

2.C到C++升级

- 能够获取 `register` 类型变量的地址（只是为了兼容）
- 可以在任意的地方声明变量
- 不允许定义多个同名的全局变量

C语言中多个同名的全局变量最终会被链接到全局数据区的同一个地址空间上

Struct关键字加强：

C语言中的 `struct` 定义一组变量的集合，其中定义标识符并不是一种新的类型

C++中的 `struct` 用于定义一个全新的类型

C++中所有的标识符都必须显示的指明类型

C语言中的 默认类型 在C++中是不合法的

```
f(i)
{
    printf("i = %d\n", i);
}
g()
{
    return 5;
}
```

```
int f()      int f(void)
```

在C语言中前者接受任意参数，返回int；后者不接受参数，返回int

C++中两者意义相同，都是不接受参数，返回int

3.进化后的const

C语言中的const：

修饰的变量是只读的，本质还是变量

修饰的局部变量在栈上分配空间

修饰的全局变量在只读存储区分配空间

const只在编译期有用，在运行期无用

const修饰的变量 不是真的常量，只是告诉编译器该变量不能出现在赋值符号左边

C语言中真正的常量只有enum

C++中的const：

修饰的是真正意义上的常量

遇见const声明时在符号表中放入常量

编译时若发现使用常量则**直接以符号表中的值替换**

编译时若

1. 对const常量使用了**extern**
2. 对const常量使用 **&** 操作符

则给对应的常量分配存储空间

C++编译器虽然可能为const常量分配空间，但不会使用其存储空间中的值（兼容C语言）

与**宏定义**的区别：

const由**编译器**处理，宏由**预处理器**处理

编译器会对const常量进行**类型检查**和**作用域检查**

4.布尔类型

bool 可取值只有 **true** 和 **false**

true 代表真值，用 1 表示

false代表非真值，用 0 表示

C++编译器会将**非0** 转换为 **true**，**0**转换为 **false**

5.引用及其本质

引用可以看作一个已定义**变量**的**别名**

语法：`Type& name = var;`

```
int a = 4;
int& b = a; // b为a的别名
b = 5;      // 操作b就是操作a
```

普通引用在**定义**时必须用同类型的变量进行初始化

毕竟引用是一个“已定义变量的别名”，定义时要表明该变量

意义：作为**变量别名**存在，可**代替指针**，但可读性更好

const 引用

`const Type& name = var;`

const引用让变量拥有只读属性

```
int a = 4;
const int& b = a;
int *p = (int*)&b;

b = 5; // Err
*p = 5; // Correct
```

当使用**常量**对const引用进行初始化，编译器会为常量值分配空间，并将引用名作为这段空间的别名

```
const int& b = 1;
int* p = (int*)&b;

b = 5; // Err
*p = 5; // Correct
```

使用常量对const引用初始化后将生成一个**只读变量**

引用的本质

引用在C++内部实现是一个**指针常量**

Type& name <-> Type* const name

引用所占用的空间大小与指针相同

Warning：不可返回普通局部变量的引用！

6.内联函数分析

内联函数 和 宏代码块

宏代码块由预处理器处理，简单的文本替换，缺陷大，为了解决这一问题，同时满足运行效率，定义了内联函数来取代宏代码块，**内联函数省去了函数调用时压栈，跳转和返回的开销。**

```
#define FUNC(a,b) (a)<(b)?(a):(b)

inline int func(int a, int b)
{
    return (a)<(b)?(a):(b);
}
```

FUNC(++a,b) -> (++a)<(b)?(++a):(b)：文本替换为导致与**预期不符的错误**

func(++a,b) ->：若内联编译成功，**编译器直接将函数体(函数的汇编代码)扩展到函数调用的地方**，保证了效率同时也避免出现错误。

inline只是一种请求，编译器**不一定允许**。

现代编译器提供扩展语法，能够对函数进行**强制内联**：

g++：**attribute((always_inline))**

C++中**inline**内联编译的限制：

- 不能存在任何形式的循环语句
- 不能存在过多的条件判断语句
- 函数体不能过于庞大
- 不能对函数进行取址操作
- 函数内联声明必须在调用语句之前

现代的编译器基本没有以上限制，只要不是函数体过于复杂

7. 函数参数的扩展

函数参数的默认值

C++可以在函数声明时为参数提供一个**默认值**，当函数调用没有提供参数的值，则使用默认值。

参数的默认值必须在**函数声明中指定**

```
int mul(int x = 0);

int main()
{
    printf("%d\n", mul()); //mul(0)
    return 0;
}

int mul(int x)
{
    return x * x;
}
```

函数默认参数的规则

- 在设计函数时，参数的默认值必须**从右向左提供**
- 函数调用时使用了默认值，则后续参数必须使用默认值

```
int add(int x, int y = 1, int z = 2);
int add(int x, int y, int z)
{
    return x * y * z;
}

add(0);           // x = 0, y = 1, z = 2
add(2, 3);        // x = 2, y = 3, z = 2
add(3, 2, 1);     // x = 3, y = 2, z = 1
```

函数占位参数

- 占位参数**只有参数类型的声明，没有参数名声明**
- 一般情况下，在函数体内部**无法使用占位参数**

```
int func(int x, int)
{
    return x;
}

func(1, 2); //OK
```

意义：

- 占位参数与默认参数结合起来使用

- 兼容C语言程序中可能出现的不规范写法

8、9.函数重载分析

重载 (overload) :

同一个标识符在**不同的上下文有不同的意义**。

“洗”和不要词汇搭配后有不同的含义：洗衣服，洗脸，洗脑等

“play”：play chess, play piano, etc.

函数重载 (function overload) :

用**同一个函数名定义不同的函数**，当函数名和**不同的参数搭配**时函数的含义不同

```
int func(int x)
{
    return x;
}

int func(int a, int b)
{
    return a + b;
}

int func(const char* s)
{
    return strlen(s);
}
```

函数重载**至少**满足下面一个条件：

- 参数个数不同
- 参数类型不同
- 参数顺序不同

```
int func(int a, const char* s)
{
    return a;
}

int func(const char* s, int a)
{
    return strlen(s);
}
```

当函数**默认参数**遇上**函数重载**

```
int func(int a, int b, int c = 0)
{
    return a * b * c;
}
```

```

}

int func(int a, int b)
{
    return a + b;
}

int main()
{
    int c = func(1, 2); // which to call?

    return 0;
}

```

编译器调用重载函数的准则：

- 将所有同名函数作为**候选者**
- 尝试寻找可行的候选函数
 1. 精确匹配实参
 2. 通过默认参数能够匹配实参
 3. 通过默认类型转换匹配实参
- 匹配失败
 1. 最终找到的候选函数不唯一，编译失败
 2. 无法匹配所有候选者，函数未定义，编译失败

WARNING：

- 重载函数**本质上**是相互独立的**不同**函数（编译器认为重载函数具有不同的标识符，相同的函数名拥有不同的内存地址）
- 重载函数的函数类型不同
- **函数返回值不能作为函数重载的依据**

函数重载是由**函数名**和**参数列表**决定的，通过参数列表区分不同的同名函数

当**函数重载**遇上**函数指针**

将重载函数名赋值给函数指针时

1. 根据**重载规则**挑选与**函数指针参数列表一致**的候选者
2. **严格匹配**（考虑函数返回值了）候选者的**函数类型**与函数指针的**函数类型**

WARNING：

- 函数重载必然发生在同一个作用域中
- 编译器需要用**参数列表**或**函数类型**进行函数选择
- 无法直接通过**函数名**得到重载函数的**入口地址**

C++和C相互调用

`extern` 关键字能**强制让C++编译器进行C方式的编译**

```
extern "C"
{
    // C-style
}
/*****
// c++ code    C调用C++函数
extern "C" int add(int, int);
int add(int a, int b)
{
    return a + b;
}
```

如何保证代码能被C和C++编译器编译通过

- `__cplusplus` 是C++编译器内置的标准宏定义

意义：

确保C代码以统一的C方式被编译成目标文件

```
#ifdef __cplusplus
extern "C"{
#endif

// C-style

#ifdef __cplusplus
}
#endif
```

WARNING：

- C++编译器**不能以C的方式编译重载函数**
- **编译方式决定函数名被编译后的目标名**
 1. C++编译方式将**函数名和参数列表**编译成**目标名**
 2. C编译方式只将**函数名**作为目标名进行编译

extern "C" 代码块里绝对不能有重载函数！！！！

10.C++中的新成员

动态内存分配

- C++中通过 `new` 关键字进行动态内存申请
- C++中的动态内存申请是基于类型进行的
- `delete` 关键字用于内存释放

```
/* 变量申请 */
Type* pointer = new Type;
//....
delete pointer;

/* 数组申请 */
Type* pointer = new Type[N];
//....
delete[] pointer;
```

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 在申请单个类型变量时可进行初始化
- malloc 不具备内存初始化的特性

new关键字的初始化

```
int* pi = new int(1);
float* pf = new float(2.0f);
char* pc = new char('c');
```

C++中的命名空间

- C语言中只有一个**全局作用域**
 1. C语言中所有的全局标识符共享同一个作用域
 2. 标识符之间可能发生冲突
- C++中提出了**命名空间**的概念
 1. 命名空间**将全局作用域分成不同的部分**
 2. **不同命名空间中的标识符可以同名而不会发生冲突**
 3. 命名空间可以相互嵌套
 4. 全局作用域也叫**默认命名空间**

命名空间的定义：


```
namespace Name
{
    namespace internal
    {
        /* ... */
    }

    /* ... */
}
```

