Technology Tarena 达内科技

标准C++语言

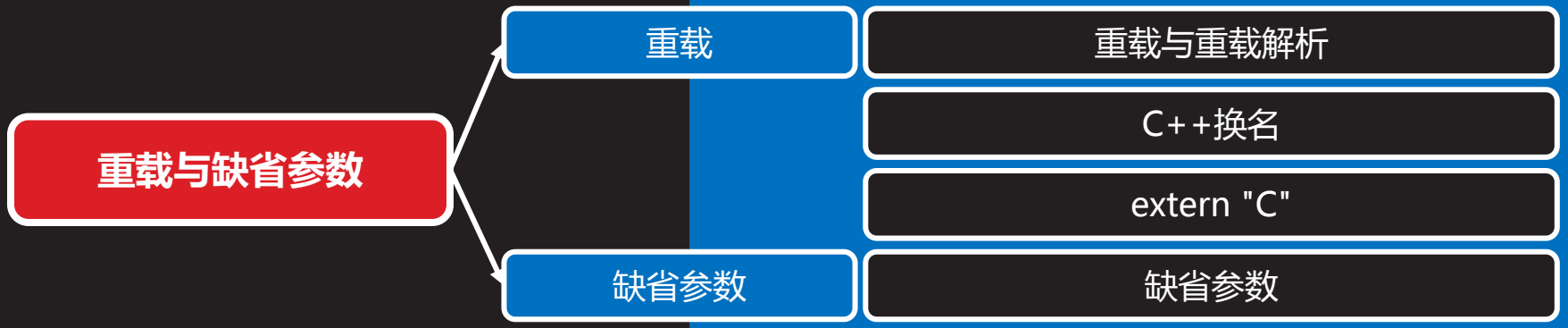
PART 1

DAY02

内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	重载与缺省参数
	10:30 ~ 11:20	内联与动态内存分配
	11:30 ~ 12:20	引用
下午	14:00 ~ 14:50	显式类型转换
	15:00 ~ 15:50	类和对象
	16:00 ~ 16:50	
	17:00 ~ 17:30	总结和答疑

重载与缺省参数



Technology Tarena 达内科技

重载

重载与重载解析

- 同一作用域中，函数名相同，参数表不同的函数，构成重载关系
  - void foo (void);
    void foo (int n);
    void foo (int* p);
    char const* foo (int n, double d);
    char const* foo (double d, int n);
- 重载与返回类型无关，与参数名也无关，相同类型的引用与非引用不构成重载
  - int foo (void); // 错误
    void foo (int& r); // 错误
    char const* foo (int d, double n); // 错误

知识讲解

重载与重载解析（续1）

- 调用函数时，根据实参与形参的类型匹配情况，选择一个确定的重载版本，这个过程称为重载解析
  - void foo (void);
    void foo (int n);
    void foo (int* p);
    int foo (double d);
    char const* foo (int n, double d);
    char const* foo (double d, int n);
  - int n; double d;
    foo (); foo (n); foo (&n);
    foo (d); foo (n, d); foo (d, n);
- 函数指针的类型决定其匹配的重载版本

知识讲解

重载与重载解析（续2）

- 只有同一作用域中的同名函数才涉及重载问题，不同作用域中的同名函数遵循名字隐藏原则
  - void foo (void);
    namespace ns1 {
    void foo (int a);
    namespace ns2 {
    void foo (int a, int b);
    void bar (void) { foo (10, 20); }
    }
    void hum (void) { foo (30); }
    }
    void fun (void) { foo (); }

知识讲解

重载与重载解析

【参见：TTS COOKBOOK 】

课堂练习

- 重载与重载解析

C++换名

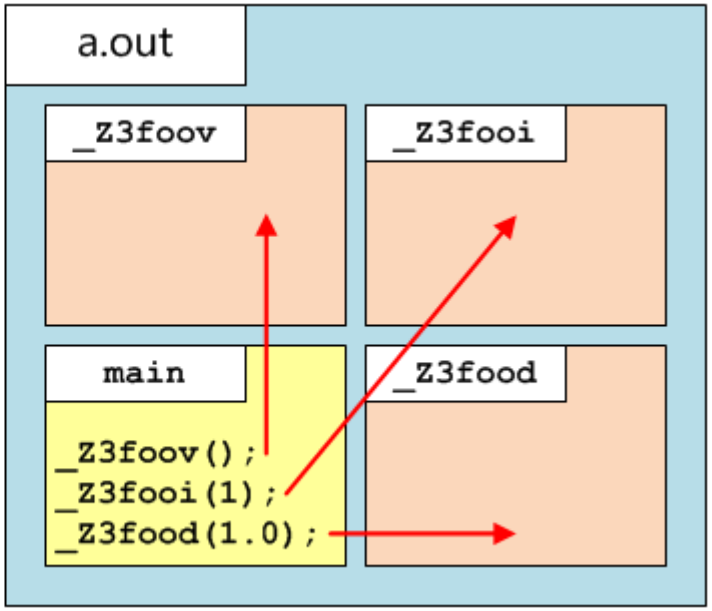
- 重载是通过C++换名实现的
- 换名机制限制了C和C++模块之间的交叉引用

知识讲解

overload.cpp

```
void foo (void) { ... }
void foo (int i) { ... }
void foo (double d) { ... }

int main (void) {
    foo ();
    foo (1);
    foo (1.0);
    return 0;
}
```



extern "C"

