Technology Tarena 达内科技

标准C++语言

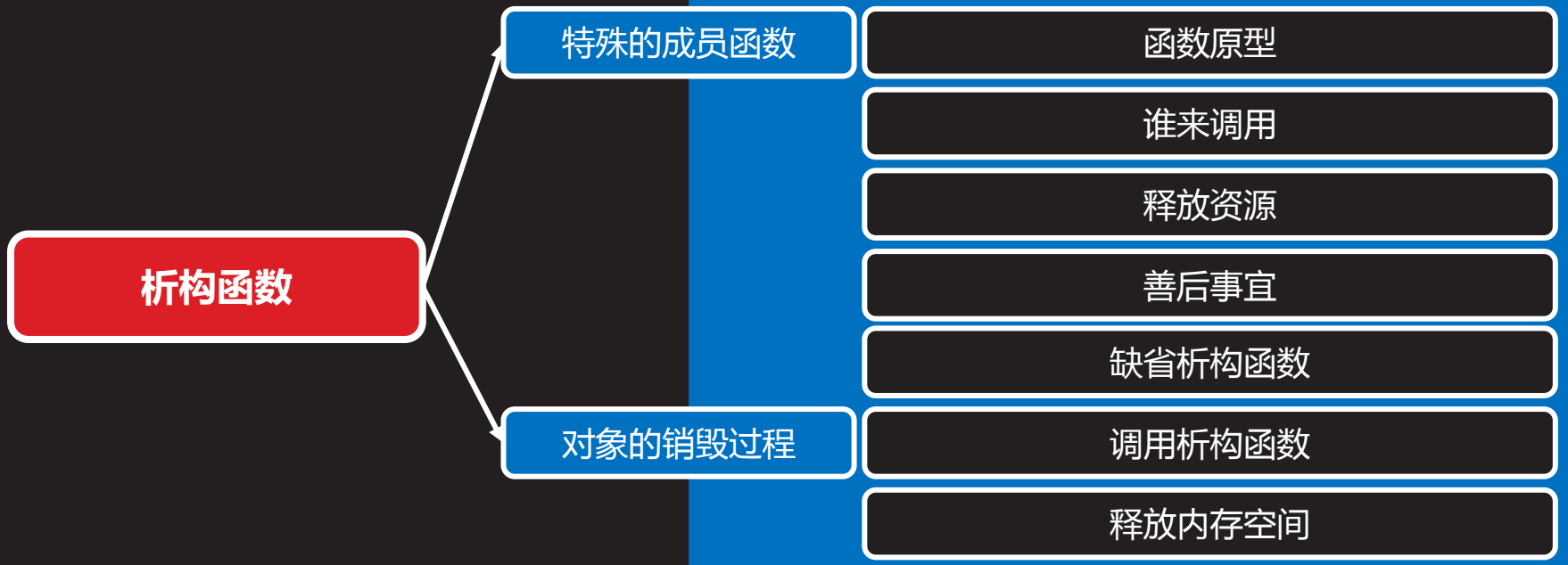
PART 1

DAY04

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	析构函数
	10:30 ~ 11:20	拷贝构造与拷贝赋值
	11:30 ~ 12:20	
下午	14:00 ~ 14:50	静态成员与单例模式
	15:00 ~ 15:50	
	16:00 ~ 16:50	成员指针
	17:00 ~ 17:30	总结和答疑

析构函数


析构函数
特殊的成员函数
函数原型
谁来调用
释放资源
善后事宜
缺省析构函数
对象的销毁过程
调用析构函数
释放内存空间

Technology Tarena 达内科技

特殊的成员函数

函数原型

- 析构函数是特殊的成员函数
  - 析构函数的函数名就是在类名前面加“~”
  - 析构函数没有返回类型
  - 析构函数没有参数
  - 析构函数不能重载

```
class Array {
public:
    ~Array (void) { ... }
};
```

知识讲解

谁来调用

- 析构函数在销毁对象时自动被调用
  - 对象所在作用域的终止花括号调用析构函数

```
{
    Array array (10);
    ...
}
```

  - delete操作符调用析构函数

```
Array* array = new Array (10);
...
delete array;
```

- 析构函数在对象的整个生命周期内，最多被调用一次

知识讲解

释放资源

- 析构函数负责释放在对象的构造过程或生命周期内所获得的资源
  - class Array {
    public:
    Array (size_t size) : m_array (new int[size]) {}
    ~Array (void) {
    delete[] m_array;
    }
    private:
    int* m_array;
    };

知识讲解

善后事宜

- 析构函数的功能不仅限于释放资源，它可以执行任何作为类的设计者希望在最后一次使用对象之后执行的动作
  - class Child {
    public:
    Child (Parent* parent) : m_parent (parent) {
    m_parent->register (this);
    }
    ~Child (void) {
    m_parent->unregister (this);
    }
    private:
    Parent* m_parent;
    };

