# 模板

## 模板的继承

可以继承父类类模板的所有实例，也可以继承父类类模板的特例

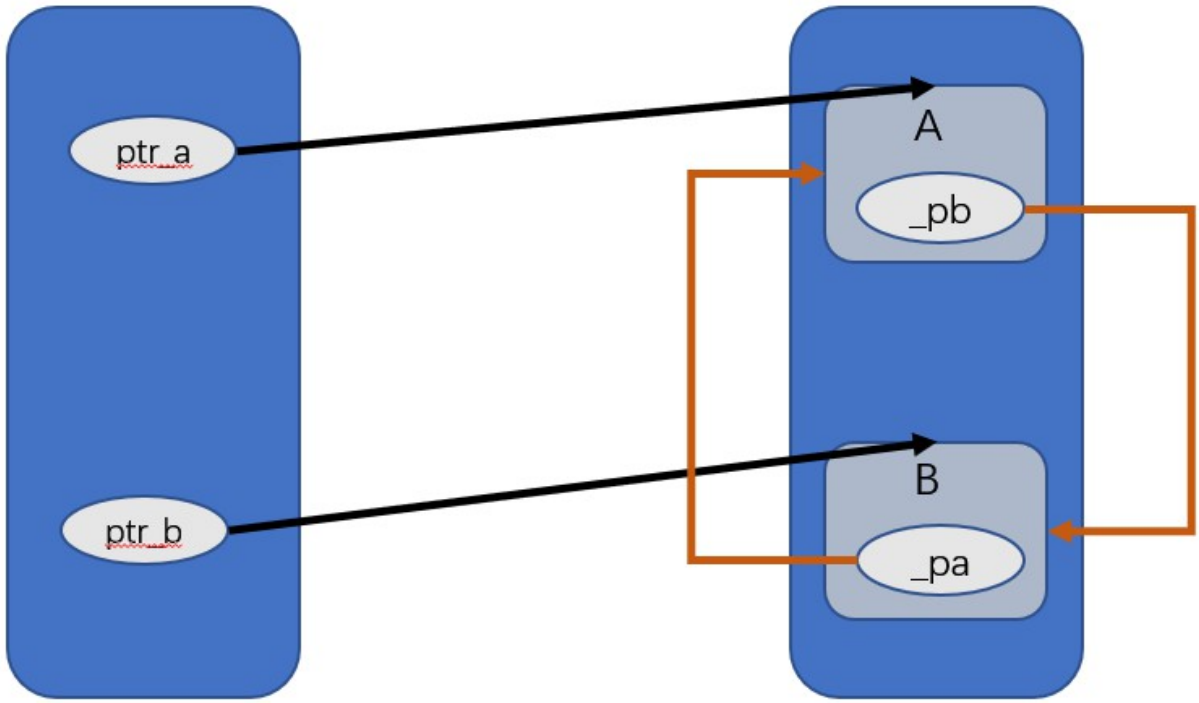
```
template <typename T>
class A {
    ...
};

template <typename T>        // 继承所有实例
class B : public A<T> {
    ...
};

template <typename T>        // 继承int特化
class C : public A<int> {
    ...
};
```

# 智能指针

## 循环引用引来的问题

```
class B;
class A {
    ...
    void setptr(shared_ptr<B> *ptr) { _pb = ptr; }
private:
    shared_ptr<B> _pb;
    ...
};

class B {
    ...
    void setptr(shared_ptr<A> *ptr) { _pa = ptr; }
private:
    shared_ptr<A> _pa;
    ...
};

...
```

```
shared_ptr<A> ptr_a = new A;     // 对象A引用计数+1
shared_ptr<B> ptr_b = new B;     // 对象B引用计数+1

// 循环引用
ptr_a.setptr(ptr_b);             // 对象B引用计数+1
ptr_b.setptr(ptr_a);             // 对象A引用计数+1

// 释放:
// 对象A引用计数-1，还剩1，不会调用析构
// 对象B引用计数-1，还剩1，不会调用析构
```




所有的"强"指针（`shared_ptr`、`unique_ptr`）都会有循环引用的问题，导致内存泄漏

可以利用"弱"指针（`weak_ptr`）解决此类问题

# 异常

C++异常处理使得错误的发现和错误处理分开，使用`try-catch`与`throw`配合完成

- 可能抛出异常的代码放到`try`语句块中，使用`catch`来接受抛出的异常，使用`throw`来抛出异常（可以抛出任意类型）
- 当`throw`语句执行后，程序执行流程跳出`try`语句块，转向`catch`语句块，沿着`catch`语句块查找类型一致的块执行
- 如果一个异常被抛出，但是没有`catch`块接收，程序最终会调用`terminate`，`termintate`会调用`abort`来结束程序，可以使用`set_terminate`函数来设定自己的最终处理函数（一定要调用`abort`或者`exit`）
- 异常处理是完全匹配，不会进行隐式类型转换，但是可以用基类来匹配派生类
- 支持异常的嵌套，抛出异常会逐层捕获直到有`catch`块处理

# try块

将一或多个异常处理块（catch 子句）与复合语句关联

> *try 复合语句 处理块序列*
>
> 其中 *处理块序列* 是一或多个 *处理块* 的序列，它有下列语法：
>
> 1) *catch* ( *attr*(可选) *类型说明符序列 声明符* ) *复合语句*
>
> 2) *catch* ( *attr*(可选) *类型说明符序列 抽象声明符*(可选) ) *复合语句*
>
> 3) *catch* ( ... ) *复合语句*

1. 声明一个具名形参的 catch 子句

```
try { /* */ } catch (const std::exception& e) { /* */ }
```

2. 声明一个无名形参的 catch 子句

```
try { /* */ } catch (const std::exception&) { /* */ }
```

3. catch-all 处理块，可被任何异常激活

```
try { /* */ } catch (...) { /* */ }
```

# 异常与对象

- 在对象遇到异常时，会调用对象的析构
- 构造中抛出异常无法调用析构
- 析构中不能抛出异常

# noexcept关键字

指定函数是否抛出异常

> 1) *noexcept*
>
> 2) *noexcept* ( *表达式* )
>
> 3) *throw()* --- 在**C++20**中被废弃

表达式为`true`，跟`noexcept`一致不抛出异常