a / b;
        *valid = 1;
    }
    else //valid来区分是出异常，还是0/1此类情况
        *valid = 0;
    return ret;
}
int main(int argc, char* argv[]){
    if( valid ){
        cout << "r = " << r << endl;
    }
    else
        cout << "Devide by zero" << endl;
    return 0;
}
```

缺陷

- `divide` 函数有 3个参数，难以理解其用法
- `divide` 函数调用后必须判断 `valid` 代表的结果
  - 当 `valid` `true` ，运算结果正常
  - 当 `valid``false`，运算结果异常
- 通过 `setjmp() longjmp()` 进行优化
  - `int setjmp(jmp_buf env)`
    - 将当前上下文保存在 `jmp_buf` 结构体中
  - `void longjmp(jmp_buf env, int val)`
    - `jmp_buf` 结构体中恢复 `setjmp()` 保存的上下文
    - 最终从 `setjmp` 函数调用点返回，返回值为 `val`

```
#include <stdio.h>

static jmp_buf env;
double divide(double a, double b){
    const double delta = 0.000000000001;
    double ret = 0;
    if(!((-delta < b) && (b < delta))){
        ret a / b;
    }
    else //valid来区分是出异常，还是0/1此类情况
        longjmp(env, 1);
```
```

```
    return ret;
}
int main(int argc, char* argv[]){
    if( setjump(env) == 0 ){
        //程序先执行下面这句，然后longjmp跳转到上面一句，env = 1 !=0。
        //破坏了结构化设计，工程很少使用。
        double r = divide(1, 0);

        cout << "r = " << r << endl;
    }
    else
        cout << "Devide by zero" << endl;
    return 0;
}
```

缺陷

- `setjmp longjmp` 的引入
    - 必然涉及到使用全局变量
    - 暴力跳转导致代码可读性降低
    - 本质还是`if...else...`异常处理方式

C语言中的经典异常处理方式会使得程序中逻辑中混入大量的处理异常的代码。

正常逻辑代码和异常处理代码混合在一导致代码迅速膨胀，难以维护。。。

问题

- C++中有没有更好的异常处理方式?

小结

- 程序中不可避免的会发生异常
- 异常是在开发阶段就以预见的运行时问题
- C语言中通过经典的 `if else` 方式处理异常
- C++中存在更好的异常处理方式

4.9 C++ 中的异常处理

- C++内置了异常处理的语法元素 `try...catch...`
    - `try` 语句处理正常代码逻辑
    - `catch` 语句处理异常情况
    - `try` 语句中的异常由对应的 `catch` 语句处理
    - ```
      try {
          double r= divide( 1, 0 );
      }
      catch(){
          cout << "error" << endl;
      }
      ```
- C++通过`throw`语句抛出异常信息
      ```

```
double divide(double a, double b) {
    const double delta= 0 . 000000000000001 ;
    double ret = 0;
    if(!((-delta < b) && (b <delta))) {
        ret = a / b;
    }
    else {
        throw 0 ;
    }
return ret;
}
```

- C+ +异常处理分析
  - `throw` 抛出的异常必须被 `catch` 处理
    - 当前函数能够处理异常 程序维续往下执行
    - 当前函数无法处理异常 则函数停止执行 并返回
  - 未被处理的异常会顺着函数调用栈向上传播，直到被处理为止，否则程序将停止执行

```
#include <stdio.h>

double divide(double a, double b){
    const double delta = 0.000000000001;
    double ret = 0;
    if(!((-delta < b) && (b < delta))){
        ret a / b;
    }
    else
        throw 0;
    return ret;
}
int main(int argc, char* argv[]){
        try{
            double r = divide(1, 1);
            cout << "r = " << r << endl;
        }
        catch(...){
            cout << "Devide by zero" << endl;
        }
    return 0;
}
```

同一个 `try` 语句可以跟上多个 `catch` 语句

- `catch` 语句可以定义具体处理的异常类型
- 不同类型的异常由不同的 `catch` 语句负责处理
- `try` 语句中可以抛出任何类型的异常
- `catch(...)` 用于处理所有类型的异常
- 任何异常都只能被捕获 `catch` 一次

异常处理的匹配规则

- 异常抛出后，至上而下严格匹配每一个 `catch` 语句处理的类型。异常处理匹配时不进行任何的类型转换。

```
try{
    throw 1;
}
catch(Type1 t1){

}
catch(Type2 t2){

}
catch(TypeN tn){

}
```

```
#include <stdio.h>
#include <string>
void Demo1(){
    try{
        throw 1;
    }
    catch(char c){
        cout << "catch(char c)" << endl;
    }
    catch(short c){
        cout << "catch(short c)" << endl;
    }
    catch(double c){
        cout << "catch(double c)" << endl;
    }
    catch(...){
        cout << "catch(...)" << endl;
    }
}
int main(int argc, char* argv[]){
        Demo1();
        try{
            double r = divide(1, 1);
            cout << "r = " << r << endl;
        }
        catch(...){
            cout << "Devide by zero" << endl;
        }
    return 0;
}
```