命名空间的使用：

- 使用整个命名空间：`using namespace name`
- 使用命名空间中的变量：`using name::variable`
- 使用默认命名空间（全局空间）中的变量：`::variable`

11. 新型的类型转换

C语言的强制类型转换过于粗暴，潜在问题不易被发现，出现问题很难从源代码中快速定位。

C++将强制类型转换分为4种不同的类型

<code>static_cast</code>	<code>const_cast</code>
<code>dynamic_cast</code>	<code>reinterpret_cast</code>

用法： `xxx_cast< Type >(Expression)`

static_cast：

- 用于**基本类型**间的转换
- **不能**用于**基本类型指针**间的转换
- 用于有**继承关系类对象**之间的转换和**类指针**之间的转换

const_cast：

- 用于**去除**变量的只读属性
- 强制转换的**目标类型**必须是**指针**或**引用**

reinterpret_cast：

- 用于**指针类型**间的强制转换
- 用于**整数**和**指针类型**间的强制转换

dynamic_cast：

- 用于有**继承关系**的**类指针**间的转换
- 用于有**交叉关系**的**类指针**间的转换
- 具有**类型检查**的功能
- 需要**虚函数**的支持

12.经典问题（const，引用）

关于const的疑问

什么时候为**只读变量**？什么时候为**常量**？

- 只有用**字面量**初始化的const常量才会**进入符号表**
- 使用其他变量初始化的const常量仍然是**只读变量**
- 被**volatile**修饰的const常量**不会进入符号表**

volatile：代表声明的标识符是**易变的**，只不过改变的可能性不在当前编译的文件中，可能发生在外部（其他文件中的多线程，中断等），每次访问被volatile修饰的标识符时应该去**内存中**直接读取，故被volatile修饰的标识符不可能进入符号表。

在编译期间不能直接确定初始值的const标识符，都被作为只读变量处理。

const 引用的类型与初始化变量的类型

- 相同：初始化变量成为只读变量
- 不同：生成一个新的只读变量

```
char c = 'c';
char& rc = c;
const int& rrc = c; // 生成新的只读变量

rc = 'a';
printf("c = %c\n", c);      // a
printf("rc = %c\n", rc);    // a
printf("rrc = %c\n", rrc);  // c
```

关于引用的疑问

引用与指针有什么关系？

指针是一个变量

- 值为一个内存地址，不需要初始化，可以保存不同的地址
- 通过指针可以访问对应内存地址中的值
- 指针可以被const修饰为常量或只读变量

引用只是一个变量的新名字

- 对引用的操作都会传递到代表的变量上
- const引用使其代表的变量具有只读属性
- 引用**必须在定义时初始化**，之后无法代表其他变量

C++中不支持**引用数组**！！！！

```
int& array[] = {a, b, c};
```

为了兼容C语言，数组元素在内存里会相邻排列

但是当a,b,c变量所储存的区域各不相同，就会与上面的准则冲突

故C++不支持引用数组！

13、14.进阶面向对象

分类的思想

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

类：指的是一类事物，是一个抽象概念，是一种模型，可以创建出不同的对象实体

对象：指的是属于某个类的具体实体，对象实体是类模型的一个具体实例

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

类之间的基本关系

- 继承
 1. 从**已存在类细分出来的类**和原类之间具有继承关系（is-a）
 2. 继承的类（子类）**拥有**原类（父类）的**所有属性和行为**
- 组合
 1. 一些类的存在**必须依赖于**其他的类，这种关系叫组合
 2. 组合的类在**某一个局部上由其他的类组成**

```
struct Biology
{
    bool living;
};

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

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

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

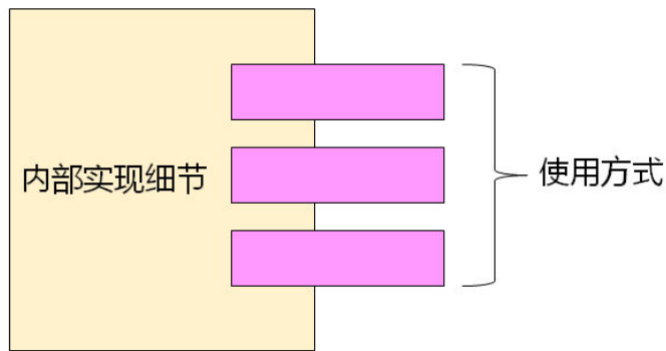
struct Plant:Biology
{
    bool growable;
};
```

15.类和封装的概念

类的封装

类通常分为两部分：

- 类的实现细节
- 类的使用方式



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
- 而有一些类的属性是对外公开的
- 必须在类的表示法中定义属性和行为的公开级别

C++中的类封装

- 成员变量：用于表示**类属性**的变量
- 成员函数：用于表示**类行为**的函数
- 可以给**成员变量**和**成员函数**定义访问级别
 1. `public`：成员变量和成员函数可以在**类的内部**和**外界**进行访问和调用
 2. `private`：成员变量和成员函数**只能在类的内部**被访问和调用

2：15-1.cpp

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
- 类成员的作用域和访问级别没有关系

类的作用域，仅仅是指类的成员变量和成员函数的作用域（在类的范围之内），如果在类之外想访问成员变量和成员函数必须建立一个类对象，通过类对象来访问，但是否访问成功需要看这些成员的访问级别。

3-3：20，15-2.cpp

16.类的真正形态

新的关键字 `class` 用于类定义

`class` 和 `struct` 的用法是完全相同的

区别：

- `class`定义类时，所有成员的**默认访问级别**是 `private`
- `struct`定义类时，所有成员的**默认访问级别**是 `public`

1-6：20，16-1.cpp

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 `setPatameter` 函数设置运算参数
 - 提供 `result` 函数进行运算
 1. 其返回值表示运算的合法性
 2. 通过引用参数返回结果
-

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**完成类的其他实现**
成员函数的具体实现

17、18、19.对象的构造

对象的初始化

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

初始状态（出厂设置）是对象普遍存在的一个状态

构造函数

C++中可以定义与**类名**相同的特殊**成员函数**：

构造函数

- **没有任何**返回类型声明
- 在对象定义时**自动被调用**

类的构造函数用于对象的初始化

带有参数的构造函数

- 构造函数可**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++重载规则

```
class Tst
{
    public:
        Tst(int v)
        {
            // using v to initialize member
        }
};
```

WARNING :

对象定义和对象声明不同

- 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

构造函数的调用

1. 自动调用

```
class Test
{
    public:
        Test(){}
        Test(int v){}
};

int main()
{
    Test t;      // 调用 Test()
    Test t1(1);  // 调用 Test(int v)
    Test t2 = 2; // 调用 Test(int v)

    return 0;
}
```

2. 手动调用

一般情况下，构造函数在对象定义时被自动调用，有时也需要手工调用

```
class Test
{
    public:
        Test(){}
        Test(int v){}
};

int main()
{
    /* 创建对象数组 */
    Test ta[3] = {Test(), Test(1), Test(2)}; // ta[0] = Test() ta[1] = Test(1)

    return 0;
}
```

实例

需求：开发一个数组类解决原生数组安全性问题（数组越界）

- 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

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

拷贝构造函数

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

浅拷贝：拷贝后对象的物理状态相同（对象所占字节相同）

深拷贝：拷贝后对象的逻辑状态相同（所占内存上的数据相同）

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、打开了外存中的文件、使用了系统中的网络端口等）时，需要进行深拷贝

编译器提供的拷贝函数只进行浅拷贝，仅仅只是变量的赋值，当变量指向动态内存空间等系统资源，会导致两个指针指向该内存空间，释放时就会导致两次释放从而产生错误。

自定义拷贝构造函数，必须实现深拷贝！！！！

20.初始化列表的使用

类中定义 `const` 成员

```
class Test
{
    private:
        const int ci;
    public:
        int getCI()
        {
            return ci;
        }
};
```

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：

```
ClassName::ClassName() : m1(v1), m2(v1,v2), m3(v3)
{
    // other initialize operation
}
```

WARNING：

- 成员的**初始化顺序**和成员的**声明顺序相同**，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

类中的const成员

- 类中的const成员会被分配空间（取决于定义的对象所在空间）
- 本质是**只读变量**
- **只能在初始化列表中**指定初始值

初始化与赋值不同

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

21.对象的构造顺序

局部对象

- 当程序执行流**到达对象的定义语句**时进行构造

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**

```
int i = 0;
Test* a1 = new Test(i);

while(++i < 10)
    if (i % 2)
        new Test(i);

if (i < 4)
    new Test(*a1);
else
    new Test(100);
```


全局对象

- 对象的构造顺序是**不确定的**
- 不同的编译器**使用不同的规则**确定构造顺序

在开发过程中，尽量避免使用全局变量，提高内聚性

22.对象的销毁

析构函数

- C++的类中可以定义一个特殊的清理函数——>**析构函数，功能与构造函数相反**
- 定义：`~ClassName()`
 1. 析构函数**没有参数也没有返回值类型**的声明
 2. 在对象销毁时**自动被调用**

当类中自定义了构造函数，并且构造函数中**使用了系统资源**（内存申请，文件打开.etc），则需要自定义析构函数

23.神秘的临时对象

Q：直接调用构造函数的行为会发生什么？

A：

- **直接调用构造函数会产生一个临时对象**
- 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++中的**灰色地带**

```
Test a = Test(1); // 过程分析：先产生一个临时对象，再通过拷贝构造函数将临时对象赋值给a
                  // 编译器优化：Test a = 1;
```