- 通过extern "C"可以要求C++编译器按照C方式处理函数接口，即不做换名，当然也就无法重载
  - C调C++，在C++中

```
extern "C" int add (int x, int y);
extern "C" {
    int add (int x, int y);
    int sub (int x, int y);
}
```

  - C++调C，在C++中

```
extern "C" {
    #include "chead.h"
}
```

知识讲解

C形式的调用规格

【参见：TTS COOKBOOK 】

课堂练习

- C形式的调用规格

Technology Tarena 达内科技

缺省参数

缺省参数

- 可以为函数的参数指定缺省值，调用该函数时若未指定实参，则与该实参相对应的形参取缺省值
  - void foo (int a, int b = 456);
  - foo (100); // 编译为foo (100, 456);
- 如果函数的某一个参数具有缺省值，那么该参数后面的所有参数必须都具有缺省值
- 函数的缺省参数是在编译阶段解决的，因此只能用常量、常量表达式或者全局变量等非局部化数值作为缺省参数
  - int g = 123; void foo (int a = g, int b = 400+56);
  - void foo (int a, int b = a); // 错误
- 如果需要将函数的声明与定义分开，那么函数参数的缺省值只能出现在函数的声明部分

知识讲解

Technology Tarena 达内科技

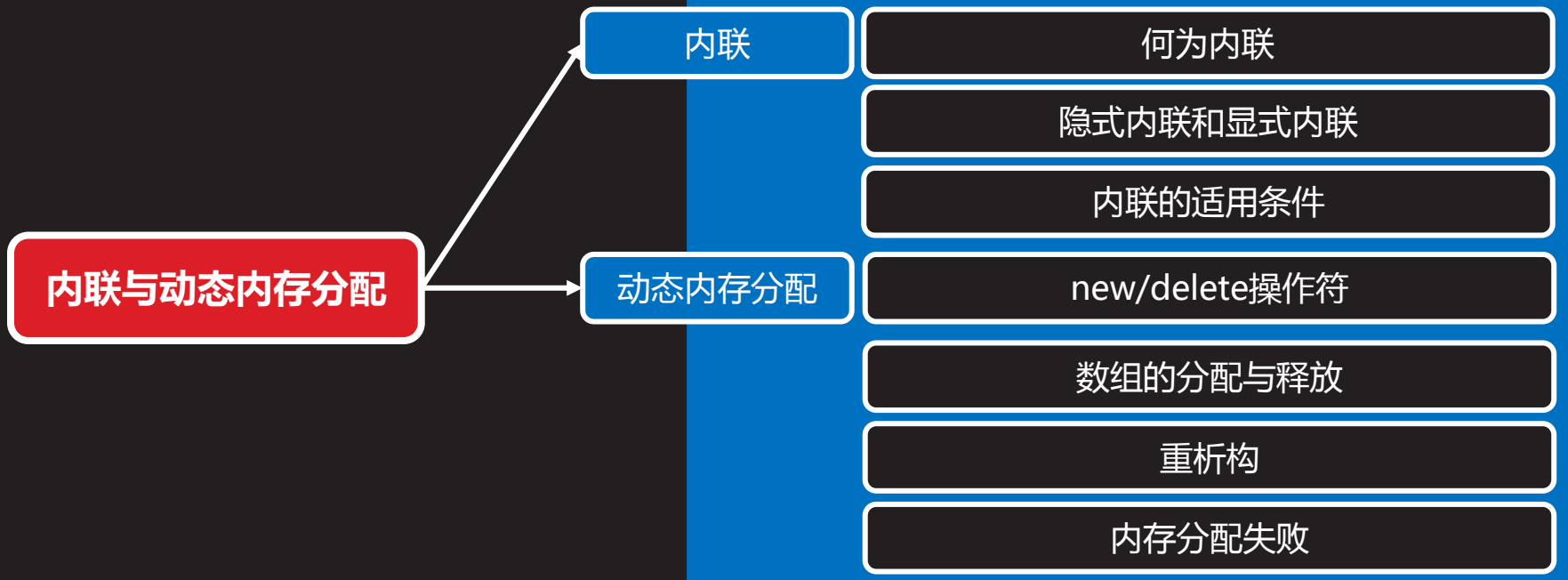
缺省参数

【参见：TTS COOKBOOK 】

课堂练习

- 缺省参数

内联与动态内存分配


内联与动态内存分配
内联
动态内存分配
何为内联
隐式内联和显式内联
内联的适用条件
new/delete操作符
数组的分配与释放
重析构
内存分配失败

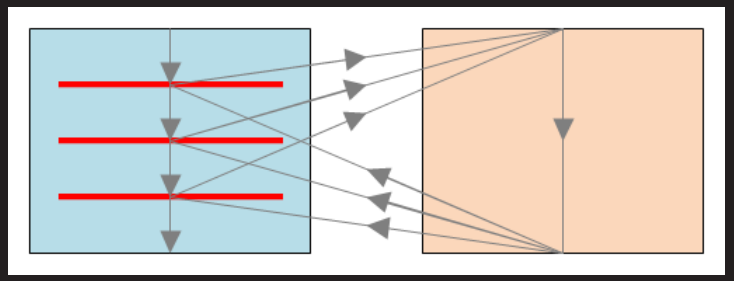
Technology Tarena 达内科技

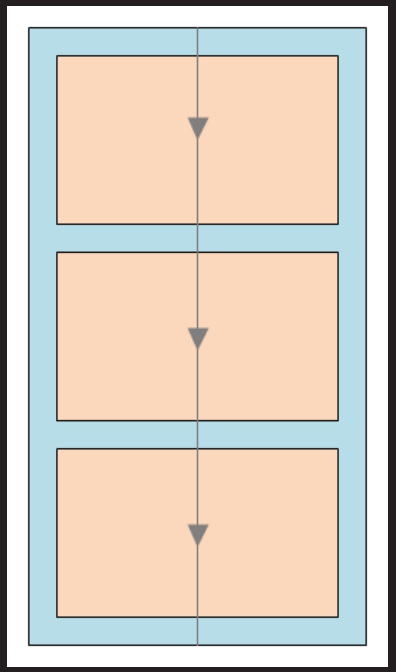
内联

何为内联

- 内联就是用函数已被编译好的二进制代码，替换对该函数的调用指令
  - 内联在保证函数特性的同时，避免了函数调用的开销
  - 内联通过牺牲代码空间，赢得了运行时间

知识讲解





隐式内联和显式内联

- 内联通常被视为一种编译优化策略
- 若函数在类或结构体的内部直接定义，则该函数被自动优化为内联函数，谓之隐式内联