知识讲解

缺省析构函数

- 通常情况下，如果对象在其生命周期的最终时刻，并不持有任何动态分配的资源，也没有任何善后工作可做，那么完全可以不为其定义析构函数
- 如果一个类没有定义析构函数，那么编译器会为其提供一个缺省析构函数
  - 对基本类型的成员变量，什么也不做
  - 对类类型的成员变量和基类子对象，调用相应类型的析构函数
- 缺省析构函数由编译器提供，它只负责释放编译器看得到资源，如成员子对象、基类子对象等

知识讲解

缺省析构函数（续1）

- 对于编译器看不到的资源，如通过malloc或new动态分配的资源，缺省析构函数不负责释放，必须通过自己定义的析构函数予以释放，否则将形成内存泄漏
  - class User {

```
User (string const& name) :
    m_name (new string (name)),
    m_info (static_cast<char*> (malloc (1024))) {}
~User (void) {
    delete m_name;
    free (m_info); }
string* m_name;
char* m_info;
};
```

知识讲解

缺省析构函数不释放动态分配的资源

【参见：TTS COOKBOOK 】

课堂练习

- 缺省析构函数不释放动态分配的资源

对象的销毁过程

调用析构函数

- 销毁对象的第一步即调用析构函数，完成如下任务
  1. 执行析构函数体代码
  2. 逆序调用类类型成员的析构函数，析构所有成员子对象
  3. 逆序调用各个基类的析构函数，析构所有基类子对象
- 注意，执行析构函数体代码是整个析构过程的第一步，这保证了析构函数体代码所依赖的一切资源和先决条件，在该代码被执行时尚且未被销毁，并保持析构前的状态

