2.C++程序结构

包含警戒

```
#pragma once
//codes


#ifndef  A1_H
#define  A1_H
//codes
#endif
```

头文件

先在本地(当前工程)的目录中查找相应的头文件；若找不到，再到系统目录中查找相应的头文件。

3.类型与变量

typedef

typedef本质上没有增加新类型

```
typedef unsinged char uchar;
```

enum

```
enum WeekDay{MON=1,TUR,WED,THU,FRI,SAT,SUN=0};
```

内部就是 int

```
enum class AAA:unsigned int{
        A='a' ,
        EF=120
};
int main(){
        cout<<(int)(AAA::A)<<endl;
        cout<<(int)(AAA::EF)<<endl;
}
```

不加 (int) 或不加 AAA:: 会报错。
枚举值可以指定类型、指定数值;
枚举值不再允许与 int 隐式转换;
使用时必须指明scope(即枚举类名，如上文 AAA)。

声明

```
extern int a;                   // 声明外部整型变量a
extern const int c;             // 声明外部整型常量c
int f(int);                     // 声明函数f
class A;                        // 声明类A
typedef int Int;                // 声明类型 Int
extern X anotherX;              // 声明外部变量anotherX
using N::d;                     // 声明名字d
```

函数外的 static 变量

```
static int c;
{
        static int a;
}
```

函数外的 static 变量(c)，表示该变量仅在此文件中生效;
块内的 static 变量(a)，表示该变量仅在块中生效。

4.指针、数组、引用、常量

奇怪的反例

```
class Q{
public:
        int a=11;
        void f(){intot(b),pt(32);}
private:
        const int b=22;
};
int main(){
        Q q;
        q.f();
        *((&q.a)+1)=5;
        q.f();
}
```

22 5

define

```
#define STR(x) #x
#define VAR(x) n##x
#define qwq(x) qwq_##x##_pwp
int n1=1,n2=4,n3=7,qwq_ovo_pwp=3434;
int main(){
        cout<<STR(abc)<<endl;
        cout<<VAR(2)<<endl;
        cout<<qwq(ovo)<<endl;
        cout<<__LINE__<<endl;
        cout<<__FILE__<<endl;
        cout<<__FUNCTION__<<endl;
}
```

abc
4

3434
16(原程序这句是第16行)
1.cpp
main

指针

```
int*p1;
```

普通指针

```
int const *p2;
const int *p2;
```

(二者等价)
指向常量的普通指针

```
int * const p3;
```

指向变量的常指针

```
const int * const p4;
```

指向常量的常指针

别名

```
int a=2;
int &b=a;
const int &c=b;
```

将不能通过 `c` 直接或间接修改 `a` 。

```
int &d=c;
```

Error: binding reference of type `int` to value of type `const int` drops `const` qualifier.
因为此处将使得：可以通过 `d` 修改 `a` ， 这就是间接通过 `c` 修改 `a` 。

5函数

函数重载

```
int Func();
int Func(int);
int Func(int,int);
int Func(int,int) const;
int Func(int) throw( );
int Func(int) throw(int,MyE,YourE);
int Func(MyClass obj);
```

多个同名函数，但它们具有不同参数类型、参数顺序、参数个数、const修饰、异常说明数时，可以同时存在，称为函数重载。
引用和指针类型参数，是否可以改变实参，可作为区分标志。
返回值类型、缺省参数、值类型参数的const型与非const型不作为区分标志。

```
void f(float a){printf("float");}
void f(double a){printf("double");}
int main(){f(1.0);}
```

double

```
void f(float a){printf("float");}
void f(double a){printf("double");}
int main(){f(1);}
```

Error: call to `f` is ambiguous.