24.经典问题解析（析构，const对象，类成员）

析构

Q：当程序中存在多个对象时，如何确定这些对象的析构顺序？

A：

单个对象创建时构造函数的调用顺序：

1. 调用父类的构造过程
2. 调用**成员变量的构造函数**（调用顺序和声明顺序相同）
3. 调用**类自身的构造函数**

析构函数和对应构造函数的调用顺序相反

const

- const关键字能够修饰对象
- 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（成员变量为只读变量）
- 只读对象是编译阶段的概念，运行时无效

const成员函数：

- const对象只能调用const成员函数
- const成员函数中只能调用const成员函数
- const成员函数中不能直接改写成员变量的值

const成员函数定义：

```
Type ClassName::function(Type p) const
```

类中的函数声明与实际函数定义中都必须带const关键字

类成员

- 所有的对象共享类的成员函数
- 成员函数能够直接访问所有对象的成员变量
- 成员变量中的隐藏参数 this 用于指代当前对象
- 每一个对象拥有自己独立的成员变量

25.类的静态成员变量

静态成员变量

- 静态成员变量属于整个类所有
- 可以通过类名直接访问公有的静态成员变量
- 可以通过对象名访问公有的静态成员变量
- 所有对象共享类的静态成员变量

特性：

- 定义时直接用 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;
```

26.类的静态成员函数

静态成员函数：

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

定义：

```

class Test
{
    public:
        static void func1(){};
        static int func2();
};

int Test::func2()
{
    return 0;
}

```

	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

27.二阶构造模式

1. 如何判断构造函数的执行结果？
2. 在构造函数中**执行return语句**会发生什么？
3. 构造函数**执行结束**是否意味着**对象构造成功**？

You must know

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后构造函数**立即结束**
- 将构造函数**私有**，就**无法通过类名**创建对象

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！！！！

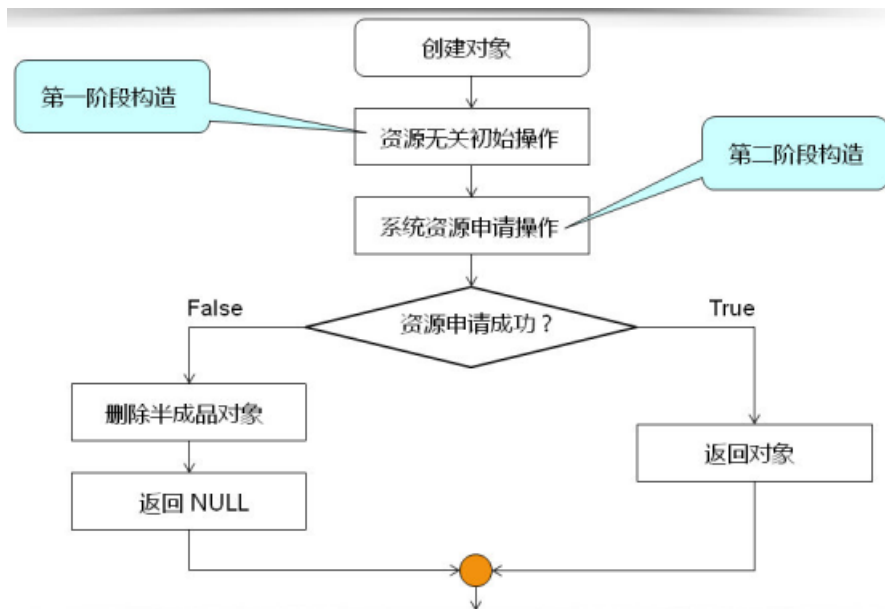
半成品对象

- 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对象，**也是BUG的重要来源**

二阶构造

构造过程分两步：

- 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：

```
class TwoPhaseCons
{
private:
    TwoPhaseCons() // 第一阶段构造函数，构造函数私有后，无法通过类名创建对象
    {
    }

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

```
TwoPhaseCons* TwoPhaseCons::NewInstance()
{
    TwoPhaseCons* ret = new TwoPhaseCons();

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

    return ret;
}
```

28.友元的尴尬能力

友元

- 友元是一种关系
- 是描述**函数与类**或**类与类**之间的关系
- 友元关系是单项的，不能传递

用法：

- 在类中以 `friend` 关键字声明友元
- 类的友元可以是**其他类**或**具体函数**
- 友元**不是**类的一部分
- 友元**可以直接访问**具体类的所有成员

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

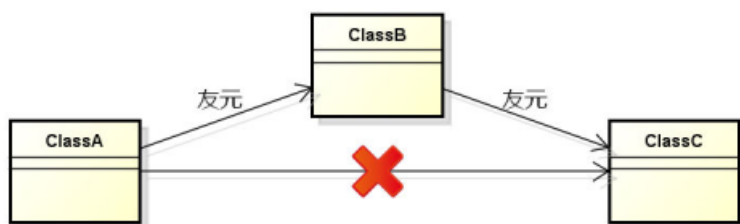
    friend void func(Point& p);
    friend class ClassOther;
};
void func(Point& p){}
class ClassOther{};
```

尴尬之处

- 友元是为了兼顾C语言的高效诞生的
- 破坏了面向对象的封装性
- 在实际产品中的高效是得不偿失的
- 已经逐渐被遗弃

WARNING：

- 友元关系**不具备传递性**
- 类的友元可以是**其他类的成员函数**
- 类的友元可以是**某个完整的类**
 - 所有的成员函数都是友元



对类的友元传入该类的对象，通过对象可访问该类的任意成员

29.类中的函数重载

函数重载必然发生在同一个作用域中

类中的**成员函数**之间可以进行重载，但是**全局函数**无法和**普通成员函数**以及**静态成员函数**之间构成重载。

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

30.操作符重载的概念

- 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：

```
/* Sign为系统中预定义的操作符，+，-，*，/，etc. */
Type operator Sign(const Type& p1, const Type& p2)
{
    Type ret;

    return ret;
}
```

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& p2)
    {
        Type ret;

        return ret;
    }
};
```

操作符重载的本质是通过函数扩展操作符的功能，本质为函数定义

31.完善的复数类

复数类应该具有的操作

- 运算：+ , - , * , /
- 比较：== , !=
- 赋值：=
- 求模：modules

利用操作符重载

```
Complex operator + (const Complex& c);
Complex operator - (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);
```

WARNING :

- 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

32.初探C++标准库

Fascinating Overloading

1<<2 , 重载左移操作符 , 将变量或常量左移到一个对象中

```
static const char endl = '\n';

class Console
{
public:
    Console& operator << (int a)
    {
        printf("%d", a);
        return *this;
    }
    Console& operator << (char c)
    {
        printf("%c", c);
        return *this;
    }
};

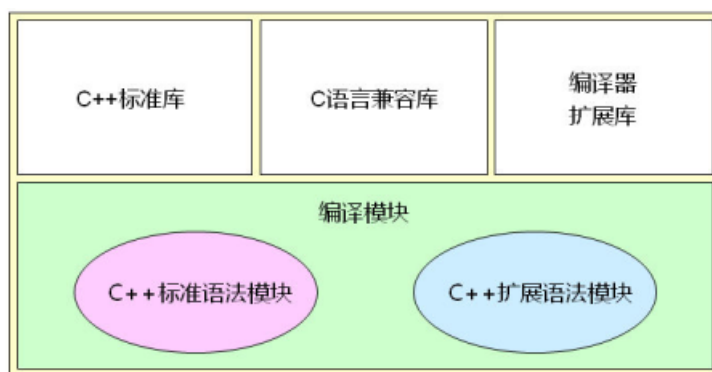
int main()
{
    Console cout;
```

```
cout << 1 << endl;  
  
return 0;  
}
```

C++标准库

- C++标准库**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++编译环境组成



33.C++中的字符串类

C语言用字符数组和一组函数实现字符串操作

C++中可以通过类完成字符串类型的定义

标准库中的字符串类

`string` 类型

- 字符串连接
- 字符串大小比较
- 子串查找和提取 (`substr`, `find`)
- 字符串的插入和替换 (`insert` , `replace`)

字符串与数字的转换

字符串流类 (`sstream`) 用于`string`的转换

- 头文件
- `istringstream` 字符串输入流
- `ostringstream` 字符串输出流

`string` TO 数字


```
istringstream iss("123.45");
double n;
iss >> n;
```

数字 TO string

```
ostringstream oss;
oss << 543.21;
string s = oss.str();
```

34.数组操作符的重载

字符串类的兼容性

可以按照使用C字符串的方式使用 `string` 对象

```
string s = "a1b2c3d4e";
int n = 0;

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

重载数组访问操作符

```
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**能且仅能**使用一个参数
- 可以定义不同参数的**多个重载函数**

数组访问操作符的重载能够使得对象模拟数组的行为

35.函数对象分析

编写一个函数：

- 函数可以获得斐波那契数列每项的值
- 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;
}
```

解决方案：

- 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

函数调用操作符 (`()`)

- 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

函数对象用于在工程中取代函数指针

36.经典问题解析

赋值

Q：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？

A：

- 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
- 赋值操作符与拷贝构造函数具有相同的存在意义

编译器默认提供的函数

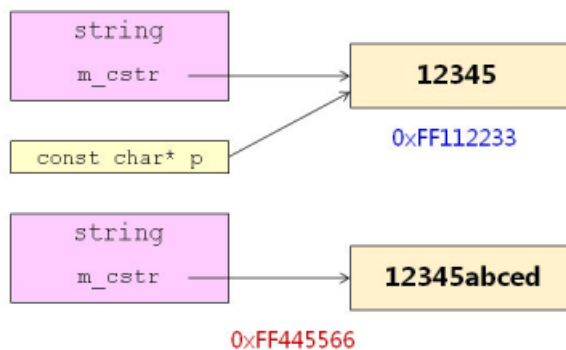
<pre>class Test { Test&); };</pre>	<p>等价</p> <p><=====></p>	<pre>class Test { public: Test(); Test(const Test&); Test& operator = (const ~Test(); };</pre>
--	--------------------------------	--

string

