

OOP-Review

RandomStar

目录

OOP-Review	1
1. C++ 的新特性	2
1.0 输入输出流	2
1.1 变量和动态内存分配	2
1.2 引用 Reference	4
1.3 const 类型	5
2. 类 class	8
2.0 类的基本概念	8
2.1 构造函数和析构函数	8
2.2 static 类型	12
2.3 Inline Function 内联函数	14
2.4 继承 Inheritance	14
2.5 友元 friend	16
2.6 多态和虚函数	17
2.7 虚函数使用总结	22
2.8 强制类型转换	26
3. 重载	29
3.0 函数的重载	29
3.1 运算符的重载	30
3.2 输入输出流的重载	31
4. 模板	32
4.1 namespace 命名空间	32
4.2 template 编程	33
4.3 STL 和迭代器	35
5. Exceptions 异常处理	36
6. Smart Pointer 智能指针	38

1. C++ 的新特性

1.0 输入输出流

- C++ 可以使用输入输出流 (cin, cout) 进行输出, 比如 `cout<<"Hello World";` 需要包含头文件 `#include<iostream>`
- 文件的输入输出, 使用 `ofstream` 和 `ifstream`

```
#include<iostream>
#include<fstream>
#include<istream>
#include<string>

using namespace std;

int main()
{
    string str="Hello World!";
    ofstream fout("out.txt");
    fout<<str<<endl;
    ifstream fin("out.txt");
    string str1,str2;
    fin>>str1>>str2;
    return 0;
}
```

1.1 变量和动态内存分配

- C++ 中的变量类型
 - global variable 全局变量, 存储在全局变量区
 - * 可以在不同的 cpp 文件之间共享, 可以使用关键字 `extern` 来使用别的 cpp 文件中的全局变量
 - static global variable 静态全局变量, 不能在 cpp 文件之间共享
 - local variable 存储在栈区上

- static local variable 静态局部变量

- * 存储在全局变量区

- * 在初次使用的时候初始化, keeps its value between visit to the function

- allocated variable 动态分配的变量

- * 存储在内存的堆结构中

```
string s="hello";  
string *ps=&s;  
(*ps).length();  
ps->length();
```

- C++ 动态内存分配

- new 用于动态分配内存给变量, 如 new int, new double[1000]

- * 与 malloc 的区别: **malloc** 不执行类的构造函数, 而 new 出新的对象的时候会执行对象的构造函数 new Class_Name[x] 会执行 x 次 Class_Name 的构造函数

- * 内存泄漏 Memory Leak

```
int *p = new int;  
*p = 123;  
p = new int;
```

- 上面这段代码中一开始为指针 p 分配了一段内存空间并赋值了 123, 但是第三行代码又为 p 赋值了一段新的内存空间, 原来的内存空间存储了 123, 但是这一段内存空间已经没有指针指向, 因此不能访问, 也不能删除, 造成了内存泄漏

- delete 用于删除动态分配的内存

- * 用法 delete p; delete[] p;

- * 和 new 类似, delete 会执行所删除对象的构造函数, 不能 delete 没有定义过的变量, 同一个变量不能 delete 两次

- * 两个指针 p1,p2 指向同一个数据，如果 p1 被 delete 了，p2 也不能访问原本 p1 指向的变量的值，因为 delete 删除的是内存里的数据

```
int *p1 = new int;
int *p2 = new p1;
*p2 = 1;
delete p1;
cout<<*p2<<endl; // error!!!
```

- 指向同一个数据的两个指针实际上只是两个不同的变量名而已，delete 删除的不是变量名而是数据

1.2 引用 Reference

- a new type in C++, 相当于给变量取了一个别名，使用方法为 `type &refname = name;` 引用的对象不能是表达式
- 引用和指针的区别
 - 不能定义空引用，引用必须连接到一块合法的内存
 - 一旦引用被初始化为一个对象就不能更改
 - 引用在创建的时候必须要初始化

```
int *f(int *x)
{
    (*x)++;
    return x;
}
```

```
int &g(int &x)
{
    x++;
    return x;
}
```

```
int x;
```

```

int &h()
{
    return x;
}

int main()
{
    int a=0;
    f(&a);
    g(a);
    h()=16; //这里全局变量 x 被赋值为 16
}

```

1.3 const 类型

- 用于定义常量类型，如 `const int x=12345;`, `const` 类型的变量在初始化之后就不能改变其值，`const` 型变量不能在连接单元外使用
- Run-time constants 运行时常量
 - 数组的定义时的长度值必须在**编译期**就已知，所以宏定义中的常数可以作为数组长度，而 `int n; int a[n]` 这样的语法就是错误的，不过现在似乎有了编译器优化，在 Dev-cpp 中这样写也可以通过编译

```

const int size=100;
int a[size];//OK

int x;
cin>>x;
const int size=x;
int a[size];//Error!!!