缺省参数

缺省参数，应在函数原型(声明)中提供；若无函数原型，才在定义中给出。
缺省参数在编译时匹配，而不是运行时。
第一个带缺省值参数的右侧必须都有缺省值。

static

```
int &f(int x){
        static int a=x;
        return a;
}
int main(){
        intot(f(2)),pt(32);
        f(4)=6;
        intot(f(8)),pt(32);
}
```

6.类和对象

不能递归定义

不能递归定义，但可用指针。

抽象数据类型(ADT)

一个ADT：一个数学模型+可施加其上的操作集合(类型名称，数据集，数据间的关系，操作集)。
与具体表示无关；与现实世界无关；任意性和无穷性。

引用形式数据成员

必须使用初始化列表。
对象禁止赋值。

对象占用的存储空间

与非静态数据成员的个数、类型、是否含有虚函数有关；
与编译时字节对齐方式设置有关；
与成员函数的个数、静态数据成员的多少、类型无关；
与访问控制无关；
一定非0。

实例化

产生一个对象。
之前必须已经有类的完整定义。

7.成员函数

this指针

是一个关键字，也是一个保留字；
是非静态成员函数隐含的第一个形参；
作用域和生存期是在非静态成员函数的 {} 内，永远指向当前对象。
相当于：

```
T * const this;
```

内联函数

内联的作用：建议编译器在调用处直接展开函数代码，只是建议。
`inline` 关键字只在实现同时存在时，才有意义。

类外实现

```
class A{
public:
        int v;
        A(int n):v(n){printf("1");}
        ~A(){printf("3");}
        int f()const{return 1;}
private:
        void g(){printf("2");}
};
```

```
class A{
public:
        int v;
        A(int n);
        ~A();
        int f()const;
private:
        void g();
};
A::A(int n):v(n){printf("1");};
A::~A(){printf("3");}
int A::f()const{return 1;}
void A::g(){printf("2");}
```

两段代码大致等效。
注意第二段代码的 `f()` 的两个 `const` 需要同时存在。
重载运算符的类外实现请用自由函数式。

访问控制

默认是 `const` 。
`public`： 任何类都可访问;
`private`： 本类(不是本对象)或友元可以访问;
`protected`： 可继承的 `private` 。

const A

```
class Q{
public:
        int a=11;
        void g(){a=99;}
};
int main(){
        const Q q2;
        q2.g();
}
```

Error: `this` argument to member function `g` has type `const Q`, but function is not marked const.
`const` 的类也不能调用和修改类内数据的成员函数。

8.构造和析构

构造函数

创建对象的同时，进行初始化工作
无返回值。
可重载，可设置不同访问控制。

隐式调用

```
class A{
public:
        int a;
        A(int x):a(x){}
};
void f(const A&o){intot(o.a);}
int main(){f(233);}
```

233

```
class A{
public:
        int a;
        A(int x):a(x){}
};
void f(A&o){intot(o.a);}
int main(){f(233);}
```

Error: no matching function for call to `f`.
Candidate function not viable: no known conversion from `int` to `A &` for 1st argument.

```
class A{
public:
        int a;
        explicit A(int x):a(x){}
};
void f(const A&o){intot(o.a);}
int main(){f(233);}
```

Error: no matching function for call to `f`.
Candidate function not viable: no known conversion from `int` to `const A` for 1st argument.
`explicit` 关键字用于禁止隐式调用。

缺省值的自定义构造函数

```
class Name {
public:
        Name():n1(0),n2(0){intot(n1);}
        Name(int v1):n1(v1){intot(n1);}
        Name(int v1,int v2=999):n1(v1),n2(v2){intot(n1);}
private:
   int n1,n2;
};


Name a;
Name c(11,12);
```

011

```
Name b(100);
```

Error: call to constructor of `Name` is ambiguous.