```
string s = "12345";
const char* p = s.c_str();

cout << p << endl; // 12345

s.append("abcde");
cout << p << endl; // 12345
```



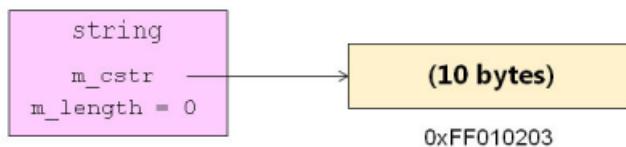
KEY：string 类通过一个数据空间保存字符数据，在 `append` 后 `m_cstr` 指向了一块新的内存区域，同时释放原来指向的内存，此时 `p` 指针为野指针。

```
const char* p = "12345";
string s = "";

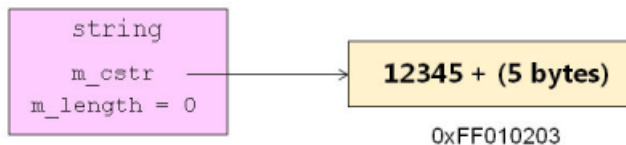
s.reserve(10);
for (int i=0; i<5; i++)
{
    s[i] = p[i];
}

if (!s.empty())
{
    cout << s << endl;
}
```

for 循环执行前：



for 循环执行后：



KEY：string 类通过一个成员变量保存当前字符串长度，`s.empty()` 的判断标准是成员变量 `m_length` 的值，故无法满足条件。

37.智能指针分析

内存泄露

- 动态申请堆空间，用完后不归还
- C++中没有垃圾回收机制
- 指针无法控制所指堆空间的生命周期

What we need

- 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

智能指针

- 重载指针特征操作符 (`->` 和 `*`)
- **只能通过成员函数重载**
- 重载函数**不能使用参数**
- 只能定义一个重载函数

WARNING : 智能指针只能用来指向堆空间中的对象或者变量

38.逻辑操作符的陷阱

- 重载操作符的是利用函数调用来扩展操作符的功能
- 本质还是函数调用，**在进入函数体前必须完成所有参数的计算**
- 在重载的逻辑操作符函数被调用前，如果函数参数不是确定的，需要进行计算或调用其他的函数
- 由于函数参数的计算顺序不确定，此时逻辑操作符的短路规则就不起作用

故不推荐重载 `&&` 和 `||`，无法满足短路规则，无法完全实现其原生语义。

可通过**重载比较操作符**代替逻辑操作符重载，`== true or false`

也可**使用专用成员函数**代替逻辑操作符重载

39.逗号操作符的分析

复习：

逗号操作符 (`,`) 可以构成逗号表达式

- 逗号表达式用于**将多个子表达式连接为一个表达式**
- 逗号表达式的值为**最后一个子表达式的值**
- 逗号表达式中的**前N-1个子表达式可以没有返回值**
- 逗号表达式按照**从左向右的顺序**计算每个子表达式的值

使用**全局函数**对逗号操作符进行重载

重载函数的**参数**必须有一个是**类类型**

重载函数的**返回值**类型必须是**引用**

```
Class& operator , (const Class& a, const Class& b)
{
    return const_cast<Class&>(b);
}
```

- 重载操作符的是利用函数调用来扩展操作符的功能
- 本质还是函数调用，**在进入函数体前必须完成所有参数的计算**
- 在重载的逻辑操作符函数被调用前，如果函数参数不是确定的，需要进行计算或调用其他的函数
- 由于函数参数的计算顺序不确定，重载后无法严格从左向右计算表达式

不要重载逗号表达式！！！！

40.前置操作符和后置操作符

```
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
```

通过不同的编译器编译查看汇编，发现两者的汇编代码是一样的

- 现代编译器会优化代码，使得最终的二进制程序更高效
- 不可能从编译后的二进制程序还原C/C++程序

++操作符重载

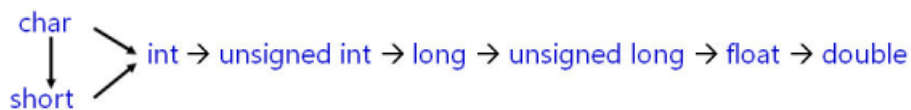
- 全局函数和成员函数均可进行重载
- 重载前置++操作符不需要额外的参数
- 重载后置++操作符需要一个int类型的占位参数

所以，i++ 和 ++i 的真正区别？

- 对于基础类型变量
 - 前置++ 的效率和后置++ 的效率基本相同
 - 根据项目组编码规范进行选择
- 对于类类型的对象
 - 前置++ 的效率高于后置++（后置++需要创建一个临时对象，最后还要销毁）
 - 尽量使用前置++操作符提高程序的效率

类型转换函数

隐式类型转换规则：



普通类型到类类型之间的类型转换

- 构造函数可以定义不同类型的参数
- 当参数仅有一个，类型是基本类型或其他类类型时，构造函数称为转换构造函数

```
Test(int i)
{
    mValue = i;
}
```

```
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);"

隐式类型转换是工程中BUG的重要来源！！

使用 explicit 关键字杜绝编译器的转换尝试，被 explicit 修饰的转换构造函数只能进行显示转换

转换方式：

- static_cast<ClassName>(value)
- ClassName(value)

类类型到普通类型之间的类型转换

类型转换函数

- 用于将类对象转换为其他类型
- 与转换构造函数具有同等的地位
- 编译器能够隐式的使用类型转换函数

规则：

```
operator Type()  
{  
    Type ret;  
    //...  
    return ret;  
}  
operator int ()  
{  
    return mValue;  
}
```

```
Test t{1};  
int i = t;
```



"t 这个对象为 Test 类型，怎么可能用于初始化 int 类型的变量呢！现在就报错吗？不急，我看看有没有类型转换函数！Ok，发现 Test 类中定义了 operator int ()，可以进行转换。"

类类型之间的类型转换

类型转换函数 VS 转换构造函数

```
// Test  
operator value ()  
{  
    value ret;
```

```

ret.wValue(mValue);

cout << "operator Value()" << endl;
return ret;
}
// Value
value(Test& t)
{

}

```

如果不对转换构造函数加上**explicit**关键字，会使两者发生冲突，编译器无法判断调用哪个函数

解决方法：

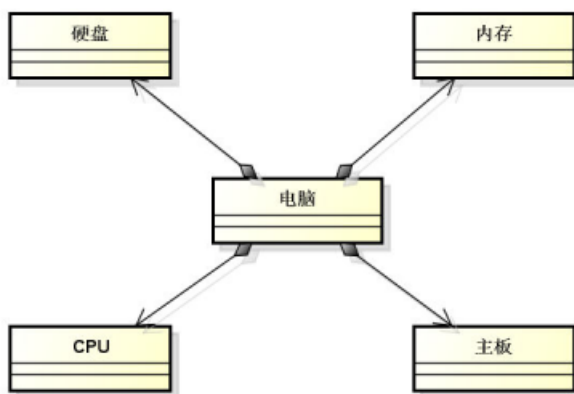
```

Type toType() // 使用公有成员函数代替类型转换函数

```

43.继承的概念和意义

组合关系：整体和部分的关系



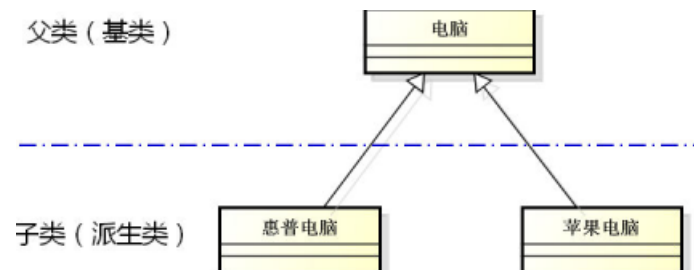
一个类由多个其他类组合而成。

组合关系特点：

- 将其他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

继承关系

父子关系：



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**
- 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

```
class Parent
{
    int mv;
public:
    void method() { };
};

class Child : public Parent // 描述继承关系
{
};
```

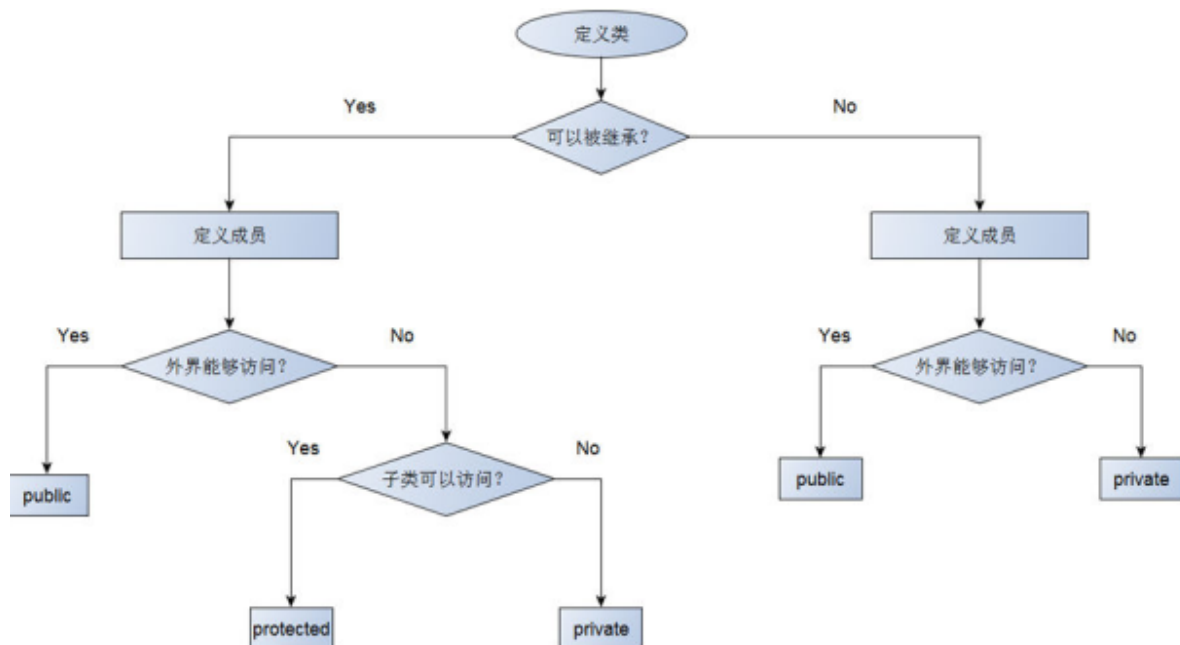
44.继承中的访问级别

可以定义 `protected` 访问级别

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**

定义类时访问级别的选择：



45.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**，其他成员保持不变

父类成员访问级别

继承方式		public	protected	private
	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private