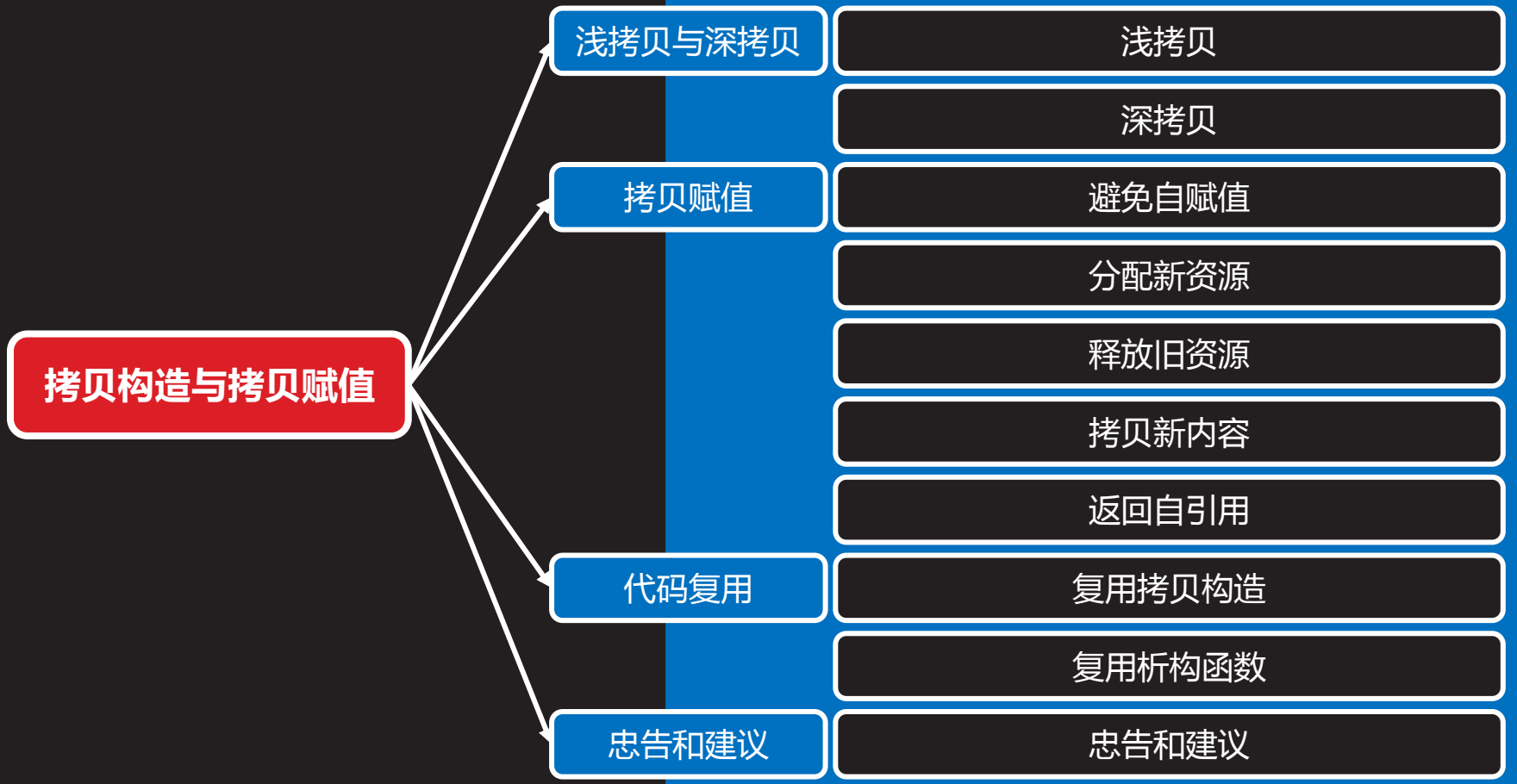
知识讲解

释放内存空间

- 销毁对象的第二步是用类似free的机制，将包括成员子对象、基类子对象等在内的，整个对象内存空间释放掉
- 根据目前绝大多数C++编译器的实现，销毁对象的过程和创建对象的过程，通常是严格相反的
  - 创建：分配内存->构造基类->构造成员->执行构造代码
    基类：按继承顺序，从左至右，依次构造
    成员：按声明顺序，从上至下，依次构造
  - 销毁：执行析构代码->析构成员->析构基类->释放内存
    基类：按继承顺序，从右至左，依次析构
    成员：按声明顺序，从下至上，依次析构

知识讲解

拷贝构造与拷贝赋值

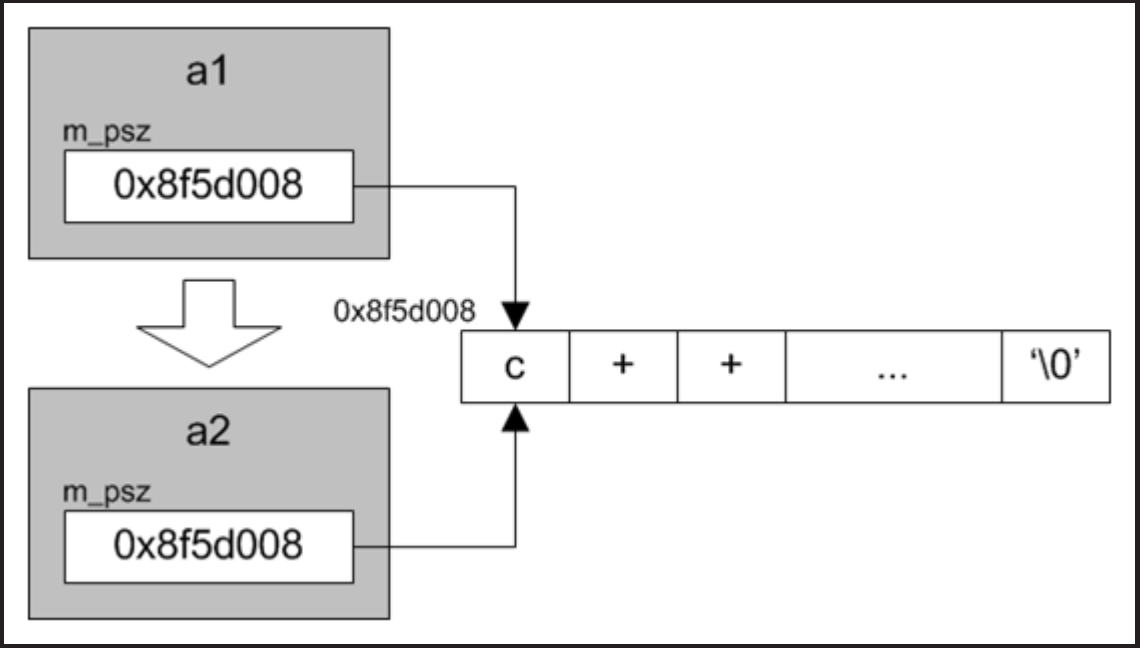

拷贝构造与拷贝赋值
浅拷贝与深拷贝
浅拷贝
深拷贝
拷贝赋值
避免自赋值
分配新资源
释放旧资源
拷贝新内容
返回自引用
代码复用
复用拷贝构造
复用析构函数
忠告和建议
忠告和建议

Technology Tarena 达内科技

浅拷贝与深拷贝

浅拷贝

- 缺省方式的拷贝构造和拷贝赋值，对包括指针在内的基本类型成员变量按字节复制，导致浅拷贝问题
  - String::String (const String& that) :
    m_psz (that.m_psz) {}
  - String a1 = "C++ is very easy !", a2 = a1;