```

- `const` 和指针 `pointer`
 - 常量指针: `char * const p = "abc";` 不能赋予这个指针新的地址，相当于地址是 `const` 类型，但是 `p` 指向的值可以改变

- `const char* p = "abc";` 这种情况下 `p` 指向的是一个 `const char` 类型的值，因此指向的值不能改变，而指针指向的对象可以改变

```
//第一种情况是 q 是一个 const 指针，但是 q 指向的东西可以变
char * const q = "abc";
*q = 'c'; // OK
q++; // Error!
//第二种情况下，(*q) 是一个 const 的值，此时 q 所指向的值不能变
const char *p = "abc";
*p = 'c'; //error!
//区别这三种东西
string p1 = "zyc";
const string *p = &p1;
string const* p = &p1;
string *const p = &p1;
```

- 不能将 `const` 类型的变量赋值给对应的指针，因为可能会带来 `const` 变量的改变，这是 `const` 类型的变量不允许的
- 可以把非 `const` 类型的值赋给对应的 `const` 型变量，函数中可以将参数设置为 `const` 类型表明这些参数在函数中不能被修改原本的值，也可以将返回值类型设置为 `const` 表示返回值不能被修改

```
#include<iostream>
using namespace std;

struct student{
    int id;
};

void foo(const student *ps)
{
    //*ps could not be changed in the function
    cout<<ps->id<<endl;
    cout<<(*ps).id<<endl;
}
```

```

void bar(const student &s)
{
    //s could not be changed in the function
    cout<<s.id<<endl;
}

```

```

int main()
{
    student s;
    s.id=2;
    foo(&s);
    return 0;
}

```

- `char * s="Hello World!";` 实质上是 `const char *` 类型，不要去修改 `s` 中的内容，这是一种未定义的行为 (undefined behavior)，应该写成 `char s[]="Hello World!";`

```

#include<iostream>
using namespace std;

```

```

int main()
{
    const char *s1="Hello World";
    const char *s2="Hello World";

    cout<<(void*)s1<<endl;
    cout<<(void*)s2<<endl;
    return 0;
    //输出的结果是 s1 和 s2 的地址，他们的结果是一样的
}

```

2. 类 class

2.0 类的基本概念

- C++ 中的对象 = 属性 + 操作 $objects = attributes + operations$, 对于面向对象的编程而言, 任何东西都是一个对象, 任何对象都有对应的类型, 程序就是一系列对象互相之间传递信息来完成功能
 - C++ 的类包含成员变量和成员函数
 - C++ 中 class 的定义, 具体实现和调用可以分成三个文件
 - * 头文件 header 是具体实现和调用之间的接口, 每个类的定义需要用 1 个头文件
 - :: 操作符可以用来访问类中的内容

2.1 构造函数和析构函数

- constructor 构造函数
 - 构造函数的函数名和类的名字相同, 可以传入一些参数用来初始化一个对象, 在类的对象被定义的时候会自动调用构造函数
 - default constructor 默认构造函数, 不需要参数也可以使用的构造函数
 - 初始化列表: 在函数签名后面, 大括号之前直接对类中定义的变量进行赋值
 - * const 类型的成员变量初始化只能用初始化列表
 - * 构造函数的执行分为两个阶段: 初始化阶段和函数执行阶段, 会先执行初始化列表里的赋值, 在进入函数主体进行对应的操作
- Destructor 析构函数
 - 析构函数的函数名是类名前面加一个 ~, 析构函数不需要参数, 在类的生命周期结束的时候会被编译器自动调用
- function overloading 函数重载:
 - 函数名相同而参数的个数和类型不同的几个函数构成重载关系, 一个类可以有多个不同的构造函数来解决不同情况下的构造

- default value: 缺省值，可以在函数参数表中直接声明一些参数的值，但是必须要从右往左，当传入的参数缺省时函数默认将已经声明的值作为参数的值

- constant object 常量对象

- 需要加 `const` 声明，在声明之后就不能改变这个对象内部变量的值
- 会有一些成员函数不能正常使用
- 在成员函数参数表后面加 `const` 可以成为 `const` 型成员函数，`const` 类型的成员函数不能修改成员变量的值

- * `const` 声明写在函数的开头表示函数的返回值类型是 `const`

- * `const` 声明写在函数签名的末尾表示这个成员函数不能修改类中定义的成员变量，被称为常成员函数

- 但是如果是成员变量中有指针，并 不能保证指针指向的内容不被修改

- `const` 类型的函数和非 `const` 类型的函数也可以构成重载关系，比如：

```
class A {
public:
    void foo() {
        cout << "A::foo();" << endl;
    }
    void foo() const {
        cout << "A::foo() const;" << endl;
    }
};

int main()
{
    A a;
    a.foo(); //访问的是非 const 类型的 foo
    const A aa;
    aa.foo(); //访问的是 const 类型的 foo
    return 0;
}
```

```
}
```

- `const` 类型的成员函数的使用规则如下：
- `non-const` 成员函数不能调用 `const` 类型对象的成员变量，而 `const` 类型可以访问
- `const` 类型函数不会改变任何成员变量的值
- 构成重载关系的时候，`const` 类型的对象只能调用 `const` 类型的成员函数，不能调用 `non-const`，而非 `const` 类型的对象优先调用 `non-const` 的成员函数，如果没有 `non-const` 再调用 `const` 类型的

- copy constructor 拷贝构造函数

- 把一个对象直接赋值给另一个对象，一般的形式为 `class_name(const class_name & copy_class_var)`，通过拷贝构造函数可以实现对象之间的互相赋值
- 如果定义变量时直接给变量用同类型的变量赋值，调用的就是拷贝构造函数，如果是定义之后再赋值，就是调用了重载之后的等号，比如下面这一段代码

```
class A {  
    A() {}  
    A(const A& a) {}  
    A& operator=(const A& a) {}  
};  
  
int main()  
{  
    A a;  
    A b = a; //调用拷贝构造函数  
    A c;  
    c = a; //调用重载之后的等号  
}
```