  - struct User {
    void who (void) { … }
    };
- 若在函数定义前面，加上inline关键字，可以显式告诉编译器，该函数被希望优化为内联函数，这叫显式内联
  - inline void foo (void) { … }

知识讲解

内联的适用条件

- 内联会使可执行文件的体积和进程代码的内存变大，因此只有频繁调用的简单函数才适合内联
- 稀少被调用的复杂函数，调用开销远小于执行开销，由内联而获得的时间性能的改善，不足以抵消空间性能的损失，故不适合内联
- inline关键字仅仅表示一种对函数实施内联优化的期望，但该函数是否真的会被处理为内联，还要由编译器的优化策略决定
- 带有递归调用或动态绑定特性的函数，无法实施内联，编译器会忽略其声明部分的inline关键字

知识讲解

Technology Tarena 达内科技

动态内存分配

new/delete操作符

- C++提供了new和delete操作符，分别用于动态内存的分配和释放
  - int* p = new int;
    delete p;
- C++的new操作符允许在动态分配内存时对其做初始化
  - int* p = new int;
  - int* p = new int ();
  - int* p = new int (100);

知识讲解

数组的分配与释放

- 以数组方式new的也要以数组方式delete
  - int* p = new int[4] {10, 20, 30, 40};
    delete[] p;
- 某些C++实现，用new操作符动态分配数组时，会在数组首元素前面多分配4或8个字节，用以存放数组的长度
- new操作符所返回的地址是数组首元素的地址，而非所分配内存的起始地址
- 如果将new操作符返回的地址直接交给delete操作符，将导致无效指针(invalidate pointer)异常
- delete[]操作符会将交给它的地址向低地址方向偏移4或8个字节，避免了无效指针异常的发生

知识讲解

重析构

- 不能通过delete操作符释放已释放过的内存
  - int* p = new int;
    delete p;
    delete p; // 核心转储
  - 标准库检测到重析构(double free)异常后将进程杀死，并转储进程映像
- delete野指针后果未定义，delete空指针安全
  - int* p = new int;
    delete p;
    p = NULL;
    delete p; // 什么也不做