缺省构造函数

由编译器提供。

```
class A{
public:
        int v;
        A(const A&a):v(a.v){}
};
class B{
public:
        int v;
        B(int n):v(n){}
};
```

当没有自定义普通构造函数时，提供 `public:A(){}` 。
当有自定义普通构造函数(如 `B(int n);`)时，不提供普通构造函数。
当没有自定义拷贝构造函数 `B(const B&){}` 时，提供 `public:B(const B&){}` 。
自定义的拷贝构造函数不当作普通构造函数，上面的 `A` 仍有缺省的无参构造函数 `public:A(){}` 。

类定义时指定初值

```
class A{
public:
        int a=1;
        A(){printf("%.2lf ",b);}
        A(int v):b(v*0.1){printf("%.2lf ",b);}
private:
        double b=2.0;
        static const int c=3;
};
int main(){
        A a(233);
        A b;
}
```

23.30 2.00
将会被构造函数覆盖。
对静态数据，只允许类内初始化常整型静态数据成员。

```
class B{
private:
        static int a=60;
        static const float b=2.5;
};
```

Error: non-const static data member must be initialized out of line.
Error: in-class initializer for static data member of type `const double` requires `constexpr` specifier.

初始化列表

必须：
常量数据成员；
没有无参构造函数的对象数据成员；
引用数据成员。p;

```
class B{public:B(int a){}};
class A{
public:
        A(){}
private:
        const double a;
        B b;
        int &c;
};
int main(){A a;}
```

Error: constructor for `A` must explicitly initialize the const member `a`, the member `b` which does not have a default constructor, and the reference member `c`.

析构函数

负责对象销毁前，最后需要执行的清理工作。
生存期结束后，自动执行。
无参，无返回值。

9.拷贝和赋值

人话

就是 `A(const A&)` 和 `=`。

浅拷贝

当没有自定义拷贝构造函数 `A(const A&){}` 时，提供 `public:A(const A&){}`。
对象数据成员：自动调用对象所属类的拷贝构造函数。
对象数据成员：按二块。

禁止类外拷贝

```
private:
        A(const A&a)=delete;
```

浅赋值

有非静态的引用成员时，禁止赋值。
其它成员类似浅拷贝。

深拷贝和深赋值

没说得那么玄乎，就自定义一个就彳亍了。

10.运算符重载

重载限定条件

不能重载三元运算符 `?:`；
操作数至少一个是自定义的类型，即不可改变内置类型的运算符语义。
运算符的结合律、优先级等固有性质不变。
不可重载的操作/运算符： `.` `::` `.*` `?:` `new` `delete` `sizeof` `typeid` 等。

二元运算符重载的形式

成员函数

```
class A{
public:
        int num;
        A(int n):num(n){}
        A operator+(const A&r)const{return A(num+r.num);}
};
```

自由函数

```
A operator+(const A&l,const A&r)const{return A(l.num+r.num);}
```

若量两个都有，则：
Error: use of overloaded operator `+` is ambiguous (with operand types `A` and `A`).

一元运算符重载

```
++a
a.operator++()
```

```
a++
a.operator++(0)
```

-> 重载

必须满足下列其中一个：
返回指针类型；
返回自定义类型，且该类型中重载了 `->` 。

仿函数(() 重载)

在类中 `operator()` ，使用对象就像调用函数。称这为仿函数，主要应用于模板库。
C++1z中，可使用lambda表达式取代仿函数。

11.动态内存管理

delete

```
class A{
public:
        int v;
        A():v(9){intot(3);}
        ~A(){intot(v);}
};
int main(){
        A a;
        A*p=&a;
        delete p;
        pt('a');
}
```

39 (Runtime Error)
若 `intot(v)` 换成 `intot(6)` ：
36a6

```
class A{
public:
        int v;
        A():v(9){intot(3);}
        ~A(){intot(v);}
};
int main(){
        A*p=new A;
        delete p;
        pt('a');
        A*q=new A[3];
        delete[] q;
        pt('b');
}
```

39a333999b
(这才是正确代码)

定位分配

```
int *p1=new int;
int *p2=new(p1+1) int;
int *p3=new(p1+2) int;
printf("%X %X %X ",p1,p2,p3);
```

FF6EE0 FF6EE4 FF6EE8
(运行多次，只有每个的前两位变化)

12.转换函数，名字空间，友元，嵌套类，流

→ 转换函数

自动调用相应构造函数

```
class A{
public:
        int v;
        A(int n):v(n+3){};
};
class B{
public:
        int v;
        B(int n):v(n+10){};
        B(const A a):v(a.v+100){};
};
void f(const A&a){intot(a.v);pt(32);}
void g(const B&b){intot(b.v);pt(32);}
int main(){
        f(2.0);
        g(A(0));
}
```