- C++ 中的拷贝：浅拷贝和深拷贝

- 浅拷贝：在原来已有的内存中增加一个新的指针指向这一段内存

- 比如 `string s1="zyc"; string s2(s1);` 就是一种浅拷贝
- 深拷贝：分配一块新的内存，复制对应的值，并定义一个新的指针指向这一块内存
- 缺省的拷贝构造函数和赋值运算符进行的都是浅拷贝
- 拷贝构造函数和赋值运算符的区别
 - * 拷贝构造函数是在对象被创建的时候调用的、
 - * 赋值运算符只能使用于已经存在的对象，也就是进行赋值之前，这个对象已经被某个构造函数构造出来了
- 例如

```
#include<cstring>
#include<iostream>
using namespace std;

struct Person{
    char *name;
    Person(const char *s){
        name=new char[strlen(s)+1];
        strcpy(name,s);
    }
    ~Person(){
        delete[] name;
    }
}

//不需要拷贝构造函数的一种方式
Person bar(const char *s)
{
    cout<<"in bar()"<<endl;
    return Person(s);
}
```

```

int main()
{
    Person p1("Trump");
    Person p2=p1;
    cout<<(void *)p1.name<<endl;
    cout<<(void *)p2.name<<endl; //会发现输出的地址是一样的，说明指针在默认情况下也进行了 copy

    return 0;
}

```

2.2 static 类型

- 类中的成员 (变量和函数) 分为两种
 - 静态成员：在类内所有对象之间共享
 - 实例成员：只能在某个具体的对象中调用
 - 静态函数不能访问实例成员
- 之前已经讲到了 static 类型的全局变量只在当前文件有效，不能通过 extern 跨文件调用，而函数中的 static 类型的变量在第一次调用的时候会被初始化，之后再调用该函数这个 static 类型的变量保持上一次函数调用结束时的值

```

#include<iostream>
using namespace std;

class A{
public:
    A() {cout<<"A::A()"<<endl;}
    ~A(){cout<<"A::~A()"<<endl;}
}

void f(int n)
{
    if(x>10)
        static A a;
}

```

```

        cout<<"f()"<<endl;
    }
    int main()
    {
        cout<<"start"<<endl;
        f(1);
        f(11);
        return 0;
    }

```

//此时只有在第二次调用时 *a* 才会被构造

//无论什么情况，*A* 的构造和析构函数只会被执行一次

- 类中的 static

- 类中定义的 static 类型的变量是**静态成员变量**，其值会在这个类的所有成员之间共享

- * **non-const 类型的静态成员变量需要在类的外面进行定义**，比如：

```

#include <iostream>
using namespace std;

class A {
public:
    static int count;
    A() {
        A::count++;
    }
};

```

`int A::count = 0;` // 在类的外部赋值的时候不需要说明 *static*，但是需要注明 *A::*，否则就是一个

```

int main()
{
    A* array = new A[100];
    cout<<A::count<<endl;
}

```

- * `const` 类型的静态成员变量作为类内共享的一个常量，也需要在类的外部进行定义，此时要写出关键字 `const`，并且这个静态成员变量是不能被改变的
- 静态成员函数：
 - * 可以访问类定义中的静态成员变量，但是不能直接访问普通的成员变量
 - * 需要在函数定义之前加 `static` 关键字

2.3 Inline Function 内联函数

- 需要在函数名前面加关键字 `inline`
 - 内联函数在编译期会被编译器在调用处直接扩展为一个完整的函数，因此可以减少运行时调用函数的 cost
 - 内联函数的定义和函数主体部分都应该写在头文件中
 - 本质是空间换时间
 - `class` 中的函数都是默认 `inline` 的
- an inline function is expanded in place, like a preprocessor macro, so the overhead of the function call is eliminated, 不需要函数调用产生的开销，由编译器直接优化，但是可能会使得需要编译的代码量增大 (虽然写的人是看不出来的)，主要作用是减小函数调用时的开销，一般在函数比较小的时候才会使用

2.4 继承 Inheritance

- composition 组合：把其他的类作为自己的成员变量
- Inheritance: 从基类中继承生成派生类
 - 派生类继承了基类的所有变量和成员函数
 - 派生类中不能直接访问基类的 `private` 的变量和成员函数，但是可以通过基类的成员函数来访问这些成员函数和变量
 - 派生类的构造函数
 - * 可以在派生类的构造函数中调用基类的构造函数

- * 派生类在被构造的时候会先调用基类的构造函数，再调用派生类的构造函数，析构的时候先调用派生类的析构函数，再调用基类的析构函数

- 如果派生类没有定义构造函数，则直接调用基类的构造函数
- 如果派生类定义了构造函数，在执行之前会先调用基类的构造函数，如果派生类的构造函数中没有显式调用基类的构造函数，则会选择调用基类的无参构造函数

```
class A {
public:
    int i;
    A(int ii = 0): i(ii) {
        cout<<"A(): "<<i<<endl;
    }
};
```

```
class B: public A {
public:
    int i;
    A a;
    B(int ii = 0): i(ii) {
        cout<<"B(): "<<i<<endl;
    }
};
```

```
int main()
{
    B b(100);
    return 0;
}
```