知识讲解

内存分配失败

- 内存分配失败，new操作符抛出bad_alloc异常
  - char* p = (char*)malloc (0xFFFFFFFF);
    if (p == NULL) {
        cerr << "内存分配失败！" << endl;
        exit (-1);
    }
  - try {
        char* p = new char[0xFFFFFFFF];
    }
    catch (bad_alloc& ex) {
        cerr << "内存分配失败！" << endl;
        exit (-1);
    }

知识讲解

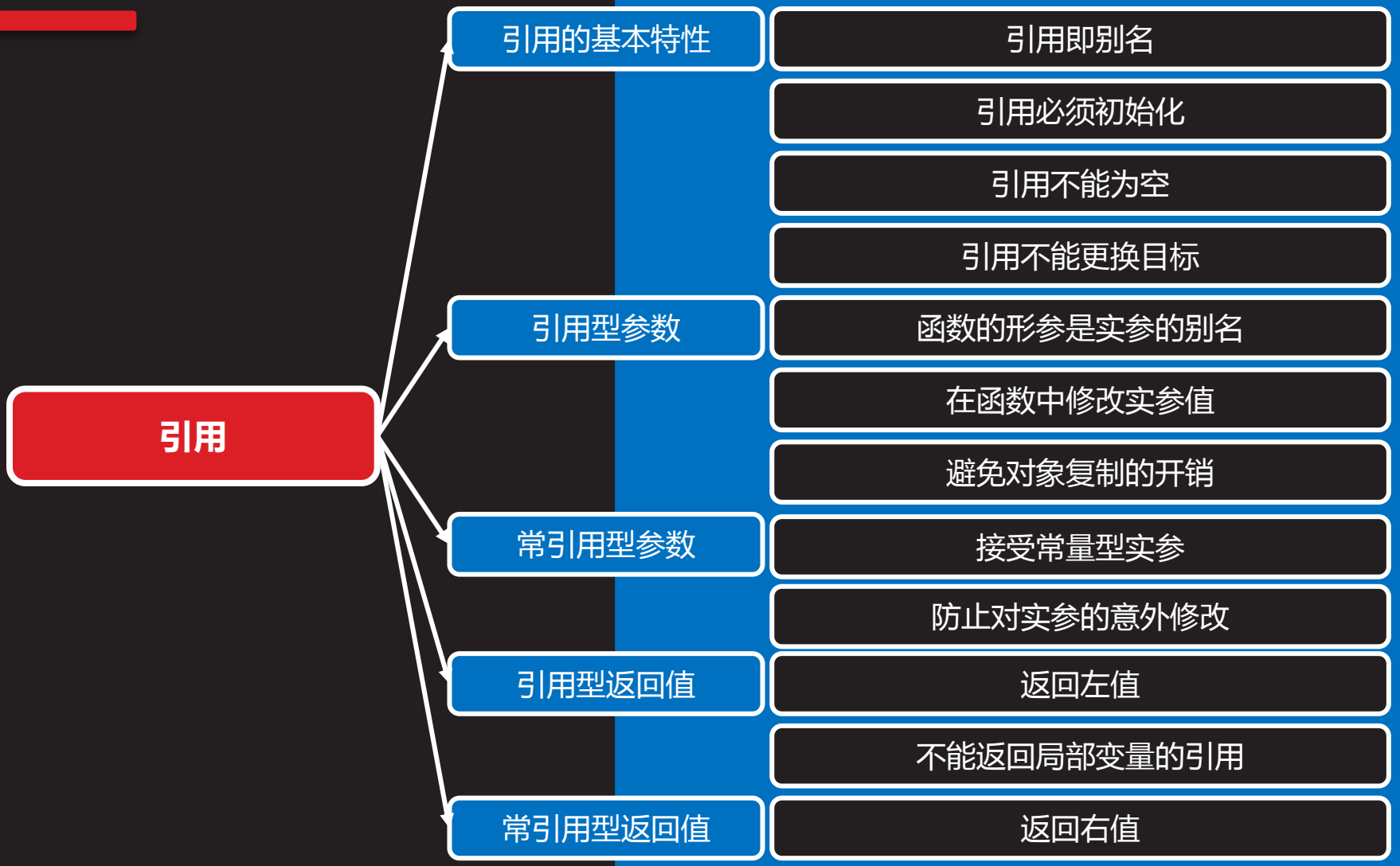
new和delete

【参见：TTS COOKBOOK 】

课堂练习

- new和delete

引用

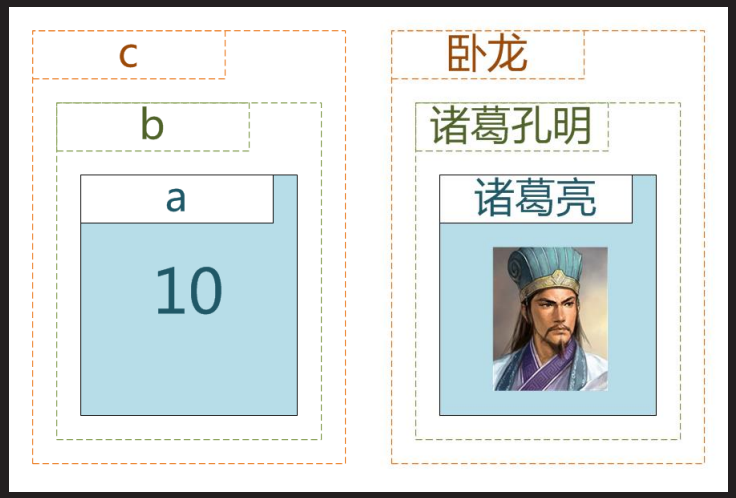

引用
引用的基本特性
引用即别名
引用必须初始化
引用不能为空
引用不能更换目标
引用型参数
函数的形参是实参的别名
在函数中修改实参值
避免对象复制的开销
常引用型参数
接受常量型实参
防止对实参的意外修改
引用型返回值
返回左值
不能返回局部变量的引用
常引用型返回值
返回右值

Technology Tarena 达内科技

引用的基本特性

引用即别名

- 声明一个标识符为引用，即表示该标识符可作为另一个有名或无名对象的别名
  - int a = 10;
    int& b = a; int& c = b;
    ++c;
    cout << a << endl; // 11
    int const& d = c;
    ++d; // ERROR



- 在C++中，无名对象通常都被处理为右值，只能通过带有常属性的引用引用之
  - int const& a = 10; // 纯右值
    int const& b = 20;
    int const& c = a + b; // 将亡右值

知识讲解

引用必须初始化