12 103 4

(f g 中无论是否加 const 或 & ，结果不变)

也可采用重载函数形式

```
class B{
public:
        int v;
        B(int n):v(n+20){};
        operator int()const{return v+100;}
};
class A{
public:
        int v=3;
        operator B()const{return B(v);}
};
void g(B b){intot(b);}
int main(){
        g(A());
}
```

123

若重载 B() int() 时加入禁止隐式调用标识符，即

```
explicit operator B()const{return B(v);}
explicit operator int()const{return v+100;}
```

则调用时需要显示给出，即

```
g(B(A()));
intot(int(b));


class B{
public:
        int v;
        B(double n):v(n+20){};
};
class A{
public:
        int v=3;
        operator int()const{return v+100;}
};
void g(B b){intot(b.v);}
int main(){
        g(B(A()));
}
```

123

调用顺序： `A()` `int(A)` `double(int)` `B(double)` `g(B)` 。

`g(B(A()))` 中 `B` 换为 `int` `double` `char` 均可行且等效；若改为 `g(A())` 则会出错。

名字空间

```
namespace A{
        int f();
        int v=2;
};
int A::f(){return 1;}
int main(){
        intot(A::f());
        intot(A::v);
}
```

12

匿名名字空间

```
namespace{
        int v;
}
```

会自动 `using namespace` 。无法在其他文件中使用。

名字汇入

```
namespace A{int v=3;};
namespace B{int v=4;};


using A::v;
intot(v);
```

3

```
{
        using A::v;
        intot(v);
        using B::v;
}
intot(v);
```

Error: target of using declaration conflicts with declaration already in scope; use of u ndeclared identifier `v` , did you mean `A::v` ?

友元

`friend` 关键字，使函数/类/成员函数可以调用 `private` 的成员。
友元的本质都是友元函数，没有友员数据；友元函数不是类的成员。
友元关系是单向的，且没有传递性。

```
class A;
// class B;
class C;
// int f(A);
class B{
public:
        C*pc;
        int f(A);
};
class C{
public:
        B*pb;
        int f(A);
};
class A{
        friend int f(A);
        friend class B;
        friend int C::f(A);
private:
        int v=3;
};
int B::f(A a){return a.v;}
int C::f(A a){return a.v;}
int f(A a){return a.v;}
int main(){
        A a;
        intot(f(a));
        intot(B().f(a));
        intot(C().f(a));
}
```

333

13.类间关系

四种关联

强关联： A.h 必须包含 B.h 才能编译成功，则 A 与 B 之间存在**强关联**；
硬关联： A 和 B 间有双向关系，且 B 与 A 之间有**强关联**，那么 A 与 B 之间存在**硬关联**；
弱关联： A.cpp 中必须包含 B.h 才能编译通过，则 A 与 B 之间存在**弱关联。**
软关联： A 只使用了 B 的指针或引用，那么 A 与 B 之间存在**软关联。**

编译期依赖性

类间的联系强弱顺序:
继承(类间的垂直关系)
硬联系(硬关联)

强联系(强关联)
弱联系(弱关联)
软联系(软关联)
降低编译期依赖性:
自定义类型的数据成员：使用指针形式
函数的自定义类型参数：使用指针和引用形式
函数返回的自定义类型：尽肯能不使用对象形式
适当使用前置声明和外联实现
缺点:
引用形式数据成员：生存期需要长于对象；对象不可赋值，必须使用初始化列表。
对象形式数据成员：定义该类前，必须有数据成员所属类的完整定义。

聚集关联

聚合： `B` ***has a*** `A` ，但 `B` 类不负责 `A` 类对象的生存与消亡。
组合： `B` ***contain a*** `A` ，且 `B` 类负责 `A` 类对象的生存与消亡。