C++中的默认继承方式为 `private`

46.继承中的构造与析构

子类对象的构造

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

```
class Child : public Parent
{
    public:
        Child() /* 隐式调用 */
        {
            cout << "Child()" << endl;
        }
        Child(string s) /* 显示调用 */
            : Parent("Parameter to Parent")
        {
            cout << "Child() : " << s << endl;
        }
};
```

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**显示调用**

对象创建时构造函数的调用顺序

- **调用父类的构造函数**
- 调用**成员变量**的构造函数
- 调用**类自身**的构造函数

口诀：先父母，后客人，再自己。

子类对象的析构

- 执行**自身**的析构函数
- 执行**成员变量**的析构函数
- 执行**父类**的析构函数

47.父子间的冲突

- 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

```
Child c;  
  
c.mi = 100;  
c.Parent::mi = 1000;
```

再论重载

类中的成员函数可以进行重载

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

父子冲突：

- 子类中的函数将**隐藏父类的同名函数**
- 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

48.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

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

相当于父类的指针（引用）指向了子类对象中父类的成员，可以访问这些成员，但无法访问子类自己新的成员

由于同名覆盖无法通过子类对象直接访问父类的同名函数，除了通过作用域分辨符来访问外，可通过指针或引用来访问被覆盖的同名成员

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**函数重写**
- 函数重写是同名覆盖的特殊情况

函数重写

```
class Parent
{
public:
    void print()
    {
        cout << "I'm Parent."
              << endl;
    }
};

class Child : public Parent
{
public:
    void print()
    {
        cout << "I'm Child."
              << endl;
    }
};
```



当函数重写遇上赋值兼容

- **编译期间**，编译器**只能根据指针的类型判断**所指向的对象
- 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 how_to_print(Parent* p)
{
    p->print();
}
```

在编译这个函数时，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就是调用父类的 print 函数。因为父类和子类都有相同的 print 函数。

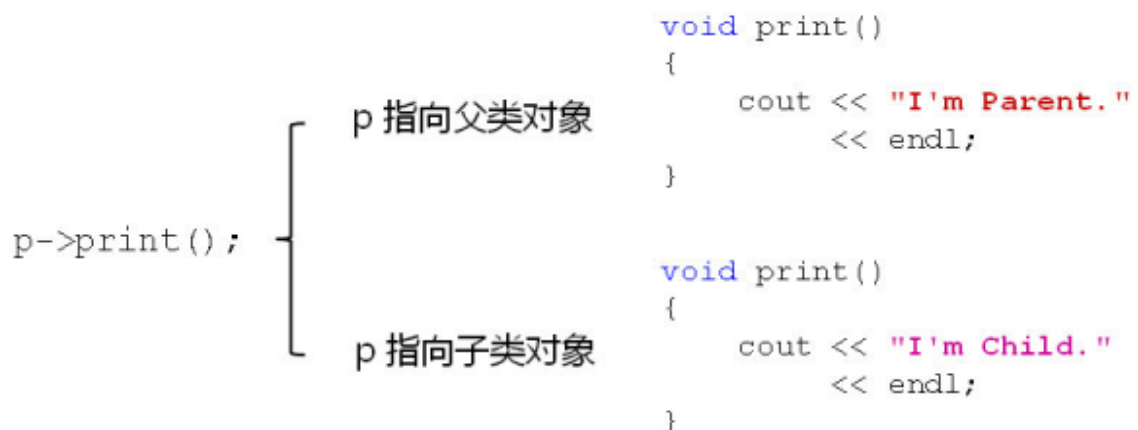
49.多态的概念和意义

面向对象中期望的行为

- 根据**实际对象类型判断**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



函数重写只可能发生在父类与子类之间

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，是支持多态的唯一方式
- 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

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（一直都是调用父类函数）
- 多态是面向对象组件化程序设计的基础特性

静态联编

- 在程序的编译期间就可以确定具体的函数调用，如：重载函数

动态联编

- 在程序实际运行后才能确定具体的函数调用，如：函数重写

50、51.C++对象模型分析

对象本质

`class` 是一种特殊的 `struct`

- `class` 和 `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

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

```
class A
{
    int i;
    int j;
    char c;
    double d;
};
struct B
{
    int i;
    int j;
    char c;
    double d;
};
A a;
B* p = reinterpret_cast<B*>(&a); // 通过强制类型转换改变A类中私有权限
p->i = 1;
p->j = 2;
p->c = 'c';
p->d = 3;
```

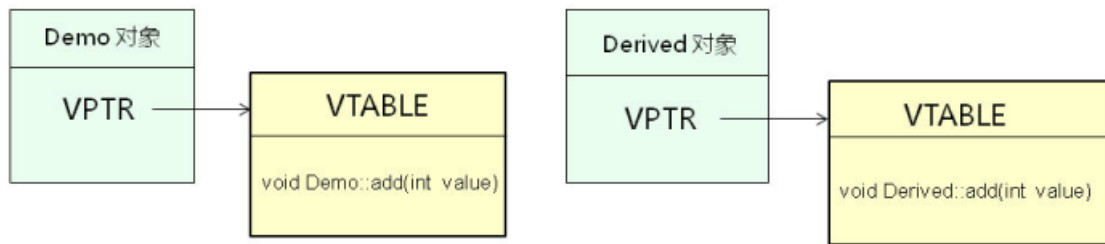
- 类中的**成员函数**位于**代码段**中
- 调用成员函数时**对象地址**作为**参数隐式传递**
- 成员函数**通过对象地址**访问**成员变量**
- **C++语法规则**隐藏了对象地址的传递过程

继承对象模型

C++多态实现原理

- 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**，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

虚函数的调用效率**低于**普通成员函数



```
void run(Demo* p, int v)
{
    p->add(v);
}
```



编译器确认 `add()` 是否为虚函数？

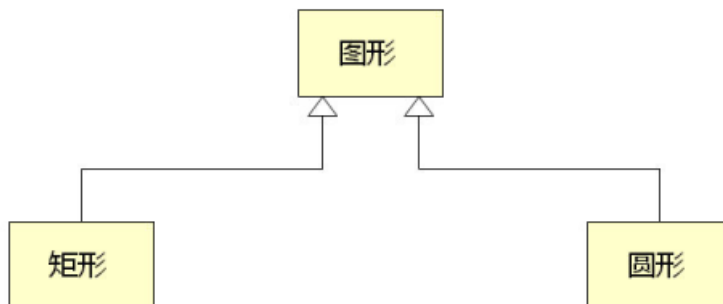
1. Yes → 编译器在对象 VPTR 所指向的虚函数表中查找 `add()` 的地址
2. No → 编译器直接可以确定被调成员函数的地址

52.C++中的抽象类和接口

抽象类

面向对象中的抽象概念

图形的面积如何计算？



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，没有具体对象
```

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
- 类里的**相关函数没有完整的实现**

抽象类与纯虚函数

- 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

```
class Shape
{
public:
    virtual double area() = 0; // 纯虚函数
};
// “= 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

class Circle : public Shape
{
    int mr;
public:
    Circle(int r)
    {
        mr = r;
    }
    double area() // 纯虚函数被实现后成为虚函数
    {
        return 3.14 * mr * mr;
    }
};
```

- 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**

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**且都是**纯虚函数**
- 接口是一种**特殊的抽象类**

53、54.被遗弃的多重继承

C++中的多重继承

- 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

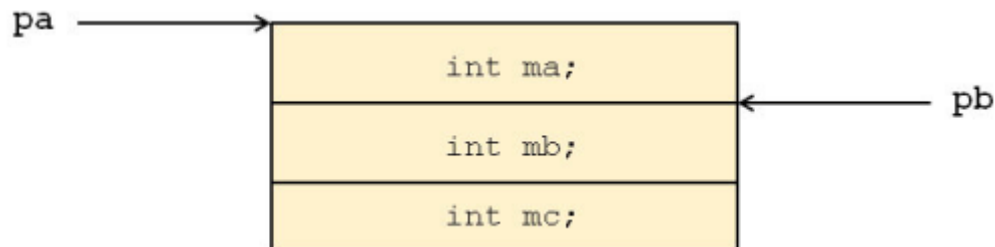
```
class Derived : public Base A, public Base B, public Base C
{
    //...
};
```