知识讲解

缺省拷贝构造函数仅支持浅拷贝

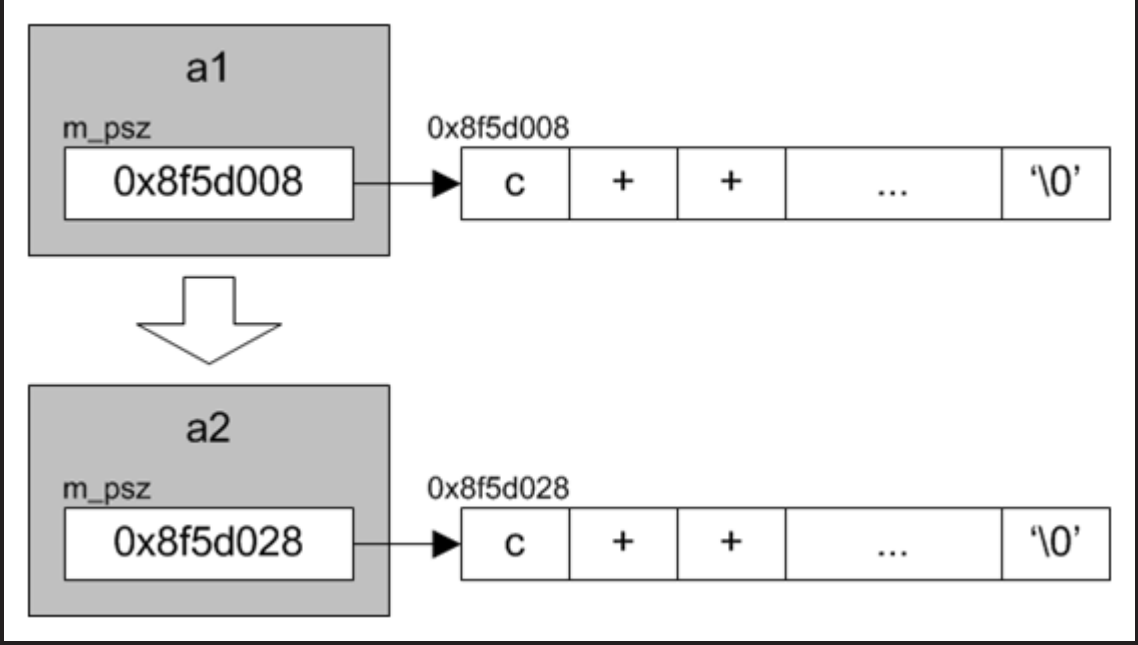
【参见：TTS COOKBOOK 】

课堂练习

- 缺省拷贝构造函数仅支持浅拷贝

深拷贝

- 为了获得完整意义上的对象副本，必须自己定义拷贝构造和拷贝赋值，针对指针型成员变量做深拷贝
  - String::String (const String& that) :
    m_psz (strcpy (new char[strlen (that.m_psz) + 1],
    that.m_psz)) {}



自定义拷贝构造函数支持深拷贝

【参见：TTS COOKBOOK 】

课堂练习

- 自定义拷贝构造函数支持深拷贝

Technology Tarena 达内科技

拷贝赋值

避免自赋值

- 将一个对象赋值给自身，最妥当的做法就是什么也不做
  - void String::operator= (String const& rhs) {
    if (&rhs != this) {
    }
    }

知识讲解

分配新资源

- 为了使目标对象拥有独立的拷贝，需要为其分配足够的资源
  - void String::operator= (String const& rhs) {
    if (&rhs != this) {
    char* psz = new char[strlen (rhs.m_psz) + 1];
    }
    }

知识讲解

释放旧资源

- 释放目标对象原先所持有的资源，防止内存泄漏
  - void String::operator= (String const& rhs) {
    if (&rhs != this) {
    char* psz = new char[strlen (rhs.m_psz) + 1];
    delete[] m_psz;
    }
    }
- 先分配新资源再释放旧资源是有意义的，即使目标对象的新资源分配失败，其原有的内容也不至于遭到破坏

知识讲解

拷贝新内容

- 将源对象的内容复制到目标对象中
  - void String::operator= (String const& rhs) {
    if (&rhs != this) {
    char* psz = new char[strlen (rhs.m_psz) + 1];
    delete[] m_psz;
    m_psz = strcpy (psz, rhs.m_psz);
    }
    }

知识讲解

返回自引用

- 返回对调用对象自身的引用，以满足赋值表达式的语义
  - String& String::operator= (String const& rhs) {
    if (&rhs != this) {
    char* psz = new char[strlen (rhs.m_psz) + 1];
    delete[] m_psz;
    m_psz = strcpy (psz, rhs.m_psz);
    }
    return *this;
    }
- 赋值表达式的值应该是其左操作数的引用
  - (a = b) = c; // b->a, c->a

知识讲解

支持深拷贝的拷贝赋值运算符函数

【参见：TTS COOKBOOK 】

课堂练习

- 支持深拷贝的拷贝赋值运算符函数

Technology Tarena 达内科技

代码复用

复用拷贝构造

- 在目标对象内部直接构造源对象的副本，复用拷贝构造函数
  - String& String::operator= (String const& rhs) {
    if (&rhs != this) {
    String str (rhs);
    }
    return *this;
    }

知识讲解

复用析构函数

- 交换目标对象和副本对象的资源指针，一方面使目标对象获得副本对象的资源，另一方面利用副本对象的局部变量身份，复用其析构函数，释放目标对象原有的资源
  - String& String::operator= (String const& rhs) {
    if (&rhs != this) {
    String str (rhs);
    swap (m_psz, str.m_psz);
    }
    return *this;
    }