14.类的设计

封装和信息隐蔽

面向对象三大特征之一。
封装：通过对客观事物的抽象，分析事物的本质特征，总结和提炼事物的行为和属性，并用类和对象表示的过程。
信息隐蔽：在用类和对象表示事物时，只将行为和属性公开给可信的外部事物。相应地，隐藏自身的内部特征信息。

15.继承

复用

黑盒复用：类 `B` 以关联(普通、聚合、组合)方式复用类 `A` 的功能.
白盒复用：继承。

继承

```
class C:public A,private B{};
```

`A` `B` 称基类， `C` 称派生类。
其中的 `public` 继承， `A` 称父类， `B` 称子类。
基类的构造、析构、拷贝、赋值、自动转换函数不会被自动继承。

派生类的构造函数

构造顺序：先基类，再派生类。
初始化列表中可以指定基类的构造函数或拷贝构造函数。
多重继承时，基类按先后顺序构造。

派生类的析构函数

先执行派生类的析构，再自动执行基类析构。

名词

redefine: 派生类中新定义的函数(基类中有同名、同原型的函数，非 `virtual`);
overload: 派生类中多个同名的重载函数;
overwrite: 派生类中定义了某个函数，且基类中有同名的函数，则派生类的函数会将基类的同名函数hide掉;
override: 若基类中的某个函数是 `virtual` 的，并在派生类中定义了同名的同原型函数，称**override**。

```
class A{
public:
        int f(){return 1;}
};
class B:public A{
public:
        int f(int n){return n;}
};
int main(){
        intot(B().f());
}
```

Error: too few arguments to function call, expected 1, have 0; did you mean `A::f` ?
`B` 的 `f(int)` 也能hide掉 `A` 的 `f()` ；可以在 `B` 中使用 `A::f()` 。

组合和继承对比

组合:
具有 ***has a*** 或 ***contain a*** 的关系;
子对象所属类的源代码，可有可无。；
类间是水平关系，相比继承可减少类的层次。
继承:
继承方式不同，目的不同;
基类的源代码必须有。
`public` ：表示 ***is a*** 的关系;

`private` ：表示 ***has a*** 或 ***contain a implement of*** 的关系，完全可换成组合；
protected：同 `private` ，同时便于在多层继承中保持这种关系。
选择：
优先选择组合，而不是继承；
通常用组合取代 `private` 继承；
在需要派生新的子类的情况下，才应采用公有继承。

16.继承和类型转换

`public` 继承的类型转换

```
class B:public A;
```

`A(B)` 子类变成父类，向上类型转换
`B(A)` 父类变成子类，向下类型转换

`static_cast`

`static_cast<T>(exp) = (T)exp`
强制类型转换。

`const_case`

`const_cast<T>(exp)`
用于添加或移除表达式中的 `const` 或 `volatile` 约束。

`reinterpret_cast`

`reinterpret_cast<T>(exp)`
对表达式的类型做出重新解释，常用于重新解释函数。

`dynamic_cast`

`dynamic_cast<T>(exp)`
动态转换：
在运行时刻，尝试将 `exp` 转换成 `T` 类型。多用于 `public` 继承下，将父类类型转换成子类类型；
`T` 只能为类的指针、类的引用、 `void *` 三种形式。
`dynamic_cast` 要用到RTTI(运行时类型识别)，通常各编译器都是通过虚拟表(VTable)来实现RTTI的，因此类中要有虚函数，才能使用。

`volatile`

```
volatile int a;
a=1;
a=2;
a=3;
a=4;
a=5;
```

该关键字提醒编译器不做任何优化，应用于多线程。
若不加该关键字，编译时上述代码中前4个对 `a` 的赋值操作都会被优化掉。

17.多重继承

名字冲突问题

```
class A{
public:
        int v='a';
        void f(){pt(v);}
};
class B{
public:
        int v='b';
        void f(){pt(v);}
};
class C:public A,public B{};
int main(){
        C c;
        c.f();
}
```

Error: member `f` found in multiple base classes of different types.
出现名字冲突的成员不会被直接继承，但是可通过名字空间调用。

```
class C:public A,public B{
public:
        using A::v;
};
int main(){
        C c;
        pt(c.v);
        c.A::f();
        c.B::f();
}
```

aab
也要注意访问控制。

菱形继承

```
class A;
class B:public A;
class C:public A;
class D:public B,public C;
```