多重继承的问题一

- 通过多重继承得到的对象可能拥有“不同的地址”！

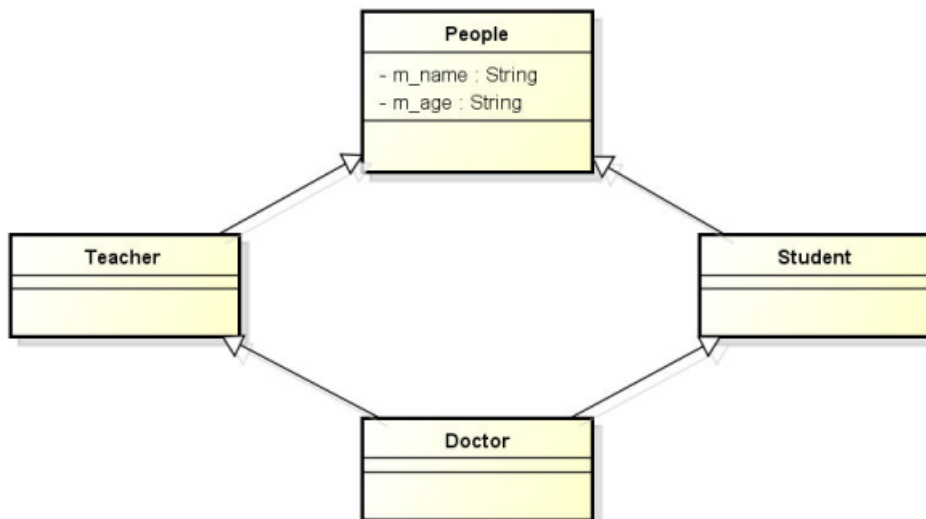
解决方案：无

```
Derived d(1, 2, 3);  
BaseA* pa = &d;  
BaseB* pb = &d;
```



多重继承的问题二

- 多重继承可能产生冗余的成员



当多重继承关系出现闭合时将产生数据冗余问题。

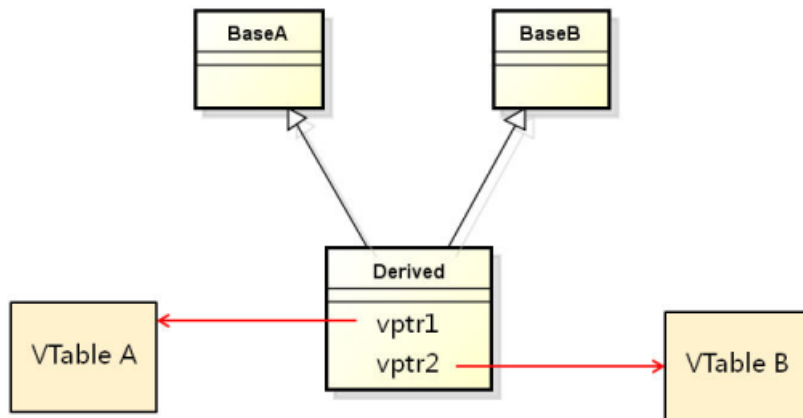
解决方案：虚继承

```
class People {};  
class Teacher : virtual public People{};  
class Student : virtual public People{};  
class Doctor : public Teacher, public Student  
{  
  
};
```

- 中间层父类不再关心顶层父类的初始化
- 最终的子类必须直接调用顶层父类的构造函数

多重继承的问题三

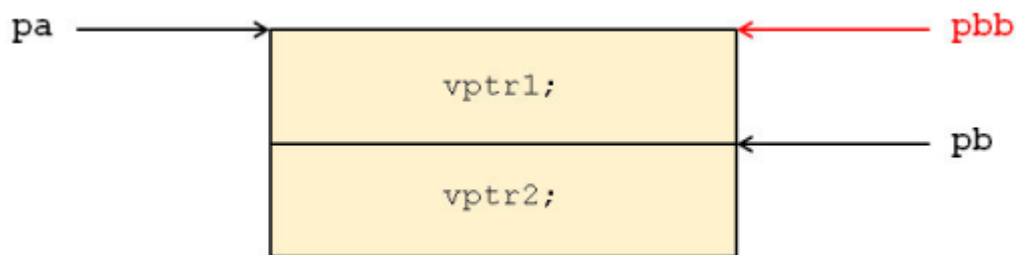
- 多重继承可能产生多个虚函数表



解决方案：`dynamic_cast`

强制类型转换时，使用新式类型转换关键字！

```
Derived d;  
BaseA* pa = &d;  
BaseB* pb = &d;  
BaseB* pbb = (BaseB*)(pa);
```



通过指针找到地址，通过地址找到**虚函数表指针**，进而到虚函数表指针指向的虚函数表得到函数地址

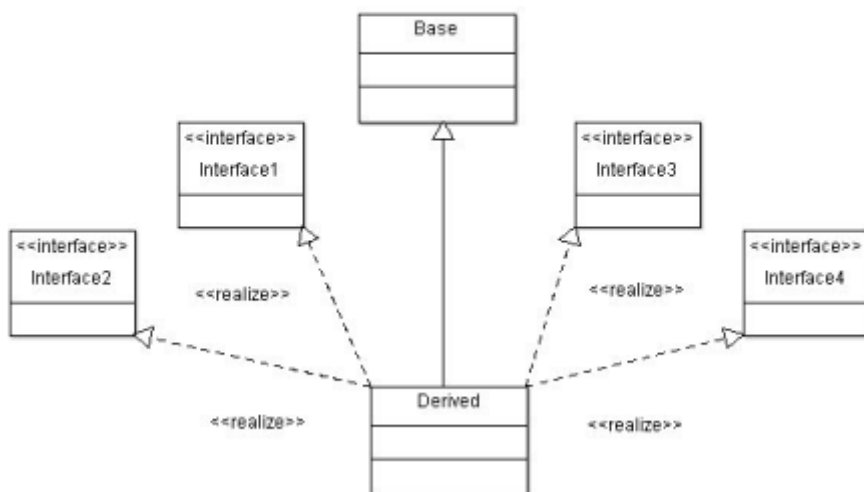
```
BaseB* pbb = dynamic_cast<BaseB*>(pa);
```

编译器首先检测 `pa` 指针指向的是 `d` 对象，进而会检查 `d` 对象有哪些父类，发现有 `BaseA` 和 `BaseB`，编译器会认为 `pa` 进行的强制类型转换是合法的，**在强制类型转换的时候会对指针修正**，使得 `pbb` 指向 `d` 对象里 `BaseB` 部分的地址。

正确的使用多重继承

- 工程开发中的“多重继承”方式：

单继承某个类 + 实现（多个）接口



- 先继承自一个父类，然后实现多个接口
- 父类提供 `equal()` 成员函数，用于判断指针是否指向当前对象

```
bool equal(Base* p)
{
    return (this == p);
}
```

- 与多重继承相关的强制类型转换用 `dynamic_cast` 完成

55.经典问题解析（动态内存分配、虚函数，继承中的强制类型转换）

动态内存分配

`new` 和 `malloc` 的区别

- `new` 关键字是C++的一部分，在所有C++编译器中都被支持；`malloc` 是由C库提供的函数，在某些系统开发中不能调用
- `new` 以具体类型的单位进行内存分配，能够触发构造函数的调用；`malloc` 仅仅以字节为单位进行内存分配
- 对象的创建只能使用 `new`，`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

`delete` 和 `free` 的区别

- `delete` 在所有C++编译器中都支持；`free` 在某些系统开发中不能调用
 - `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`
-

关于虚函数

—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初始化

—析构函数可以成为虚函数

- 建议设计类时将析构函数声明为虚函数

—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，只会调用当前类中定义的函数版本~

关于继承中的强制类型转换

- `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
- 用于有直接或间接继承关系的指针（引用）之间

— 指针

- 转换成功：得到目标类型的指针
- 转换失败：得到空指针

— 引用

- 转换成功：得到目标类型的指针
- 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，类型转换结果只能在运行阶段才能得到

56. 函数模板的概念和意义

交换变量的方法

- 定义宏代码块
 - 优点：代码复用，适合所有类型
 - 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

泛型编程

- 不考虑具体数据类型的编程方式

```
void Swap(T& a, T& b)
{
    T t = a;
    b = a;
    a = t;
}
// T泛指任意的数据类型
```

函数模板

- 一种特殊的函数可用不同类型进行调用
- 类型可被参数化

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

- `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**

告诉编译器开始泛型编程 ← `template` `<typename T>` → 告诉编译器 `T` 是一个泛指类型

```
void Swap(T& a, T& b)
{
    T t = a;
    a = b;
    b = t;
}
```

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
```

57.深入理解函数模板

函数模板深入理解

- 编译器从函数模板**通过具体的类型产生不同的函数**
- 编译器会**对函数模板进行两次编译**
 - 对模板本身进行编译
 - 对参数替换后的代码进行编译

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

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);
```

多参数函数模板无法自动推导**返回值类型**，函数模板中的返回值类型必须显示指定

```
// 可以从左向右部分指定类型参数

// T1 = int, T2 = double, T3 = double
int r1 = add<int>(0.5, 0.8);

// 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);
```

返回值参数作为第一个类型参数

重载函数模板

函数模板可以被重载

- 编译器**优先考虑普通函数**
- 如果函数模板可以产生一个**更好的匹配函数**，则选择模板
- 可以通过**空模板实参列表**限定编译器只匹配模板

```
int r1 = Max(1, 2);

double r2 = Max<>(0.5, 0.8);
```

 限定编译器只匹配函数模板

58. 类模板的概念和意义

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

```
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("Ricardo", "Tiffany");
```