- 引用必须在定义的同时初始化，不允许先定义再赋值
  - int a = 10;
    int& b = a; // OK
  - int a = 10;
    int& b; // ERROR
    b = a;

知识讲解

引用不能为空

- 无法定义一个什么都不引用的引用
  - int& r = NULL; // ERROR
- 但“野引用”或称“悬空引用”确实是存在的
  - int& r = *new int (1);
    ++r;
    cout << r << endl; // 2
    delete &r;
    ++r; // 未定义
- 解引用一个野引用，就跟解引用一个野指针一样，其结果将是未定义的，可能导致崩溃(段错误)，可能意外地修改了其它有效堆内存中的数据，也可能什么也没有发生而且结果还很正确，但没有人能保证到底会是哪种结果

引用不能更换目标

- 引用一经初始化便不能再引用其它对象
  - int a = 10, b = 20;
    int& c = a; // c是a的引用，a是c的目标
    c = b; // 将b的值赋给c的目标即a，而非令c引用b
- 引用只有在其定义及初始化语境中具有“名”语义，一旦完成了定义及初始化，引用就和普通变量名一样，被赋予了“实”语义，即代表它的目标，而不是别名本身

知识讲解

引用即别名

【参见：TTS COOKBOOK 】

课堂练习

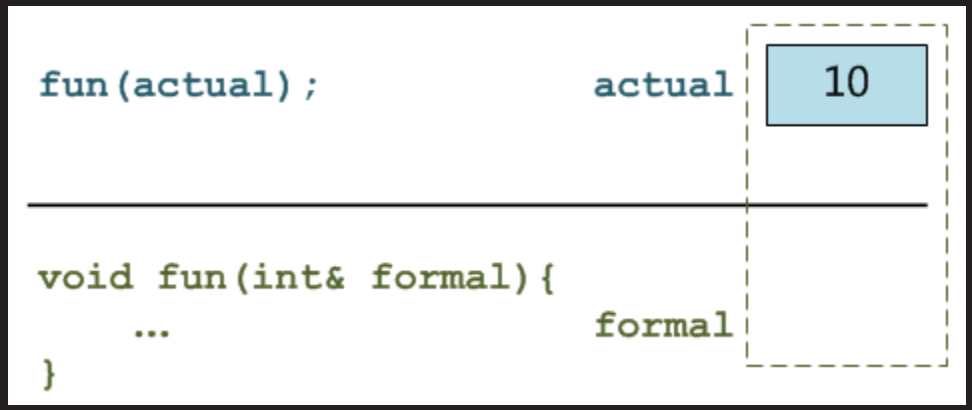
- 引用即别名

Technology Tarena 达内科技

引用型参数

函数的形参是实参的别名

- 可以将函数的形参声明为引用形式，该形参在参数传递过程中由对应位置的实参初始化，并成为该实参的别名
  - void fun (int& formal) {
    cout << &formal << " : " << formal << endl;
    }
  - int actual = 10;
    cout << &actual << " : " << actual << endl;
    fun (actual);