知识讲解

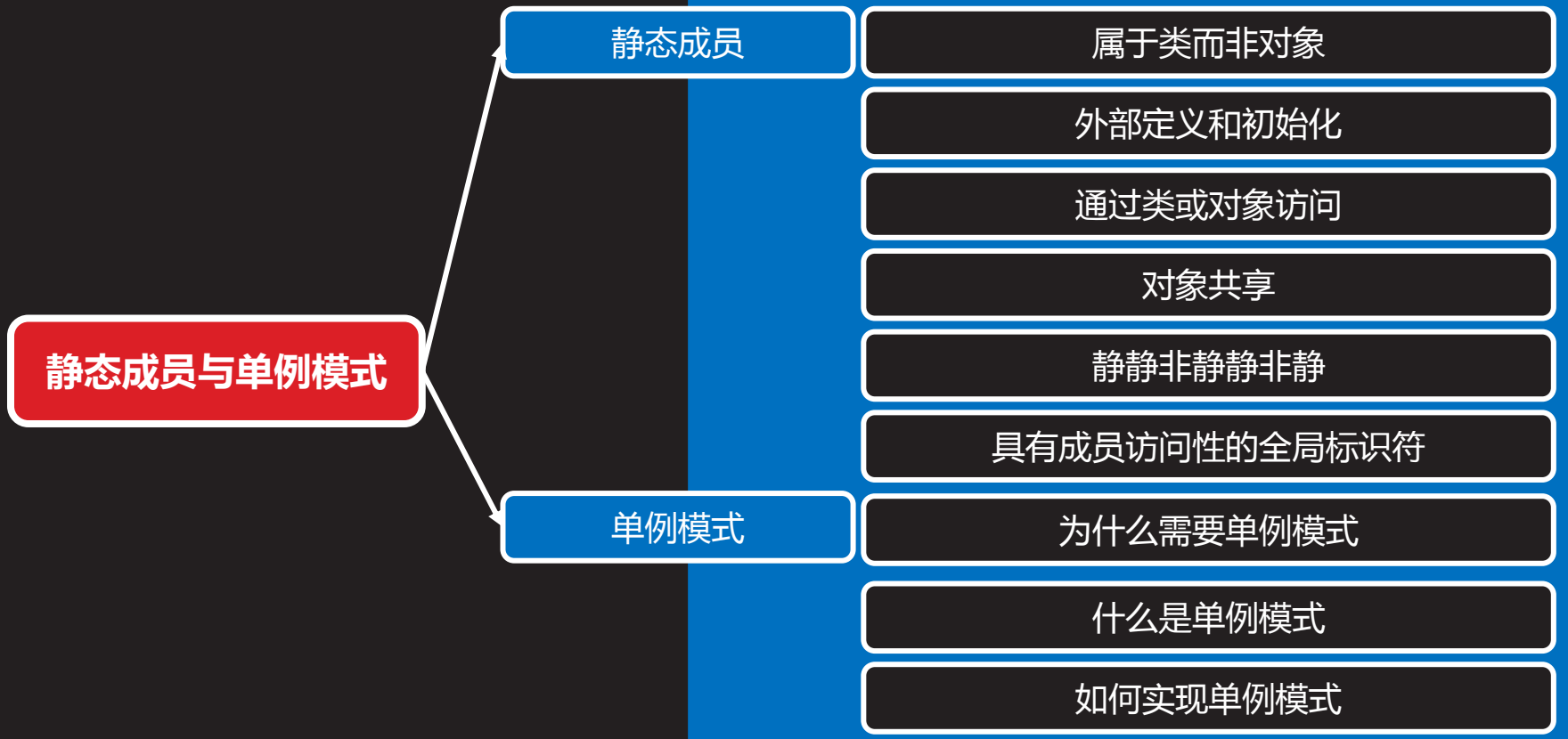
Technology Tarena 达内科技

忠告和建议

忠告和建议

- 无论是拷贝构造还是拷贝赋值，其缺省实现对类类型成员变量和基类子对象，都会调用相应类型的拷贝构造函数和拷贝赋值运算符函数，而不是简单地按字节复制，因此应尽量避免使用指针型成员变量
- 尽量通过引用或指针向函数传递对象型参数，既可以降低参数传递的开销，也能减少拷贝构造的机会
- 出于具体原因的考虑，确实无法实现完整意义上的拷贝构造和拷贝赋值，可将它们私有化，以防误用
- 如果为一个类提供了自定义的拷贝构造函数，就没有理由不提供实现相同逻辑的拷贝赋值运算符函数

静态成员与单例模式


静态成员与单例模式
静态成员
属于类而非对象
外部定义和初始化
通过类或对象访问
对象共享
静静非静静非静
具有成员访问性的全局标识符
单例模式
为什么需要单例模式
什么是单例模式
如何实现单例模式

静态成员

属于类而非对象

- 静态成员变量不包含在对象实例中，具有进程级的生命周期
- 静态成员函数没有this指针，也没有常属性
- 静态成员依然受类作用域和访问控制限定符的约束

知识讲解

外部定义和初始化

- 静态成员变量的定义和初始化，只能在类的外部而不能在构造函数中进行
  - class Account {
    public:
    static double s_interest;
    };
  - double Account::s_interest = 0.001;

知识讲解

通过类或对象访问

- 访问静态成员，既可以通过类也可以通过对象
  - Account acc (...), pacc = &acc;
    acc.s_interest = 0.002;
    pacc->s_interest = 0.003;
    Account::s_interest = 0.004;