此时，无法通过名字空间解决。
解决方案：
1.限定只能单继承。
2.限定多个基类中，至多只能有一个基类有实例变量：
避免了多重继承下数据成员名字冲突的问题；
保留了多重继承的方便性。
3.虚继承
4.接口继承

虚基类

```
class A;
class B:virtual public A;
class C:virtual public A;
class D:public B,public C;
```

虚继承的目的是让某个类做出声明，承诺愿意共享它的基类。
其中，这个被共享的基类就称为**虚基类**(Virtual Base Class)(如上 `A`)。
在这种机制下，不论**虚基类**在继承体系中出现了多少次，在派生类中都只包含一份**虚基类**的成员。

常见的无实例变量的类：

用类型做区分标志。
工具类：
类中只放置多个工具函数或类变量。
接口类：
指明其后裔类的公共行为集(也称行为接口)，通常接口类不能实例化，但其子孙类可实例化。
接口类无实例变量，一般只给出 `public` 行为，实例方法或类方法均可。

接口继承

基类中至多只有一个普通类，其它均为接口类。

18.虚机制

静态编联与动态编联

静态编联(早绑定，静态绑定):
编译期间就决定了程序运行时将具体调用哪个函数体。
即使没有主程序，也能知道程序中各个函数体之间的调用关系。
动态编联(晚绑定，动态绑定):
在运行期间，决定具体调用哪个函数体。
实现：多种方式，如虚机制(使用虚拟函数和虚拟函数表)。

虚函数

约束:
必须是成员函数;
不能是静态成员函数、普通构造函数、拷贝构造函数;
派生类中同名函数返回类型必须与基类中虚函数的返回类型相同或相容;
通常:
若类中有其它虚函数，那么析构函数也应该是虚的;
赋值函数通常不定义为虚的。
自由:
析构函数， `const` 修饰，访问控制。

```
class A{
public:
        virtual int name(){return 'a';}
        void f(){pt(name());pt(32);}
        virtual ~A(){pt('0');f();}
};
class B:public A{
public:
        int name(){return 'b';}
        ~B(){pt('1');f();}
};
class C:public B{
public:
        int name(){return 'c';}
        ~C(){pt('2');f();}
};
int main(){
printf("bc: ");
        B*bc=new C;
        bc->f();
        delete bc;
printf("\nac: ");
        A*ac=new C;
        ac->f();
        delete ac;
printf("\nab: ");
        A*ab=new B;
        ab->f();
        intot(typeid(ab)==typeid(A*));
        intot(typeid(ab)==typeid(B*));
        intot(typeid(*ab)==typeid(A));
        intot(typeid(*ab)==typeid(B));
        pt(32);
        delete ab;
printf("\nend\n");
}
```

bc: c 2c 1b 0a

ac: c 2c 1b 0a

ab: b 1001 1b 0a

end

可见，执行某个析构函数时，只能调用(该析构函数所在的)本地的 `f()` 。

上述 `B*bc` `A*ac` `A*ab` 的真实类型是 `C` `C` `B` ，但在运行时才能确定。

若去掉 `A` 中的两个 `virtual` ，则输出为：

bc: a 1a 0a

ac: a 0a

ab: a 1010 0a

end

override:

当 A 中某一成员函数为 virtual 时， B C 中该同名函数(或“同为析构函数”)无论是否加 virtual ，都是 virtual 的。(或说 virtual 关键字可以省略。)

若该函数不为析构函数，则在 B C 类的对象中被 B C 的同名函数覆盖，且当 B C 类的对象被 A 类指针/别名时该覆盖仍然生效。

若该函数为析构函数，则意味着要从原对象类开始，向上一层一层地调用析构函数。在这里， virtual 地作用可理解为“提示从哪个类开始向上析构”，而不具有“覆盖”能力。

redefine:

当 A 中某一成员函数不为 virtual 时， B C 中该同名函数会覆盖掉 A 的；若 B C 中该同名函数是 virtual 的，反而会被 A 的覆盖掉。

即同名函数“覆盖”的优先级：

非 virtual 一定能覆盖 virtual 的；其次，新的覆盖旧的。

虚函数表(虚拟表，虚表，VTable)

一个指针数组，各元素存放对应虚函数的入口地址。

不是写在代码里，是运行时在内存中自动创建的。

说明：

对应的类中至少有一个虚函数；

一个类至多有一个虚表，同一个类的不同对象共享该虚表；

首次创建该类实例对象时，在内存中同时创建该类的虚表；

按照函数顺序的序号依次存放入口地址。

19.多态性及其应用

多态性

相同的消息请求，执行不同的代码体，从而有不同的行为后果。

静态多态：

根据目标对象的静态类型和参数表中参数的静态类型确定目标代码体。

如模板(不同的模板参数)，函数重载。

动态多态：

如有多个函数有多种实现，则可以每个功能开一个工具类。

每个工具类只存放一个虚函数，该函数的每种实现是该工具类的一个派生类。

原类中存储每个工具类的指针，构造时通常需要指明，调用时通过指针调用。

```
class Func1{public:virtual bool _()=0;};
class Func1A:public Func1{public:bool _(){return 1;}};
class Func1B:public Func1{public:bool _(){return 0;}};
class Func2{public:virtual bool _()=0;};
class Func2A:public Func2{public:bool _(){return 1;}};
class Func2B:public Func2{public:bool _(){return 0;}};
class Z{
public:
        Z(Func1*p1,Func2*p2):pFunc1(p1),pFunc2(p2){}
        int ifChoose(){return pFunc1->_()<<10|pFunc2->_();}
private:
        Func1 *pFunc1;
        Func2 *pFunc2;
};
int main(){
        Z a(new Func1A,new Func2B);
        intot(a.ifChoose());pt(32);
        Z b(new Func1B,new Func2A);
        intot(b.ifChoose());pt(32);
}
```

1024 1

20.面向对象程序设计

面向对象程序设计

建立模型:
用依赖、关联、组合、聚合关系建立较高层次的关系模型;
根据问题域、领域知识、经验等抽象出类型;
只使用水平关系;
主要考察类的公有行为。
细化模型:
用依赖、关联、组合、聚合关系降低模型的抽象层次;
更体现领域知识、经验等的作用;
只使用水平关系;
主要考察类的行为;
类型的抽象与表示:
用类表示抽象出的类型;
涉及类的拆分、类的合并、行为的表示、数据的表示和组织等。
封装变化:
将可能变化的部分，用类单独封装;
涉及类的行为接口变化、行为实现的变化、数据表示变化、数据组织的变化等;
需要领域知识、经验等;

使用水平关系；
用子类型化适应变化：
针对一个维度的变化，用子类型化适应未来变化；
对于多个维度的变化，用水平关系将各维度的变化独立出来。

关系

水平关系：
依赖；
聚集(整体/部分)；
组合；
聚合；
关联(一般/单向/双向)。
垂直关系：
公有继承；
私有继承(用组合代替)；
保护继承。

21.异常处理

`throw()`

不抛出任何异常。

```
class A{
public:
        void f(int x);
        void g(int x);
        void h(int x)throw();
};
void A::f(int x){
        try{
                g(x);
                h(x);
        }catch(char const*s){
                printf("s:%s",s);
        }catch(int x){
                printf("i:%d",x);
        }
        pt(10);pt('_');pt(10);
}
void A::g(int x){
        if(x==33)throw 233;
        if(x==22)throw "qwq";
}
void A::h(int x)throw(){
        if(x==233)throw 'a';
}
int main(){
        A a;
        a.f(22);
        a.f(33);
        a.f(233);
}
```

s:qwq

_
i:233

_
terminate called after throwing an instance of 'int'
(出现这行之后停止运行(RE)。)
可见 `throw()` 意味着不抛出(可被 `catch()` 捕获的)异常，中间若出现异常则仍RE。