- * 当继承和组合两种情况同时出现时，先构造基类，再构造派生类中组合的其他类，再构造派生类，析构的时候类似

- Base class is always constructed first

- 就算组合的类在初始化列表或者构造函数中没有调用构造函数，C++ 编译器也会自动调用这个类默认的构造函数
- 派生类中可以对基类函数进行重载，此时如果派生类对象调用对应的函数按照派生类中的同名函数执行
 - * 仍然调用基类的方法 `object.Base::function()`
- class 和 struct 的区别：class 中的变量和函数默认为 **private**，struct 中的函数默认为 **public**
- 访问控制
 - * **public**：所有情况下可见
 - * **protected**：可以被自己/派生类和友元函数访问
 - * **private**：对于自己和友元函数可见
- 继承的种类：public，private，protected 继承，继承之后的基类变量的访问控制**取原本类型和继承类型中较严格的**
- 可以通过**强制类型转换把派生类的对象转化为基类的对象**

2.5 友元 friend

- 友元函数
 - 在类中声明一个全局函数或者其他类的成员函数为 **friend**
 - 可以使这些函数拥有访问类内 **private** 和 **protected** 类型的变量和函数的权限
 - 友元函数也可以是一个类，这种情况下被称为是友元类，整个类和所有的成员都是友元
 - 友元函数本身不是那个类的成员函数，函数签名里不需要 `className::` 来表示是这个类的成员函数，直接作为普通函数即可

```
class A {
private:
    int val;
```



```

public:
    A(int value): val(value) {
        cout<<"A()"<<endl;
    }
    friend void showValue(A a);
};

void showValue(A a)
{
    cout<<a.val<<endl;
}

```

2.6 多态和虚函数

- 多态 Polymorphism

- 同一段代码可以产生不同效果
- 对于继承体系中的某一系列同名函数，不同的类型会调用不同的函数
- 一般情况下，有继承关系的类之间有函数构成重载关系，依然会根据变量类型来调用对应的函数，比如：

```

#include <iostream>
using namespace std;

class A{
public:
    virtual void foo() {
        cout<<1<<endl;
    }
};

class B: public A{
public:
    virtual void foo() {

```

```

        cout<<2<<endl;
    }
};

```

```

int main()
{
    A a;
    B b;
    a.foo();
    b.foo();
    return 0;
}

```

* 此时运行的结果是 1 和 2，即 A 型的变量的 foo 函数是基类中的，B 类型的变量的 foo 函数是派生类中的

- 静态链接

- 函数的调用在程序开始运行之前就已经确定了
- 对于像下面这样的情况，基类的指针 (引用) 指向派生类，并且调用了基类中也存在的同名函数，最终调用的都是基类的同名函数

```

class Shape {
protected:
    int width, height;
public:
    Shape( int a=0, int b=0)
    {
        width = a;
        height = b;
    }
    int area()
    {
        cout << "Parent class area :" <<endl;
        return 0;
    }
}

```

```

    }
};

class Rectangle: public Shape{
public:
    Rectangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
        cout << "Rectangle class area :" <<endl;
        return (width * height);
    }
};

class Triangle: public Shape{
public:
    Triangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
        cout << "Triangle class area :" <<endl;
        return (width * height / 2);
    }
};

int main( )
{
    Shape *shape;
    Rectangle rec(10,7);
    Triangle tri(10,5);
    shape = &rec;
    shape->area();
    shape = &tri;
    shape->area();
    return 0;
}

```

- 虚函数 Virtual Function

- 一种用于**实现多态**的机制，核心理念是通过基类访问派生类定义的函数
 - * 这种方式称为**动态链接**
- 用于区分派生类中和基类同名的方法函数，需要将**基类**的成员函数类型声明为 `virtual`
 - * **基类中的析构函数一定要为虚函数**，否则会出现对象释放错误
- 纯虚函数: `virtual int func() = 0;` 表明该函数没有主体，基类中没有给出有意义的实现方式，需要在派生类中进行扩展
- **override 语法**: 派生类中可以用 `override` 关键字来声明，表示对基类虚函数的重载
- 虚函数需要借助指针和引用达到多态的效果
 - * 如果基类指针/引用**指向基类**，那就正常调用基类的相关成员函数
 - * 如果基类指针**指向派生类**，则调用的时候会调用派生类的成员函数
 - * **派生类指针不能指向基类**

```
class A{
public:
    virtual void foo(){
        cout<<"A"<<endl;
    }
};

class B{
public:
    virtual void foo(){
        cout<<"B"<<endl;
    }
}

int main()
{
    A *a=new B();
    a->foo(); //结果为 B
}
```

```
    return 0;
}
```