知识讲解

对象共享

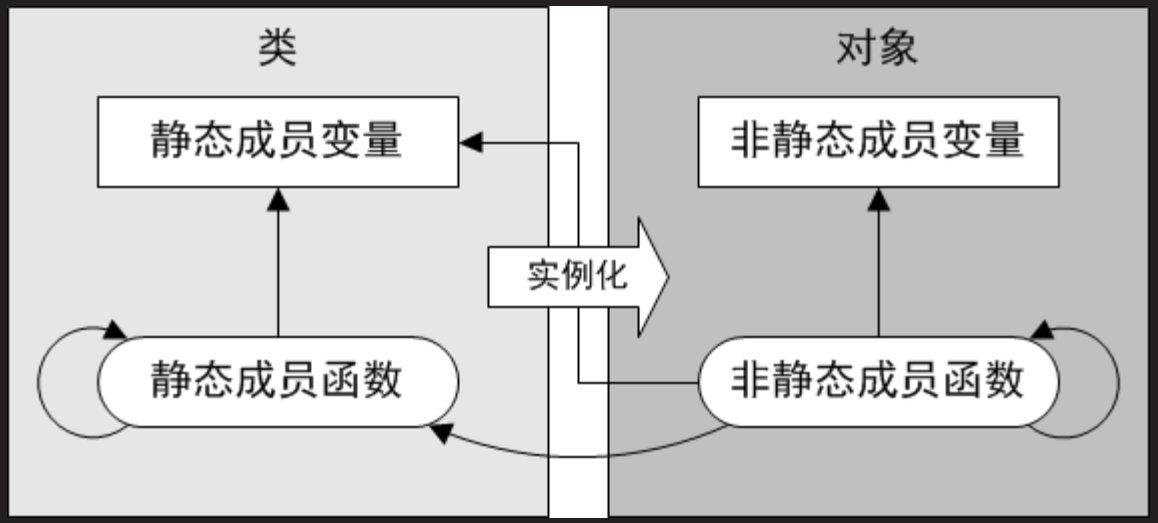
- 静态成员变量为该类的所有对象实例所共享
  - Account acc1 (...), acc2 (...), acc3 (...);

```
acc1. s_interest += 0.001;
cout << Account::s_interest << endl; // 0.002
acc2. s_interest += 0.002;
cout << Account::s_interest << endl; // 0.004
acc3. s_interest += 0.003;
cout << Account::s_interest << endl; // 0.007
Account::s_interest -= 0.004;
cout << acc1.s_interest << endl; // 0.003
cout << acc2.s_interest << endl; // 0.003
cout << acc3.s_interest << endl; // 0.003
```

静静非静静非静

- 静态成员函数只能访问静态成员变量或调用静态成员函数
- 非静态成员函数既可以访问静态成员变量或调用静态成员函数，也可以访问非静态成员变量或调用非静态成员函数

知识讲解



具有成员访问性的全局标识符

- 事实上，类的静态成员变量和静态成员函数，更象是普通的全局变量和全局函数，只是多了一层类作用域和访问控制属性的限制，相当于具有成员访问性的全局变量和全局函数
- 类的静态成员变量通常用于表示需要在该类的多个实例化对象间共享的属性，如银行账户的利率
- 类的静态成员函数一方面可以独立于对象充当静态成员变量的访问接口，另一方面它也可以在不破坏封装性的前提下，让一个类有能力管理自己的对象

知识讲解

支持结息调息功能的银行账户类

【参见：TTS COOKBOOK 】

课堂练习

- 支持结息调息功能的银行账户类

Technology Tarena 达内科技

单例模式

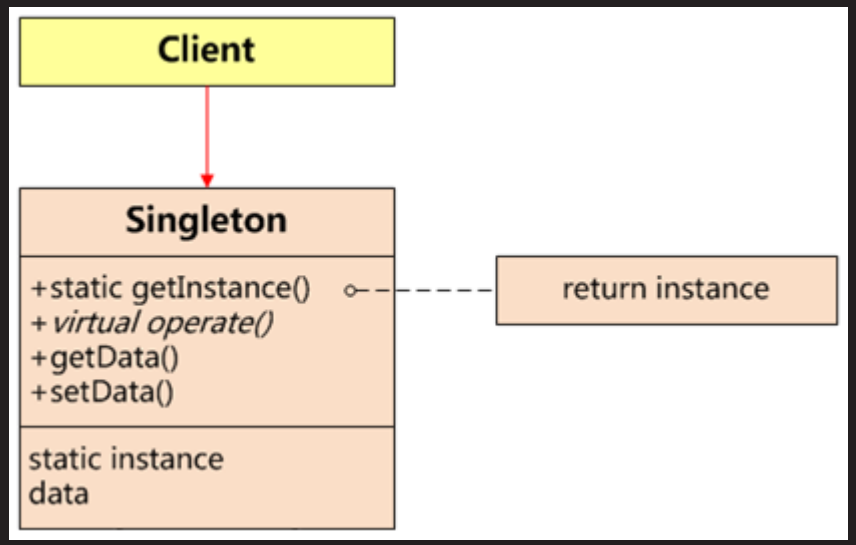
为什么需要单例模式

- 只允许存在唯一对象实例
- 单例模式的商业应用
  - 网站计数器
  - 日志管理系统
  - 连接池、线程池、内存池
- 获取对象实例的专门方法
  - 全局变量的定义不受控制，防君子不防小人
  - 专门方法是类型的一部分，我是类型我做主借助于类型禁止在外部创建对象仅在类型内部创建唯一对象实例提供用于获取对象实例公知接口