异常中立

当出现异常时，不对异常进行处理，而是完整地交由调用者处理。
(可以输出特定字符串等数据，仍视为符合异常中立。)

22.模板

模板

```
template<typename T>class A{
public:
        T v[60];
        A(){for(int i=0;i<60;i++)v[i]=i+200;}
        static void type();
};
template<typename T>static void A<T>::type(){
        printf("%s ",typeid(T).name());
}
template<typename T,int _N>class B:public A<T>{
public:
        int ot();
private:
        int vv[_N];
};
template<typename T,int _N>int B<T,_N>::ot(){
        intot(A<T>::v[_N]);pt(32);
        intot(this->v[_N]);pt(32);
        return _N;
}
int main(){
        B<int,33>b;
        b.ot();
        A<int>::type();
        B<double******,1>::type();
}
```

233 233 i PPPPPPd
因为有偏特化，所以一个模板派生类其实是不能在实例化之前就知道他的模板基类到底是谁，因此名字也无法resolve，所以只能以 `this->` 或 `A<T>::` 方式调用 `A` 的成员。

23.Lambda表达式

Lambda表达式

Lambda表达式代表的是一个匿名(无名)的函数。
Lambda表达式语法格式(4种)：

```
[capture](params)mutable exception attribute->retType{body}
[capture](params)->retType{body}
[capture](params){body}
[capture]{body}
```

其中
(1)是完整的lambda表达式形式;
(3)是省略了返回值类型的lambda表达式，函数返回值类型由 body 自动推演出来。
(4)省略了参数列表，类似于无参函数 f() 。
mutable 修饰符表明 body 内的代码可以修改被捕获(capture)的变量，并且可以访问被捕获对象的non-const方法。
exception 说明lambda表达式是否抛出异常(noexcept)，以及抛出何种异常，类似于 void f() throw(X,Y) 。
attribute 用来声明属性。
retType 指明返回类型，可以明确地指定，如 int ， bool ，自定义类型等，也可以是 decltype 推演类型。
capture 指定了 body 中可以访问的外部变量列表，具体解释如下：
[a,&b] 以传值的方式捕获 a 变量，以引用的方式捕获 b 变量；
[this] 以传值的方式捕获 this 指针；
[&] 以引用的方式捕获所有的外部自动变量，可修改外部量；
[=] 以传值的方式捕获所有的外部自动变量，不可修改外部量；
[] 不捕获外部的任何变量。

```
void show(int x,int y,int z){/*输出xyz*/}
int a=10,b=20,c=40;


auto func1=[](int &n){return n+5;};
show(a,b,c);
show(a,b,func1(c));
show(a,b,c);
```

10 20 40
10 20 45
10 20 40

```
auto func2=[](int &n){return n+=5+a;};
```

Error: 不能访问 a

```
auto func2=[](int &n){return n+=5;};
show(a,b,c);
show(a,b,func2(c));
show(a,b,c);
```

10 20 40
10 20 45
10 20 45

```
auto func3=[=](int &n){a=b;return n+=a+b;};
```

Error: 不能修改 a

```
auto func3=[=](int &n){return n+=a+b;};
show(a,b,c);
show(a,b,func3(c));
show(a,b,c);
```

10 20 45
10 20 75
10 20 75

```
auto func4=[a](int &n){return n+=a+b;};
```

Error: 不能访问 b

```
auto func4=[a](int &n){a=11;return n+=a;};
```

Error: 不能修改 a

```
auto func4=[a](int &n){return n+=a;};
show(a,b,c);
show(a,b,func4(c));
show(a,b,c);
```

10 20 75
10 20 85
10 20 85

```
auto func5=[&b,a](int &n){++b;return n+=a+b;};
show(a,b,c);
show(a,b,func5(c));
show(a,b,c);
```

10 20 85
10 21 116

10 21 116

```
auto func6=[&b,a](int n){++b;return n+=a+b;};
show(a,b,c);
show(a,b,func6(c));
show(a,b,c);
```

10 21 116
10 22 148
10 22 116