- 声明的**泛指类型** `T` 可以出现在类模板的任何地方
- 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（二次编译）

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

```
template <typename T>
class Operator
{
public:
    T add(T a, T b);
    T sub(T a, T b);
    T mul(T a, T b);
    T div(T a, T b);
};

template <typename T>
T Operator<T>::add(T a, T b)
{
    return a + b;
}
//...
```


59.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 <typename T1, typename T2>
class Test
{
    public:
        void add(T1 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**

<pre>template < typename T1, typename T2> class Test { };</pre>	 特化	<pre>template < typename T > class Test < T, T > { };</pre>
---	---	---

部分特化

- 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

<pre>template < typename T1, typename T2> class Test { };</pre>	 完全特化	<pre>template < > class Test < int, int > { };</pre>
---	--	--

特化只是模板的分开实现，本质上是同一个类模板

完全特化类模板必须显示指定每一个类型参数

特化的深度分析

- 重定义
 - 重定义会设计一个类模板和一个新的类（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
 - 统一使用类模板和特化类
 - 编译器**自动优先选择特化类**

- 函数模板只支持类型参数完全特化

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

template <>
bool Equal<double>(double a, double b)
{
    const double delta = 0.000000000000001;
    double r = a - b;
    return (-delta < r) && (r < delta);
}
```

当需要重载函数模板时，优先考虑使用模板特化；当模板特化无法满足需求时，再重载函数模板~
比如在上例加一个参数？

60. 数组类模板

模板参数可以是**数值型参数**（非类型参数）

```
template
<typename T, int N>
void func()
{
    T a[N]; //使用模板参数定义局部数组
}
func<double, 10>();
```

数值型模板参数的限制

- 变量不能作为模板参数
- 浮点数不能作为模板参数
- 类对象不能作为模板参数
- . . .

模板参数是在编译器阶段被处理的单元，在编译阶段必须准确的唯一确定

61. 智能指针类模板

智能指针意义：

- 自动内存管理的主要手段
- 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）**，`auto_ptr` 类里没有数组相关的重载操作符？
- **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**

`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（不能进行所有权转移）

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
- 析构时不会自动销毁所指对象
- **支持多个指针对象指向同一片内存**

`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时才能删除指向的对象

62.单例类模板

在设计时，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）**。

单例模式

- 控制类的对象数目，必须对外隐藏构造函数
 - 将构造函数访问属性设置为 `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
- 存在的问题
 - 需要使用单例模式时：
 - 必须定义静态成员变量 `instance`
 - 必须定义静态成员函数 `GetInstance()`

故：开发单例类模板。

C语言异常处理

异常处理

异常和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

对比：

- 异常
 - 运行时产生除以0的情况
 - 需要打开的外部文件不存在
 - 数组访问越界
- 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

C语言经典处理方式：

if...else

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

setjmp()、longjmp()

- `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)`
 - 从 `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`

缺陷：

- 必然涉及使用全局变量
- 暴力跳转导致代码可读性降低
- 本质还是if...else

64、65.C++中的异常处理

异常处理

- C++内置异常处理的语法元素 `try...catch`
 - `try` 语句处理正常的代码逻辑
 - `catch` 语句处理异常情况

```
try
{
    double r = divide(1, 0);
}
catch(...)
{
    cout << "Divided bt zero..." << endl;
}
```

- 通过 `throw` 语句抛出异常信息

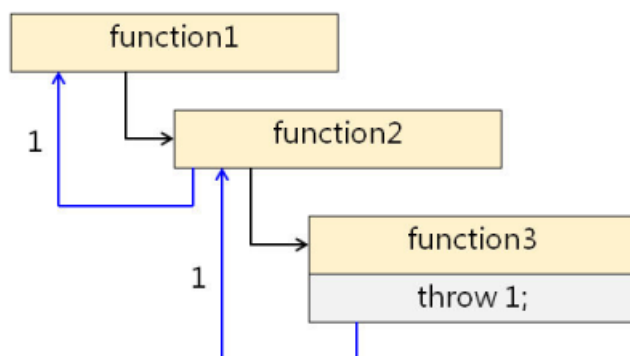
```
double divide(double a, double b)
{
    const double delta = 0.00000000001;
    double ret = 0;

    if (!( (-delta < b) && (b < delta) ))
    {
        ret = a / b;
    }
    else
    {
        throw 0; // 产生除0异常
    }

    return ret;
}
```

- `throw` 抛出的异常必须被 `catch` 处理
 - 当前函数能处理异常，程序继续执行
 - 当前函数无法处理异常，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。



- 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，只能位于多个 catch 语句的最后一个
 - 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。

```
try
{
    throw 1;
}
catch (Type1 t1)
{
}
catch (Type1 t2)
{
}
catch (TypeN tn)
{
}
```

异常处理匹配时，不进
行任何的类型转换。

异常处理必须严格匹配，不进行任何的类型转换

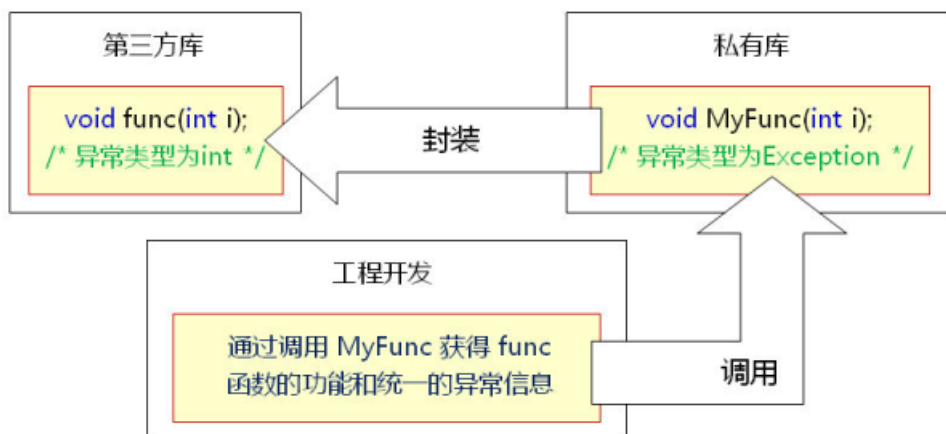
- catch 语句块中可以抛出异常

catch 中抛出的异常需要外
层的 try ... catch ... 捕获

```
try
{
    func();
}
catch (int i)
{
    throw i;
}
catch (...)
{
    throw;
}
```

将捕获的异常重新抛出

重新抛出异常是为了重新解释该异常，显示该异常更多的相关信息，达到统一异常类型的目的



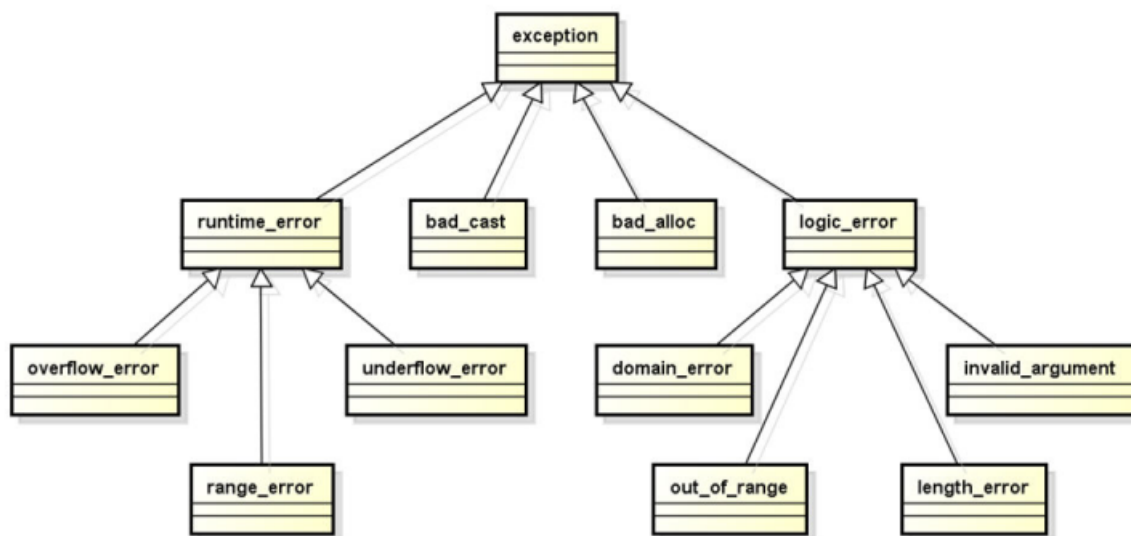
异常的类型

- 异常的类型可以是**自定义类类型**
- 类类型异常的匹配依旧是**至上而下严格匹配**
- 依旧**适用赋值兼容性原则**
 - 匹配子类异常的 `catch` 放在上部
 - 匹配父类异常的 `catch` 放在下部
- 在定义 `catch` 语句块时**推荐使用引用作为参数，避开拷贝构造提升效率**

标准库中的实用异常类族

- 标准库中的异常都是从 `exception` 类派生
- `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（空指针，函数参数错误，下标越界等）
 - `runtime_error`
 - 常用于程序中无法避免的恶性错误（运算产生越界、溢出等）

▪ 标准库中的异常



C++中的类型识别

- 基类指针指向子类对象
- 基类引用成为子类对象的别名

```
Base* p = new Derived();  
Base& r = *p;
```

静态类型

动态类型

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

```
void test(Base* b)
{
    /* 危险的转换方式 */
    Derived* d = static_cast<Derived*>(b);
}
```

动态类型识别

多态：

- 在基类中**定义虚函数**，返回具体的类型信息
- 所有的派生类都**必须实现**类型相关的虚函数
- 每个类中的类型虚函数都**需要不同的实现**

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

- 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