知识讲解

在函数中修改实参值

- 通过引用型形参，可以在函数体内部修改调用者实参的值，成为除返回值和指针参数之外，第三种由函数内部向函数外部输出数据的途径
  - double rect (double w, double h, double* c, double& s) {
    *c = (w + h) * 2;
    s = w * h;
    return sqrt (w * w + h * h);
    }
  - double c, s, d = rect (4, 3, &c, s);
    cout << "对角线长度：" << d << endl;
    cout << "矩形的周长：" << c << endl;
    cout << "矩形的面积：" << s << endl;

知识讲解

避免对象复制的开销

- 通过引用传递参数，形参只是实参的别名而非副本，这就避免了从实参到形参的对象复制，这对于具有复杂数据结构的参数类型而言意义非常
  - struct User {
    char name[64];
    char address[256];
    char mbox[128];
    };
  - void insert (User& user) { … }
  - User user = { … };
    insert (user);

知识讲解

交换变量的值

【参见：TTS COOKBOOK 】

课堂练习

- 交换变量的值

Technology Tarena 达内科技

常引用型参数

接收常量型实参

- 在C++中，一切常量均带有右值属性
- 用内容只读的右值对象初始化目标可写的左值引用，将被编译器以放松类型限定为由予以拒绝
- 常引用型的形参因其对目标的只读性约束，满足了编译器类型限定从紧不从松的原则，可以接收常量型的实参
- C++98中只有左值引用没有右值引用，因此只能用常左值引用引用右值，当然常左值引用也能引用常或非常左值，故该引用又被称为万能引用
  - int sum (int const& x, int const& y) { return x + y; }
  - int a = 20, b = 30, c = sum (10, a - b);
    // 字面值常量10是纯右值，a - b表达式的值是将亡右值

知识讲解

防止对实参的意外修改

- 即便所传递的实参不是常量，只要根据设计，函数不需要也不应该对该实参做任何修改，那么接收该实参的形参同样可以被声明为常引用，这样一方面避免了对象复制的开销，同时一旦做出对实参的意外修改，将直接引发编译错误，将修改实参所带来的风险降到最低
  - int sum (int const& x, int const& y) {
    ++x; // 错误
    y += 100; // 错误
    return x + y;
    }
  - int a = 20, b = 30, c = sum (a, b);

知识讲解

常引用型参数

【参见：TTS COOKBOOK 】

课堂练习

- 常引用型参数

Technology Tarena 达内科技

引用型返回值

返回左值

- 值形式的函数返回值通常都具有右值属性，即在函数的调用者空间根据函数的返回类型创建一个匿名对象，负责接收该函数的返回值
- 用于接收函数返回值的匿名对象和表达式的值类似，通常只具有语句级生命期且只读，即所谓将亡右值
- 如果函数返回的是一个引用，那么用于接收该返回值的就不再是一个匿名的将亡右值对象，而是一个引用该函数所返回引用的目标对象的引用，甚至可以是左值引用
- 函数返回的左值引用可与其它任何形式的左值一样，参与包括赋值、复合赋值、自增减等在内的各种左值运算

知识讲解

不能返回局部变量的引用

- 函数的局部变量只具有函数级甚至块或语句级的生命周期，函数一旦返回，所有的局部变量即刻销毁，即使通过返回值获得了对它们的引用，其目标也将是未定义的
  - int& foo (void) { int n = 123; return n; } // 危险
    int* bar (void) { int n = 456; return &n; } // 危险
    int hum (void) { int n = 789; return n; }
  - int& foo (int& n) { return n; }
    int& bar (int* n) { return *n; }
    int& hum (int n) { return n; } // 危险
  - int* foo (int& n) { return &n; }
    int* bar (int* n) { return n; }
    int* hum (int n) { return &n; } // 危险