– 虚函数的实现方式：虚函数表 virtual table

- * 每一个有虚函数的类都会有一个虚函数表，该类的任何对象中都存放着虚函数表的指针，虚函数表中列出了该类的虚函数地址

- 虚函数表是一个指针数组，里面存放了一系列虚函数的指针

- 虚函数的调用需要经过虚函数表的查询，非虚函数的调用不需要经过虚函数表

- 虚函数表在代码的编译阶段就完成了构造

- 一个类只有一张虚函数表，每一个对象都有指向虚函数表的一个指针 `__vptr`

- * 多态的函数调用语句被编译成一系列根据基类指针所指向的对象存放的虚函数表的地址，在从虚函数表中查找地址调用对应的虚函数

- * 事实上虚函数会给函数加上一个新的参数，是一个指针，占用 4 字节

- 接口 (C++ 抽象类)

- 描述了一个类应该有的功能和行为，但是不用在这个类中实现，而是在派生类中实现

- 可以使用纯虚函数来实现抽象类的定义，比如：

```
class Shape {
public:
    virtual double getArea() = 0;
    Shape(int a, int b): length(a), width(b) {}
protected:
    int length;
    int width;
};
```

```
class Rectangle: public Shape {
```

```

public:
    double getArea() {
        return length * width;
    }
};

class Triangle: public Shape {
public:
    double getArea() {
        return length * width / 2;
    }
};

```

2.7 虚函数使用总结

- 情况 1: 基类和派生类都不是 virtual
 - 此时对于基类的对象和基类的指针，执行的就是基类的 f，对于派生类的执行的就是派生类的 f

```

class A {
public:
    void f() {
        cout << "af" << endl;
    }
};

```

```

class B: public A {
public:
    void f() {
        cout << "bf" << endl;
    }
};

```

```

int main()

```

```

{
    A a;
    B b;
    a.f();
    b.f();
    A* pb = &b;
    pb->f();
}

```

- 情况 2: 派生类中的同名函数是虚函数: 无影响, 和 1 一模一样
- 情况 3: 基类中的是虚函数, 派生类中不注明是虚函数: 此时派生类的对象和指向派生类的指针执行的都是派生类的函数 f, 基类的对象和指向基类对象的指针执行的都是基类的函数 f

- 总结:

- virtual 的虚函数关键字是向下负责的, 派生类声明 virtual 对基类无任何影响
- 对于指针和引用而言
 - * 不是虚函数的时候, 调用的函数取决于指针和引用的变量类型 (基类指针调用基类, 派生类指针调用派生类)
 - * 是虚函数的时候, 调用函数取决于指针和引用指向的变量类型 (指向基类调用基类, 指向派生类调用派生类)
 - * 当然如果派生类里没有新的同名函数, 那么执行的都是基类里的
 - * 要注意派生类指针不能直接指向基类的对象
 - * 如果虚函数里还需要调用其他函数, 调用的规则也和上面的一样, 比如下面有个历年卷上面的神题:
- 历年卷上的一个题目: 写出程序的输出

```

class B {
public:
    void f() {
        cout << "bf" << endl;
    }
}

```

```

    }
    virtual void vf() {
        cout << "bv" << endl;
    }
    void ff() {
        vf();
        f();
    }
    virtual void vff() {
        vf();
        f();
    }
};

```

```

class D: public B {
public:
    void f() {
        cout << "df" << endl;
    }
    void ff() {
        f();
        vf();
    }
    void vf() {
        cout << "dv" << endl;
    }
};

```

```

int main()
{
    D d;
    B* pb = &d;
    pb->f();
    pb->ff();
}

```



```

        pb->vf();
        pb->vff();
    }

```

* 这道题的分析过程如下：

- 首先调用 f，而 f 不是虚函数，所以根据指针类型调用了 B 中的 f，输出 bf
- 再调用 ff，因为 ff 也不是虚函数，所以调用 B 中的 ff，B 中的 ff 调用了 vf 和 f，而 vf 是虚函数，B 类型指针指向的是 D，所以调用 D 中的 vf，输出 dvf，调用 f 则和上面一样输出 bf
- 再调用 vf，由于 vf 是虚函数，所以要调用 D 中的 vf，输出 dvf
- 再调用 vff，虽然是虚函数但是 D 中没有定义同名函数，所以调用 B 中的 vff，vff 中调用 vf 和 f，同 2 一样输出的是 dvf 和 bf
- 所以最终的输出是

```

bf
dvf
bf
dvf
dvf
bf

```

* 这道题涵盖了单继承虚函数的所有情况

- 基类虚函数的优先级高于派生类中的需要强制类型转换的同名函数

– 比如下面一段代码中

```

class A {
public:
    virtual void f(int i) {
        cout << 1 <<endl;
    }
};

```

```

class B: public A {
public:
    virtual void f(double i) {
        cout << 2 << endl;
    }
};

int main()
{
    A* pa = new B;
    pa->f(1);
    return 0;
}

```

- 这里输出的结果是 1，事实上两个 f 并不构成虚函数的关系，因为 f(1) 中 1 是 int 类型，所以优先调用了对于 int 匹配度高的
- 事实上如果是 f(1.1) 输出的结果仍然是 1，并且 CLion 会提示参数需要类型转换
- 事实上两个 f 不构成虚函数的多态关系，所以调用哪个并不看指针指向的对象，而是看指针本身的类型！

2.8 强制类型转换

2.8.1 static_cast

- static_cast 用于数据类型的强制转换，有这样几种用法
 - 基本数据类型的转换，比如 char 转换成 int
 - 在类的体系中把基类和派生类的指针和引用进行转换
 - * 向上转换是安全的
 - * 向下转换是不安全的
 - * 只能在有相互联系的类型中进行相互转换，不一定包含虚函数
 - 把空指针转换成目标类型的空指针