```
int i = 0;
const type_info& tiv = typeid(i);
const type_info& tii = typeid(int);
cout << (tiv == tii) << endl;
```

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

67.经典问题解析五（面试问题）

指针的判别

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
 1. 重载函数
 2. 函数模板
 3. 变参函数
- 思路

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

利用函数模板和变参函数能够判断指针变量

缺陷：

变参函数**无法解析对象参数**，可能造成程序崩溃。

解决方案：

通过 `sizeof` 关键字使得编译器精确匹配函数，但不进行实际调用。

构造和析构中的异常

如果构造函数中抛出异常会发生什么？

- 构造函数中抛出异常
 - 构造过程立即停止
 - **当前对象无法生成**
 - 析构函数不会被调用
 - 对象所占用的空间立即收回

建议：项目中当构造函数可能产生异常时，**使用二阶构造模式**

实验编译：

```
g++ -g 67-2.cpp
valgrind --tool=memcheck --leak-check=full ./a.out
```

析构中的异常：

将导致对象所使用的资源无法完全释放

68.拾遗：令人迷惑的写法

```
template <class T>
class Test
{
public:
    Test(T t){}
};

template <class T>
void func(T a[], int len)
{
}
```

- 早期的C++直接复用class关键字来定义模板
- class关键字的复用使得**代码出现二义性**

定义 `typename` 的原因：

- 自定义类类型内部的**嵌套类型**
- 不同类中的同一个标识符可能导致二义性
- **编译器无法辨识标识符究竟是什么**

`typename` 的作用：

1. 在模板定义中**声明泛指类型**
2. **明确告诉编译器**其后的标识符为类型

```
int func(int i)try
{
    return i;
}
catch(...)
{
    return -1;
}

int func(int i, int j)
throw(int)
{
    return i + j;
}
```

- `try...catch` 用于分隔**正常功能代码**与**异常处理代码**
- `try...catch` 可以直接将函数实现分隔为2部分
- 函数声明和定义时可以**直接指定可能抛出的异常类型**
- 异常声明成为函数一部分可以提高代码可读性
- 函数声明异常后**就只能抛出声明的异常**
 - 抛出其他异常将导致程序运行终止
 - 可以直接通过异常声明**定义无异常函数**

69.技巧：自定义内存管理

笔试题：统计对象中某个成员变量的访问次数

遗失的关键字 `mutable`：

- `mutable` 是为了突破 `const` 函数限制而设计
- `mutable` 成员变量将**永远处于可改变的状态**，破坏了只读对象的内部状态
- `const` 成员函数保证**只读对象的状态不变性**
- `mutable` 成员变量的出现无法保证状态不变性

```
int* const mPvalue;
int* const mPcount;
见 69-1.cpp
```


面试题：new关键字创建出来的对象位于什么地方？

- new/delete 的本质是C++预定义的操作符

new：

1. 获取足够大的内存空间（默认为堆空间）
2. 在获取的空间中调用构造函数创建对象

delete：

1. 调用析构函数销毁对象
2. 归还对象所占用的空间（默认为堆空间）

-
- 在C++中可以重载 new/delete 操作符
 - 全局重载（不推荐）
 - 局部重载（针对具体类进行重载）

意义：改变动态对象创建时的内存分配方式

```
//static member function
void* operator new (unsigned int size)
{
    void* ret = NULL;
    // ret point to allocated memory
    return ret;
}
//static member function
void operator delete (void* p)
{
    // free the memory which is pointed by p
}
```

在静态存储区中创建动态对象：69-2.cpp

面试题：如何在指定的地址上创建C++对象？

解决方案：

- 在类中重载 new/delete 操作符
- 在 new 的操作符重载函数中返回指定的地址
- 在 delete 操作符重载中标记对应的地址可用

自定义动态对象的存储空间：69-3.cpp

-
- new[] / delete[] 与 new / delete 完全不同
 - 动态对象数组创建通过 new[] 完成
 - 动态对象数组销毁通过 delete[] 完成
 - new[] / delete[] 能够被重载，进而改变内存管理方式

```
//static member function
void* operator new[] (unsigned int size)
{
    return malloc(size);
}
//static member function
void operator delete[] (void* p)
{
    free(p);
}
```

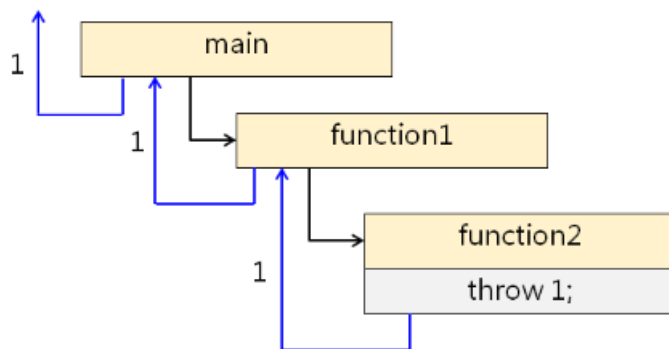
WARNING :

- `new[]` 实际需要返回的内存空间可能比期望的更多
- 对象数组占用的内存中需要保存数组信息
- 数组信息用于确定构造函数和析构函数的调用次数

71.外传1：异常处理深度解析

问题：如果在main函数中抛出异常会发生什么？

???



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

int main()
{
    static Test t;
    throw 1;
    return 0;
}
```

- 如果异常无法被处理，最后 `terminate()` 结束函数会被自动调用
- 默认情况下，`terminate()` 调用库函数 `abort()` 终止程序
- `abort()` 函数使得程序执行异常而立即退出
- C++支持替换默认的 `terminate()` 函数实现

`terminate()` 是整个程序释放系统资源的最后机会

`terminate()` 函数的替换

- 自定义一个无返回值无参数的函数
 - 不能抛出异常
 - 必须以某种方式结束当前程序 (`abort()` or `exit(1)`)
- 调用 `set_terminate()` 设置自定义的结束函数
 - 参数类型为 `void(*)()`
 - 返回值为默认的 `terminate()` 函数入口地址

结束函数可以自定义，但不能继续抛出异常

面试题：如果析构函数中抛出异常会发生什么情况？

可能导致 `terminate()` 多次调用

72.外传2：函数的异常规格说明

问题：如何判断一个函数是否会抛出异常，以及抛出哪些异常？

- 异常声明作为函数声明的修饰符，写在参数列表后面

```
// 可能抛出异常
void func1();
// 只能抛出的异常类型: char 和 int
void func2() throw(char, int);
// 不抛出任何异常
void func3() throw();
```

异常规格说明的意义：

- 提示函数调用者必须做好异常处理的准备
- 提示函数的维护者不要抛出其他异常
- 异常规格说明是函数接口的一部分

问题：如果抛出的异常不在声明列表中，会发生什么？

```
void func() throw(int)
{
    cout << "func()" << endl;
    throw 'c';
}
int main()
{
    try
```

```

{
    func();
}
catch(int)
{
    cout << "catch(int)" << endl;
}
catch(char)
{
    cout << "catch(char)" << endl;
}
return 0;
}

```

- 函数抛出的异常不在规格说明中，全局 `unexpected()` 被调用
- 默认的 `unexpected()` 函数会调用全局的 `terminate()` 函数
- 可以自定义函数替换默认的 `unexpected()` 函数实现

WARNING：不是所有的C++编译器都支持这个标准行为

`unexpected()` 函数的替换

- 自定义一个无返回值无参数的函数
 - 能够再次抛出异常
 - 当异常符合触发函数的异常规格说明时，恢复程序执行
 - 否则，调用全局 `terminate()` 函数结束程序
- 调用 `set_unexpected()` 设置自定义的异常函数
 - 参数类型为 `void(*)()`
 - 返回值为默认的 `unexpected()` 函数入口地址

73.外传3：动态内存申请的结果

问题：动态内存申请一定成功吗？

- `malloc` 函数申请失败时返回NULL值
 - `new` 关键字申请失败时（根据编译器不同）
 - 返回NULL值
 - 抛出 `std::bad_alloc` 异常
-

问题：`new`语句中的异常是怎么抛出来的？

`new` 关键字在C++规范中的标准行为：

—在堆空间申请足够大的内存

- 成功：
 - 获取的空间中调用构造函数创建对象
 - 返回对象的地址
 - 失败：
 - 抛出 `std::bad_alloc` 异常
-

— new 在分配内存时

- 如果空间不足，会调用全局的 `new_handler()` 函数
- `new_handler()` 函数中抛出 `std::bad_alloc` 异常

可以自定义 `new_handler()` 函数来处理默认的new内存分配失败的情况

```
void myNewHandler()
{
    cout << "No enough memory" << endl;
    exit(1);
}

int main()
{
    set_new_handler(myNewHandler);
    //...
    return 0;
}
```

问题：如何跨编译器统一new的行为，提高代码移植性？

解决方案：

- 全局范围（不推荐）
 - 重新定义 `new/delete` 的实现，不抛出任何异常
 - 自定义 `new_handler()` 函数，不抛出任何异常
- 类层次范围
 - 重载 `new/delete`，不抛出任何异常
- 单次动态内存分配
 - 使用 `nothrow` 参数，指明 `new` 不抛出异常

```
int bb[2] = {0};

struct ST
{
    int x;
    int y;
};

ST* pt = new(bb) ST();

pt->x = 1;
pt->y = 2;

cout << bb[0] << endl;
cout << bb[1] << endl;

pt->~ST();
```

不同的编译器在动态内存分配上的实现细节不同