知识讲解

返回有效引用

【参见：TTS COOKBOOK 】

课堂练习

- 返回有效引用

Technology Tarena 达内科技

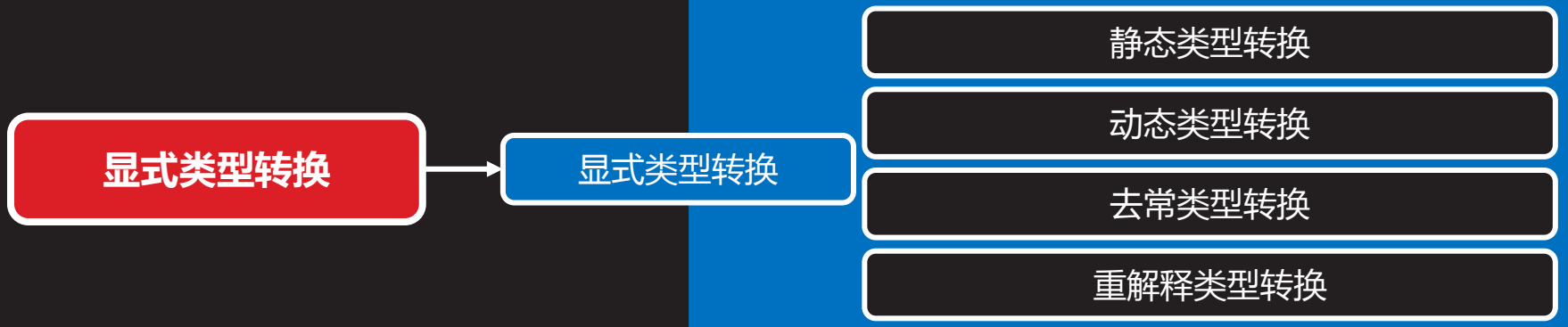
常引用型返回值

返回右值

- 值形式的函数返回值天然具有右值属性，但从函数返回值跟通过值向函数传参数一样，伴随着对象的复制过程
  - int foo (void) { ... }
    foo () = 1234; // 错误
- 为了避免对象复制的开销，同时又不想失去作为函数返回值的右值属性，可以返回一个常左值引用，模拟或者逼近右值的使用效果
  - int& foo (void) { ... }
    foo () = 1234;
  - int const& foo (void) { ... }
    foo () = 1234; // 错误

知识讲解

显式类型转换



Technology Tarena 达内科技

显式类型转换

静态类型转换

- static_cast<目标类型> (源类型对象)
  - 编译器对源类型和目标类型做相容性检查，检查不通过报错
  - 源类型和目标类型只要在一个方向上可以做隐式类型转换，那么在两个方向上就都可以做静态类型转换
  - 如果将目标类型从源类型的类型转换构造函数，或者源类型向目标类型的类型转换运算符函数，被声明为explicit，那么从源类型到目标类型的类型转换就必须显式完成，静态类型转换可用于这样的场合

知识讲解

动态类型转换

- dynamic_cast<目标类型>(源类型对象)
  - 编译器首先检查源类型和目标类型是否同为指针或引用且其类型之间存在具有多态性的继承关系，不存在直接报错
  - 编译器生成一段指令，运行时执行该指令，检查源和目标的类型是否一致，不一致通过返回空指针或抛出异常报错
  - 常被用于具有多态继承关系的父子类对象的指针或引用之间的转换

知识讲解

去常类型转换

- const_cast<目标类型> (源类型对象)
  - 编译器检查源类型和目标类型是否同为指针或引用，且其目标类型之间除常属性以外必须完全相同，否则直接报错
  - 去除指针或引用上的const属性

```
int const volatile x = 100;
int const* p = &x;
*p = 200; // 错误
int* q = const_cast<int*> (p);
*q = 200;
```

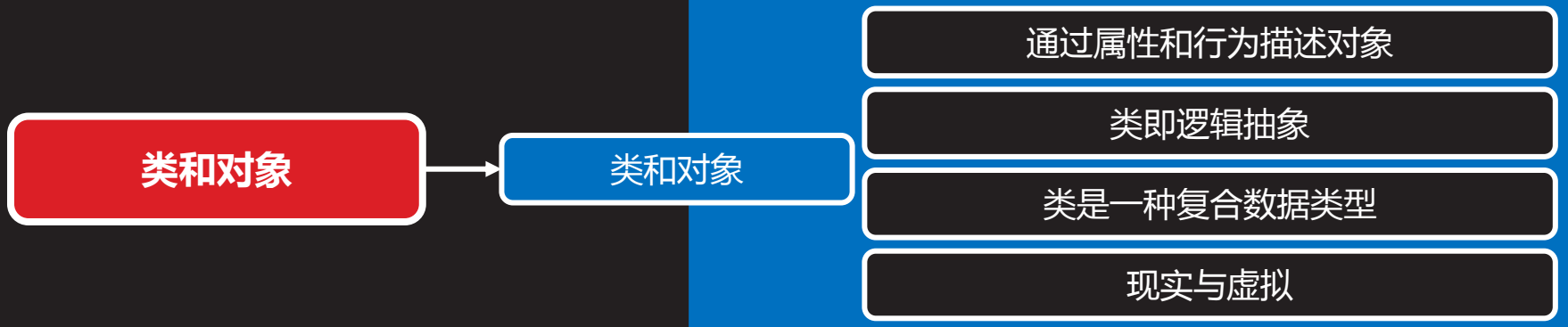
知识讲解

重解释类型转换

- reinterpret_cast<目标类型>(源类型对象)
  - 编译器检查源类型和目标类型是否同为指针或引用，或者一个是指针一个是整型，否则直接报错
  - 在任意类型的指针或引用之间转换，意味着可以将同一个对象视作不同的类型，并以不同的方式访问或处理之
  - 无论何种类型的指针，从本质上讲都与整数无异，即地址空间中一个特定字节的顺序号