知识讲解

什么是单例模式

- 禁止在类外部创建实例
  - 私有构造函数
    private
- 类自己维护其唯一实例
  - 静态成员变量
    instance
- 提供访问该实例的方法
  - 静态成员函数
    getInstance()



知识讲解

如何实现单例模式

- 饿汉式实现
  - 将单例类的唯一对象实例，实现为其静态成员变量
  - 优点：加载进程时静态创建单例对象，线程安全
  - 缺点：无论使用与否，总要创建
- 懒汉式实现
  - 直到第一次使用时，才创建单例类的唯一对象实例
  - 优点：用则创建，不用不创建，什么时候用什么时候创建
  - 缺点：首次访问时动态创建单例对象，在多线程应用中，存在线程不安全的问题

知识讲解

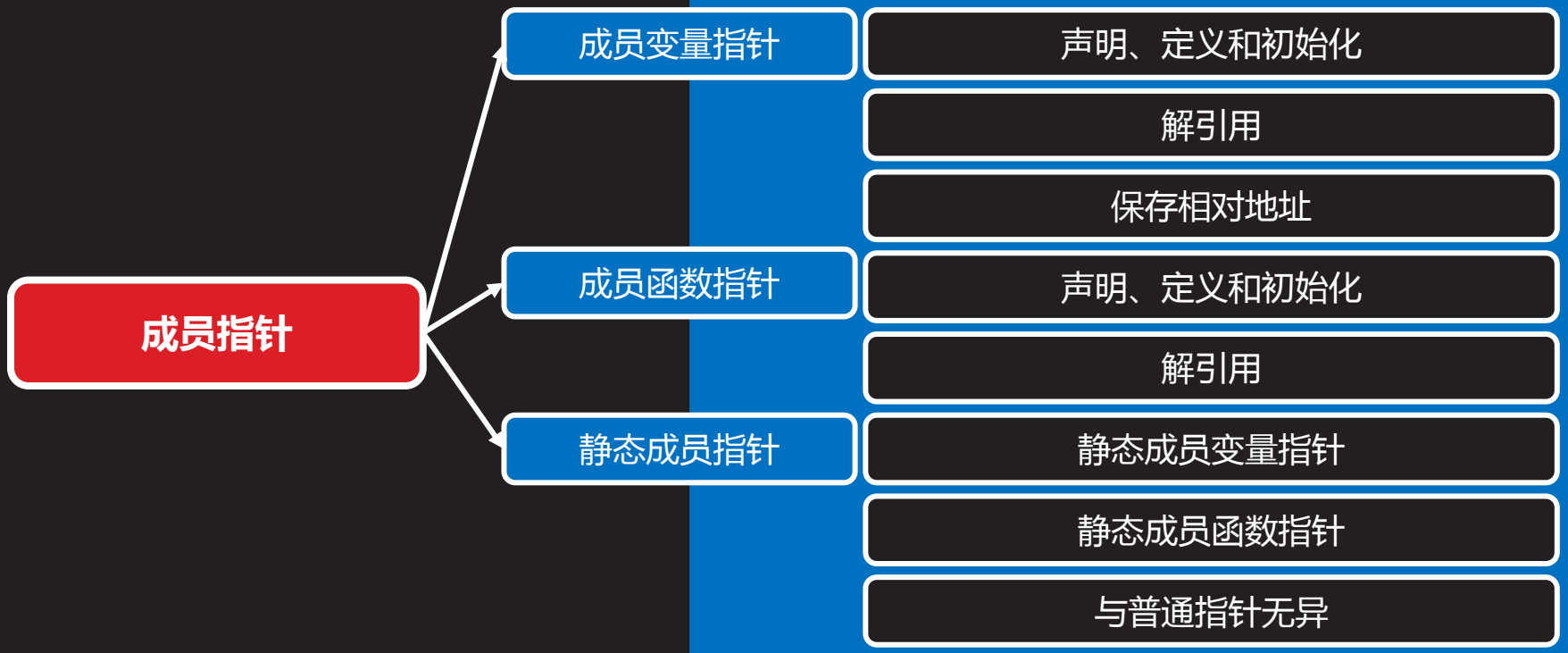
单例模式

【参见：TTS COOKBOOK 】

课堂练习

- 单例模式

成员指针


成员指针
成员变量指针
声明、定义和初始化
解引用
保存相对地址
成员函数指针
声明、定义和初始化
解引用
静态成员指针
静态成员变量指针
静态成员函数指针
与普通指针无异