- 把任何类型转换成 void 类型
- static_cast 不能转换掉有 const 的变量

2.8.2 const_cast

- const_cast 可以强制去掉 const 的常数特性，只能用在指针和引用上面
 - 常量指针被转化成非常量的指针，仍然指向原来的对象
 - 常量引用被转换成为非常量的引用，仍指向原来的对象
- 来看一段代码
 - 打印出来的结果是 a=10，而 p 和 q 所指向的值是 20，a 的地址和 pq 指向的地址是一样的
 - 事实上第五行的赋值是一种未定义行为，最好别用

```
const int a = 10;
const int *p = &a;
int *q;
q = const_cast<int *>(p);
*q = 20;
cout << a << " " << *p << " " << *q << endl;
cout << &a << " " << p << " " << q << endl;
```

2.8.3 reinterpret_cast

- reinterpret_cast 主要有三种用途
 - 改变指针或者引用的类型
 - 将指针或者引用转换成为足够长的整形
 - 将整型编程指针或者引用类型

2.8.4 dynamic_cast

- 跟其他几个不同，其他几个都是编译时完成的，dynamic_cast 是在运行时进行类型检查的

- 不能用于内置的基本数据类型的强制转换
- 如果成功的，将返回指向类的指针或者引用，转换失败的话会返回 NULL
- 转换时基类一定要有虚函数，否则无法通过编译
 - 原因是虚函数表名这个类希望可以用基类指针指向派生类，这样转换才有意义
- 在类的向上转换的时候，和 `static_cast` 效果一样，但是向下转换的时候比 `static_cast` 更安全，因此要求也更高
- 来看一段代码

```
int main()
{
    A a;
    B b;
    A *ap = &a;
    if(dynamic_cast<B*>(ap)) {
        cout << "OK1" << endl;
    }
    else {
        cout << "Fail" << endl;
    }
    if(static_cast<B*>(ap)) {
        cout << "OK2" << endl;
    }
    else {
        cout << "Fail" << endl;
    }
    ap = &b;
    if(dynamic_cast<B*>(ap)) {
        cout << "OK3" << endl;
    }
    else {
        cout << "Fail" << endl;
    }
}
```

```

    if(static_cast<B*>(ap)) {
        cout << "OK4" << endl;
    }
    else {
        cout << "Fail" << endl;
    }
    return 0;
}

```

- 运行的结果是第一个失败，其他的都成功
- 推测导致这个结果的原因：
 - * 当 ap 指向派生类的时候，进行强制类型转换变成派生类是可以成功的
 - * 当 ap 指向基类的时候，dynamic_cast 转换是否成功取决于指针指向的类型和即将转换的类型是不是一样，不一样就会失败，返回一个 NULL，而 static 是可以成功的
 - * 其实是更安全的机制导致 dynamic 的检查更多，要求更高

3. 重载

3.0 函数的重载

- 多个同名函数的参数的个数或者类型不相同，这几个函数就构成重载关系
 - 不能仅通过返回值类型的不同来形成函数的重载，事实上函数重载和返回值类型的关系不大
 - 调用的时候优先调用匹配度最高的重载函数，重载的优先级是：
 - * 先找完全匹配的普通函数
 - * 再找模板函数
 - * 再找需要隐式转换的函数
 - 当没有参数类型恰好匹配的重载函数的时候，就会将参数进行隐式转换之后来寻找可以调用的重载函数，此时如果重载函数有多个符合条件，就会产生 error

```

#include <iostream>
using namespace std;

void foo(int a, int b)
{
    cout<<1<<endl;
}

void foo(double a, double b)
{
    cout<<2<<endl;
}

int main()
{
    foo(1, 2);
    foo(1.0, 2.0);
    foo(1, 2.0); // 这一行是有语法错误的!
}

```

3.1 运算符的重载

- 重载的一些基本性质
 - 大部分运算符都可以重载
 - . :: ?: 之类的运算符不能重载, sizeof, typeid, static 之类的关键字不能重载
 - 不能重载不存在的运算符
 - 必须对于整个 class 或者 enumeration type 进行运算符的重载
 - 重载之后的运算符依然保持原来一样的运算优先级和操作数的个数
- 运算符重载的基本语法
 - 在类定义中对 member function 进行重载 return_type class_name::operator 运算符 (parameters) 此时参数为对应运算符所需参数-1

- 对外部函数进行重载返回值类型 `operator` 运算符 (参数表) 此时参数个数和对应的运算符的所需个数相同

* 如果要访问类的 `private` 内容, 需要声明这个重载函数为友元

* 单目运算符一般声明为成员函数, 双目运算符一般作为外部的函数

* `[]()` `->` 等运算符必须作为成员函数

- 前缀的 `++` 和后缀的 `++`

* 区别: 后缀的 `++` 参数表是有个 `int` 类型的变量作为参数的

```
Integer& operator++(); //prefix++
```

```
Integer operator++(int x); //postfix++
```

• 比较大小关系的运算符重载时, 可以通过代码的复用减少不必要的代码量

• 自定义的类型转换

- 例如 `Rational::operator double() const {};`

- 隐式类型转换:

* Single-argument constructors

* implicit type conversion operators

* 避免隐式转换的方法: 把复制构造函数声明为 `explicit`, 表示不能进行隐式的类型转换

3.2 输入输出流的重载

• stream 的种类

- Text stream 用 ASCII 码来解析, 包括 files 和 character buffers

- Binary stream 二进制数据, no translation

• stream 的类型:

- `cin` 标准输入, 使用流提取运算符 `>>`, 在 C++ 标准库的 `istream` 中定义

- cout 标准输出, 使用流插入运算符 <<, 在 ostream 中定义
- cerr 标准错误提示 (unbuffered error)
- clog 标准错误提示 (buffered error)
- 对 >> 和 << 的重载
 - Has to be a 2-argument free function

```
istream& operator>>(istream& is, T& obj){
    //special way to read in obj
    return is;
    //Return an istream& for chaining
}
//<< 的重载方式与之类似
ostream& operator<<(ostream& os, T& obj){
    return os;
}
```

- 控制输出的格式, 使用头文件 #include<iomanip>
- 其他的输入输出的方式:
 - int get() 支持单个字符读入
 - * Returns the **next character** in the stream
 - * Returns EOF if no characters left
 - get(char *buf, int limit, char delim='\n')
 - getline(char *buf, int limit, char delim='\n')

4. 模板

4.1 namespace 命名空间

- 使用命名空间来划分全局的各类类名可以避免名字的冲突, 可以在不同的命名空间里定义相同的变量名

- 在引用的时候加上命名空间的限制符就可以了
- 或者也可以用 `using namespace xxx` 来说明程序中接下来用哪个命名空间中的东西

```
namespace space1{
    string name = "randomstar";
}

namespace space2{
    string name = "ToyamaKasumi";
}

int main()
{
    cout<<space1::name<<endl;
    using namespace space2;
    cout<<name<<endl;
}
```

4.2 template 编程

- 模板编程是一种复用代码的手段，是 generic programming(泛型编程)，把变量的类型当作参数来声明
- 函数模板：使用关键字 `template` 后面加若干个变量作为类型声明一个函数模板，类型变量可以代替函数的返回值类型，参数类型和函数中的变量的类型
 - 函数模板需要实例化 (instantiation) 之后再使用，如果没有被调用就不会被实例化
 - * 实例化是讲模板函数中的模板替换为对应的变量类型，然后生成一个对应的函数
 - 函数模板在使用过程中不能发生参数和返回值的类型转换，必须要类型完全匹配才能使用

-

函数重载时候的优先级

- * 先找是否有完全匹配的普通函数
- * 再找是否有完全匹配的模板函数
- * 再找有没有通过进行隐式类型转换可以调用的函数

• 类模板

- 和函数模板差不多，`template` 中声明的既可以是类内各种需要变量类型的地方
- 模板类也可以继承非模板类，也可以继承模板类 (需要实例化)
- 非模板类可以继承自模板类 (需要实例化)

• Expression parameter

- 模板中的可以声明一些常数，`class` 和 `typename` 的类型名可以有默认值 `default value`，
比如 `template<typename T = int>`

```
template <class T, int bounds = 100>
class FixedVector{
public:
    T& operator[] (int x);
    T elements[bounds];
};

template <class T, int bounds>
T& FixedVector<T, bounds>::operator[] (int i){
    return elements[i]; //no error checking
}

//usage
FixedVector<int, 50> v1;
FixedVector<int, 10*10> v2;
FixedVector<int> v3 //default value
```

• Specialization 特化--用于萃取器

- 全特化：将模板类中所有的类型参数赋予明确的类型，并写了一个类名和主模板类名相同的类，这个类就是全特化类

- * 全特化之后，已经失去了 `template` 的特性

```
template<class T> bool compare(T x, T y)
{
    return x>y;
}
```

// 这个就是对上面写的模板函数的全特化

```
template<> bool compare(int x, int y)
{
    return x>y;
}
```

- 偏特化：只给模板类赋一部分的类型，得到的偏特化类/函数可以作为一个子模版使用

- * 还保留了一部分 `template` 的功能

- 模板类调用的优先级

- * 全特化类 > 偏特化类 > 主版本的模板类（就是直接调用模板类，最常见的用法）

- * 有隐式转换的优先级比较低，先考虑所有不需要隐式转换的模板，再考虑需要隐式转换的模板

4.3 STL 和迭代器

- STL=Standard Template Library 是标准库的一部分，使用 C++STL 可以减少开发时间，提高可读性和鲁棒性，STL 包含了：

- 容器
 - 迭代器
 - 函数

- STL 容器的使用：刻在 DNA 里，不需要整理

- 关于 map: map 的 key 必须是满足可以排序性质的, 如果是自定义的类需要重载 < 函数, 否则这个类不能作为 key 值
 - 迭代器: STL<parameters>::iterator xxx
 - 迭代器是一种顺序访问容器的方式: Generalization of pointers
 - 两个特殊的迭代器: begin() 和 end() 分别表示容器的头和尾
 - * 很多时候 end 并不能达到, 因此到结束的判断条件往往是 !=stl.end()
 - 迭代器支持的操作
 - iter++ iter+=2(不是任何容器的迭代器都可以)
 - * *iter
- ```
list<int> L;
list<int>::iterator li;
li=L.begin();
L.erase(li); // li=L.erase(li);
++li; //error!!!
```
- 可以看成是一种泛化的指针