知识讲解

类和对象



Technology Tarena 达内科技

类和对象

通过属性和行为描述对象

- 拥有相同属性和行为的对象被分成一组，即一个类

对象		属性	行为
狗		犬种 犬龄 体重 毛色	进食 睡眠 玩耍
学生		姓名 年龄 学号	吃饭 睡觉 学习
手机		品牌 型号 价格	接打电话 收发短信 上网 玩游戏

知识讲解

类即逻辑抽象

- 类可用于表达那些不能直接与内置类型建立自然映射关系的逻辑抽象
  - 简单类型：只能表示一个属性(变量)
  - 数组类型：可以表示多个属性(变量)，但类型必须相同
  - 结构体类型：可以表示多个属性(变量)，且类型可以不同，但缺少对行为(函数)的描述(C++的结构体可以)
  - 类类型：可以表示一到多个属性(变量)，类型可以相同也可以不同，同时提供对多种不同行为(函数)的描述

知识讲解

类是一种复合数据类型

- 类是一种用户自定义的复合数据类型，即包括表达属性的成员变量，也包括表达行为的成员函数
  - struct Dog {

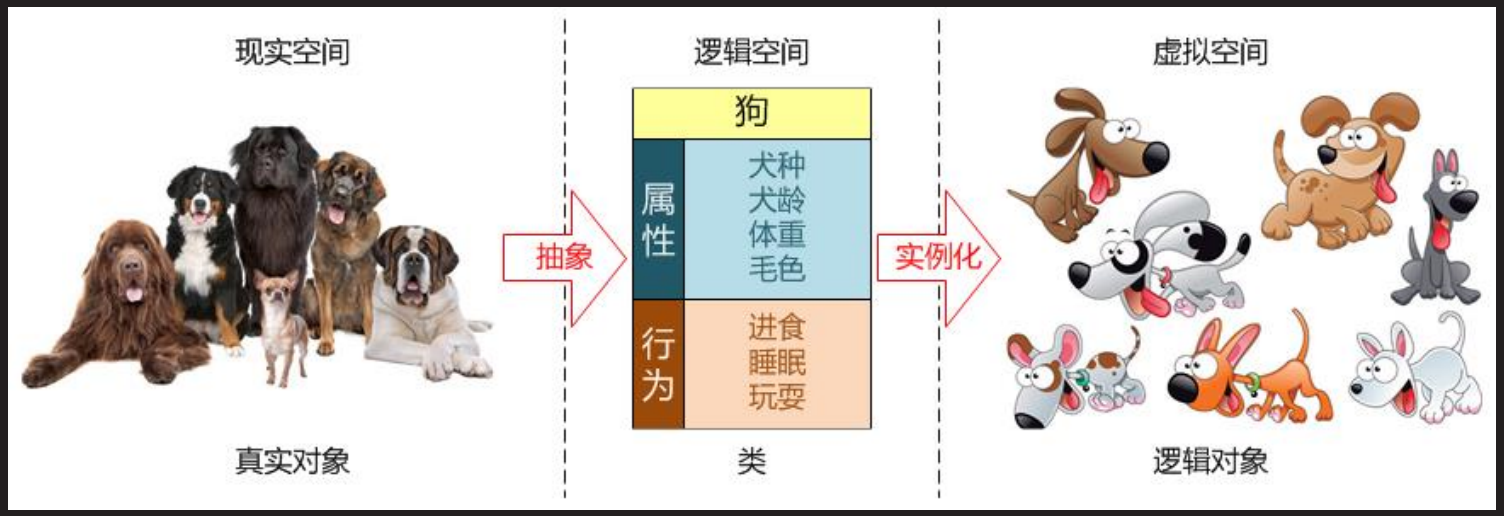
```
char breed[256];
int age;
float weight;
COLOR color;
void eat (string const& food);
void sleep (int duration);
void play (void);
};
```

知识讲解

现实与虚拟

- 类是现实世界的抽象，对象是类在虚拟世界的实例
  - Dog dog = {"吉娃娃", 1, 0.5, BROWN};
    dog.eat ("骨头"); dog.sleep (1); dog.play ();
  - 一条一岁大，半公斤重的棕色吉娃娃，啃了一块骨头，睡了一个小时，愉快地去玩耍。。。

知识讲解



Technology Tarena 达内科技

总结和答疑