小结

- C+ +中直接支持异常处理的概念
- `try...catch...`是C++中异常处理的专用语句
- `try`语句处理正常代码逻辑，`catch`语句处理异常情况
- 同一个`try`语句可以跟上多个`catch`语句
- 异常处理必须严格匹配，不进行任何的类型转换

`catch`语句块中可以抛出异常

```
try{
    func();
}
catch(int i){
    throw i; //将捕获的异常重新抛出
}
catch(...){
    throw;
}
//catch中抛出的异常需要外层的try...catch...捕获
```

问题

- 为什么要在 `catch` 中重新抛出异常?

为了工程开发中统一异常处理。

```
#include <stdio.h>
#include <iostream>
#include <string>
void Demo1(){
    try{
        try{
            throw 0;
        }
        //在catch中重新扔出
        catch(int i){
            throw i;
        }
        catch(...){
            cout << "iner" <<endl;
            throw;
        }
    }
    catch(...){
        cout << "outer" <<endl;
    }
}
/*
    假设当前的函数是第三方库的函数，我们无法修改源代码
    函数名 void func(int i)
    抛出的异常的类型：int
        -1 参数异常
        -2 运行异常
        -3 超时异常
*/

void func(int i){
    if(i < 0){
        throw -1;
    }
    if( i > 100){
        throw -2;
    }

    if( i ==11 ){
        throw -3;
```

```
    }
    cout << "Run" <<endl;
}

void MyFunc(int i){
    try{
        func(i);
    }
    catch(int i){
        switch(i){
            case -1:
                throw "";
                break;
            case -2:
                throw "";
                break;
        }
    }
}
int main(int argc, char* argv[]){
    //Demo1();

    try{
        Myfunc(11);
    }
    catch(const char* i){
        cout << "Eecption" <<endl;
    }
    return 0;
}
```

- 异常的类型可以是自定义类类型
- 对于类类型异常的匹配依旧是至上而下严格匹配
- **赋值兼容**性原则在异常匹配中依然适用
- 一般而言
    - 匹配子类异常的 `catch` 在上部
    - 匹配父类异常的 `catch` 放在下部

```
#include <stdio.h>
#include <iostream>
#include <string>
class Base{

};

class exception : public Base{
  int m_id;
    string m_desc;
public:
    exception(int id, string desc){
        m_id = id;
        m_desc = desc;
    }
    int id(){
        return id;
```

```
    }

    string desciption const{
        return m_desc;
    }
};

void MyFunc(int i){
    try{
        func(i);
    }
    catch(int i){
        switch(i){
            case -1:
                throw exception(-1, "Invalid Parameter");
                break;
            case -2:
                throw "";
                break;
        }
    }
}
int main(){
    try{
        MyFunc(11);
    }
    catch(const Base& e){ //父类在上面时，下面的子类将无法运行
        cout << "const Base& e" <<endl;
    }
    catch(const exception& e){
        cout << "exceptiom info:" <<e.id() <<endl;
    }
    return 0;
}
```

- 在工程中会定义一系列的异常类
- 每个类代表工程中可能出现的一种异常类型
- 代码复用时可能需要重解释不同的异常类
- 在定义 `catch` 语句块时推荐使用引用作为参数

标准库中的异常都是从 `exception` 类派生的

`exception` 类有两个主要的分支

- `logic_error`
    - 常用于程序中的可避免逻辑错误
- `runtime_error`
    - 常用于程序中无法避免的恶性错误
- 自行查看C++标准库常见异常

4.10 C++中的类型识别

在面向对象中可能出现下面的情况

- 基类指针指向子类对象
- 基类引用成为子类对象的别名

类型识别

- 静态类型 - 变量（对象） 自身的类
- 动态类型 - 指针（引用）所指向对象的实际类型

```
void test(Base* p){
    /*危险的转换方式*/
    Derived* d = static_cast<Derived*> (b) ;
}
//基类指针是否可以强制类型转换为子类指针，取决于动态类型
```

问题

- C+ + 中如何得到动态类型?