## 5. Exceptions 异常处理

- 用于异常处理的语法
  - try{ } catch{ }: 捕获 throw 抛出的异常
    - \* catch (...) 表示捕捉所有可能的异常
    - \* try 括号中的代码被称为保护代码
  - throw 语句: 抛出异常
    - \* throw exp; 抛出一个表达式: throw 的参数可以是任何的表达式, 表达式中的类型决定了抛出的结果类型
    - \* throw; 只有在 catch 子句中有效, 把原本捕捉到的异常抛出

- 异常处理的执行过程
  - 程序按照正常的顺序执行，到达 `try` 语句，开始执行 `try` 内的保护段
  - 如果在保护段执行期间没有发生异常，那么跳过所有的 `catch`
  - 如果保护段的执行期间有调用的任何函数中有异常，则可以通过 `throw` 创建一个异常对象并抛出，程序转到对应的 `catch` 处理段
  - 首先要按顺序寻找匹配的 `catch` 处理器，如果没有找到，则 `terminate()` 会被自动调用，该函数会调用 `abort` 终止程序
    - \* 如果在函数中进行异常处理并且触发了 `terminate`，那么终止的是当前函数
    - \* 异常类型需要严格的匹配
  - 如果找到了匹配的 `catch` 处理程序，并且通过值进行捕获，则其形参通过拷贝异常对象进行初始化，在形参被初始化之后，展开栈的过程开始，开始对对应的 `try` 块中，从开始到异常丢弃地点之间创建的所有局部对象的析构
- C++ 中自带的异常的继承体系，定义在头文件 `<exception>` 中
  - `what` 方法给出了产生异常的原因，是异常类之间都有的公共方法，已经被所有的子异常类重载
  - 自定义的异常类：需要继承 `exception` 类
- 函数定义的异常声明
  - 可以在函数名后面加 `noexcept` 关键字，说明该函数在运行的过程中不抛出任何异常，如果还是产生了异常，就会调用 `std::terminate` 终止程序
  - 可以在函数声明中列出所有可能抛出的异常类型，比如 `double f(int, int) throw(int);` 如果是 `throw()` 表示不抛出一场，就算函数里有 `throw` 也不会执行
- `throw` 会导致一个函数没有执行完毕，但是在函数 `throw` 之前会执行所有局部变量的析构函数
  - 最好不要在析构函数中抛出异常

```
class T
{
```

```

 T(){
 cout<<"T()"<<endl;
 }
 ~T(){
 cout<<"~T()"<<endl;
 }
}

void foo()
{
 T t;
 throw 1;
}

int main()
{
 try{
 foo();
 }
 catch(...){
 cout<<"Caught!"<<endl;
 }
 return 0;
}

```

//运行的结果是

```

T()
~T()
Caught

```

## 6. Smart Pointer 智能指针

- 普通指针管理内存容易造成内存泄漏 (比如用完忘记释放), C++ 提供了智能指针用于内存的管理

- 智能指针使用了一种 RAII(资源获取即初始化) 技术对普通的指针进行了封装, 使得智能指针实质上是一个对象, 但是行为表现得像一个指针
- 智能指针的相关内容包含在头文件

- 标准库中支持的智能指针类型

- `std::unique_ptr` 只允许这个指针来管理对应的资源, 不允许指针之间的拷贝, 但是
  - \* 不允许进行指针的赋值和拷贝
  - \* 可以用 `move` 方法来移动指针的所有权, 比如 `unique_ptr<int> p2 = move(p1);`
  - \* 可以用 `release` 方法来释放指针的所有权

```
class A{
public:
 A(){
 cout<<"A()"<<endl;
 }
 ~A(){
 cout<<"~A()"<<endl;
 }
 void foo(){
 cout<<"1"<<endl;
 }
}

int main()
{
 {
 unique_ptr<A> pa(new (A));
 pa->foo();
 }
}
```

- `std::shared_ptr` 非独占的指针，可以多个指针管理一个对象。`shared_ptr` 使用引用计数，每一个 `shared_ptr` 的拷贝都指向相同的内存。每使用他一次，内部的引用计数加 1，每析构一次，内部的引用计数减 1，减为 0 时，自动删除所指向的堆内存。`shared_ptr` 内部的引用计数是线程安全的，但是对象的读取需要加锁

- \* 可以用 `use_count` 函数来查看某个对象拥有的指针的个数
- \* 可以用 `get` 函数来获取原始指针
- \* 不能用一个原始指针初始化多个 `shared_ptr` 否则会造成内存多次释放

```
#include<iostream>
#include<memory>
int main()
{
 int a = 10;
 std::shared_ptr<int> ptr_a = make_shared<int>(a);
 std::shared_ptr<int> ptr_b(ptr_a);
 cout<<ptr_a.use_count<<endl; //输出的结果是 2
}
```

- `std::weak_ptr` 是没用重载 `*` 和 `->` 操作符，没有普通指针具有的行为，最大的作用在于协助其他指针的工作，查看对象中的资源使用情况
- \* `weak_ptr` 可以通过一个 `shared_ptr` 或者 `weak_ptr` 来进行构造，获取观测权，但是 `weak_ptr` 不会造成指针引用计数的增加