Technology Tarena 达内科技

成员变量指针

声明、定义和初始化

- 类型 类名::*成员变量指针 = &类名::成员变量;
  - class Student {
    public:
        string m_name;
        string m_sex;
        int m_age;
        int m_no;
    };
  - int Student::*p_age = &Student::m_age;

知识讲解

解引用

- 对象.*成员变量指针
  - “.*”是一个独立的运算符，成员指针解引用运算符

```
Student student (...);
student.*p_age = 25;
```

- 对象指针->*成员变量指针
  - “->*”是一个独立的运算符，间接成员指针解引用运算符

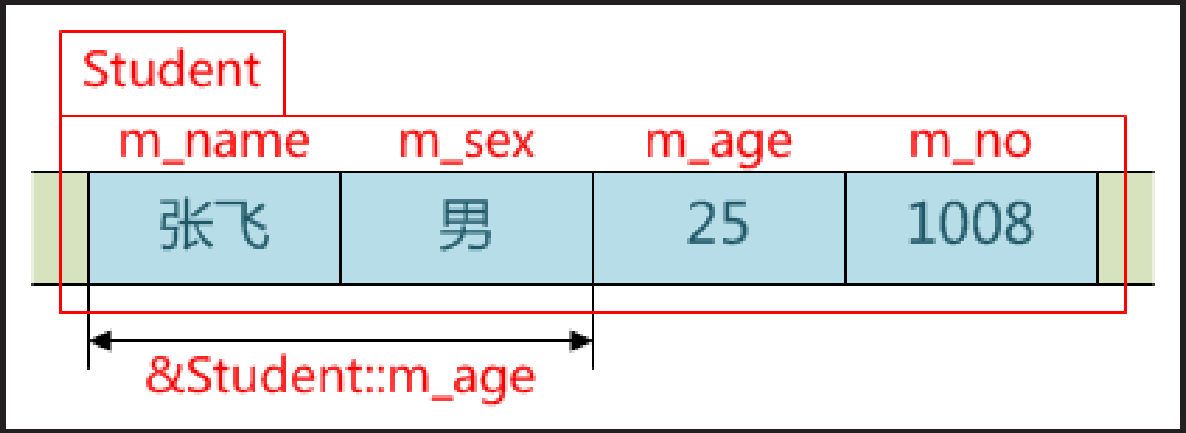
```
Student* student = new Student (...);
student->*p_age = 25;
```

知识讲解

保存相对地址

- 成员变量指针的本质，就是特定成员变量在类对象实例中的相对地址
- 成员变量指针解引用，就是根据类对象实例的起始地址，结合成员变量指针中的相对地址，计算出具体成员变量的绝对地址，并访问之

知识讲解



Technology Tarena 达内科技

成员函数指针

声明、定义和初始化

- 返回类型 (类名::*成员函数指针) (形参表) =
  &类名::成员函数名;
  - class Student {
    public:
      void who (void) const {
        cout << "我是" << m_name
          << "，性别" << m_sex
          << "，今年" << m_age
          << "岁，学号" << m_no << endl;
      }
    };
  - void (Student::*p_who) (void) const = &Student::who;

知识讲解

解引用

- (对象.*成员函数指针) (实参表)
  - Student student (...);
    (student.*p_who) ();
- (对象指针->*成员函数指针) (实参表)
  - Student* student = new Student (...);
    (student->*p_who) ();
- 虽然成员函数并不存储在对象中，不存在根据相对地址计算绝对地址的问题，但也要通过对象或对象指针对成员函数指针解引用，其目的只有一个，即提供this指针

知识讲解

Technology Tarena 达内科技

静态成员指针

静态成员变量指针

- 声明、定义和初始化
  - 类型* 静态成员变量指针 = &类名::静态成员变量;

    ```
    class Student { public: static Teacher s_teacher; };
    Teacher* p_teacher = &Student::s_teacher;
    ```
- 解引用
  - 静态成员变量指针->成员

    ```
    cout << p_teacher->m_age << endl;
    ```
  - *静态成员变量指针

    ```
    cout << (*p_teacher).m_age << endl;
    ```
- 类的静态成员变量不是对象的一部分，不需要根据相对地址计算绝对地址，也不需要通过对象或其指针解引用

知识讲解

静态成员函数指针

- 声明、定义和初始化
  - 返回类型 (*静态成员函数指针) (形参表) =
    类名::静态成员函数名;

```
class Student { public: static Student* create (void); }
Student* (*p_create) (void) = Student::create;
```

- 解引用
  - 静态成员函数指针 (实参表)

```
Student* student = p_create ();
```

- 类的静态成员函数没有this指针，无需调用对象

知识讲解

与普通指针无异

- 静态成员与对象无关，因此静态成员指针与普通指针并没有任何本质性区别
- 所有成员指针，无论静态与否，都受访问控制属性约束
  - class Student {
    private:
    int m_age;
    static Teacher s_teacher;
    };
  - int Student::*p_age = &Student::m_age; // 错误
    Teacher* p_teacher = &Student::s_teacher // 错误

知识讲解

通过成员指针访问对象

【参见：TTS COOKBOOK 】

课堂练习

- 通过成员指针访问对象

Technology Tarena 达内科技

总结和答疑