动态类型识别

解决方案 - 多态

1. 在基类中定义虚函数返回具体的类型信息
2. 所有的派生类都必须实现类型相关的虚函数
3. 每个类中的类型虚函数都需要不同的实现

```
#include <stdio.h>
#include <iostream>
#include <string>

using namespace std;

class Base{
public:
    virtual string type(){
        return "Base";
    }
};

class Derived : public Base{
public:
    string type(){
        return "Derived";
    }
};

class child : public Base{
public:
    string type(){
        return "child";
    }
} ;

void test(Base* p){
    if( b-> type() = "Derived"){
        Derived* d = static_cast<Derived*> (b) ;
    }
    //
    //cout << dynamic_cast<Derived*>(b) <<endl;
}
```

```
int main(){
    Base b;
    Derived d;
    child c;

    test(&b);
    test(&d);
    test(&c);
    return 0;
}
```

多态解决方案的缺陷

- 必须从基类开始提供类型虚函数
- 所有的派生类都必须重写类型虚函数
- 每个派生类的类型名必须唯一

类型识别关键字

C++ 提供了 `typeid` 关键字用于获取类型信息

- `typeid` 关键字返回对应参数的类型信息
- `typeid` 返回一个 `type_info` 类对象
- 当 `typeid` 的参数为 `NULL` 时将抛出异常
- `typeid` 关键字的使用
    - ```
      int i = 0;
      const type_info& tiv = typeid(i);
      const type_info& tii = typeid(int);
      cout << (tiv == tii) <<endl;
      ```
- `typeid` 的注意事项
    - 当参数为类型时，返回静态类型信息
    - 当参数为变量时：
        - 不存在虚函数表 - 返回静态类型信息
        - 存在虚函数表 - 返回动态类型信息

```
#include <stdio.h>
#include <iostream>
#include <string>

using namespace std;

class Base{
public:
    virtual ~Base{
        return "Base";
    }
};

class Derived : public Base{
public:

};
```
```

```
void test(Base* p){
    cosnt type_info& tb = typeid(*p);
    //cout << dynamic_cast<Derived*>(b) <<endl;
}

int main(){
    int i = 0;
    const type_info& tiv = typeid(i); //int 静态类型信息
    const type_info& tii = typeid(int);
    cout << (tiv == tii) <<endl;
    Base a;
    test(&a);
    //type在不同的编译器是不同的
}
```

小结

- C++中有静态类型和动态类型的概念
- 利用多态能够实现对象的动态类型识别
- `typeid` 是专用于类型识别的关键字
- `typeid` 能够返回对象的动态类型信息

4.11 经典问题解析

面试问题

1. 编写程序判断一个变量是不是指针;

拾遗

- C++中仍然支持 C语言中的可变参数函数
- C++编译器的匹配调用优先级
    - 重载函数
    - 函数模板
    - 变参函数

```
#include <stdio.h>
#include <iostream>
#include <string>

using namespace std;
void test(int i){
    cout << "void test(int i){" <<endl;
}

template
<typename T>
void test(T v){
    cout << "void test(T v){" <<endl;
}

void test(...){
    cout << "void test(...){" <<endl;
}
int main(){
```

```
    int i = 0;
    test(i);
    return 0;
}
```

思路

- 将变量分为两类： 指针 vs 非指针
- 编写函数
    - 指针变量调用时返回true;
    - 非指针变量调用返回false;

函数模板和变参函数的化学反应

```
template
<typename T>
bool IsPtr (T* v) // match pointer
{
    return true ;
}
bool IsPtr (...) // match non-pointer
{
    return false ;
}
```

```
#include <stdio.h>
#include <iostream>
#include <string>

using namespace std;

class test{
public:
};
template
<typename T>
bool IsPtr (T* v) // match pointer
{
    return true ;
}
bool IsPtr (...) // match non-pointer
{
    return false ;
}

int main(){
    int i = 0;
    int* p = &i;

    cout << "p is a pointer" << IsPtr(p) <<endl; //true
    cout << "p is a pointer" << IsPtr(i) <<endl; //false
    return 0;
}
```

存在的缺陷：

- 变参函数无法解析对象参数，可能造成程序崩溃！

进一步的挑战：

- 如何让编译器精确匹配函数，但不进行实际的调用？

```
#include <stdio.h>
#include <iostream>
#include <string>

using namespace std;

class test{
public:
};
template
<typename T>
bool IsPtr (T* v) // match pointer
{
    return true ;
}
bool IsPtr (...) // match non-pointer
{
    return false ;
}
//定义宏
#define ISPTR(p) (sizeof(IsPtr(p)) == sizeof(char))
int main(){
    int i = 0;
    int* p = &i;

    cout << "p is a pointer" << IsPtr(p) <<endl; //true
    cout << "p is a pointer" << IsPtr(i) <<endl; //false
    return 0;
}
```