



C++期末不挂科

CH2 - C++函数

本章内容

- 复习函数
- 函数默认参数
- 引用与引用传参
- 函数重载
- 内联函数

理解以下名词：

- 参数、返回值与函数体
- 声明与定义
- 形参与实参
- 默认参数
- 引用
- 按值传参、指针传参与引用传参
- 函数重载
- 内联函数 (inline关键字)

回答以下三项技术分别解决了什么问题：

- 引用传参
- 内联函数
- 函数重载

熟悉以下题型：

- 判断两个函数是否能构成重载
- 判断默认参数是否会导致歧义
- 使用引用传参实现就地修改函数参数的效果
- 判断何处适用内联函数

复习函数

函数默认参数

引用与引用传参

函数重载

内联函数 (inline)

函数

- 函数**声明** (原型)
- 函数**定义** (实现)
- 函数的**调用** (call)



函数的**参数**



函数体

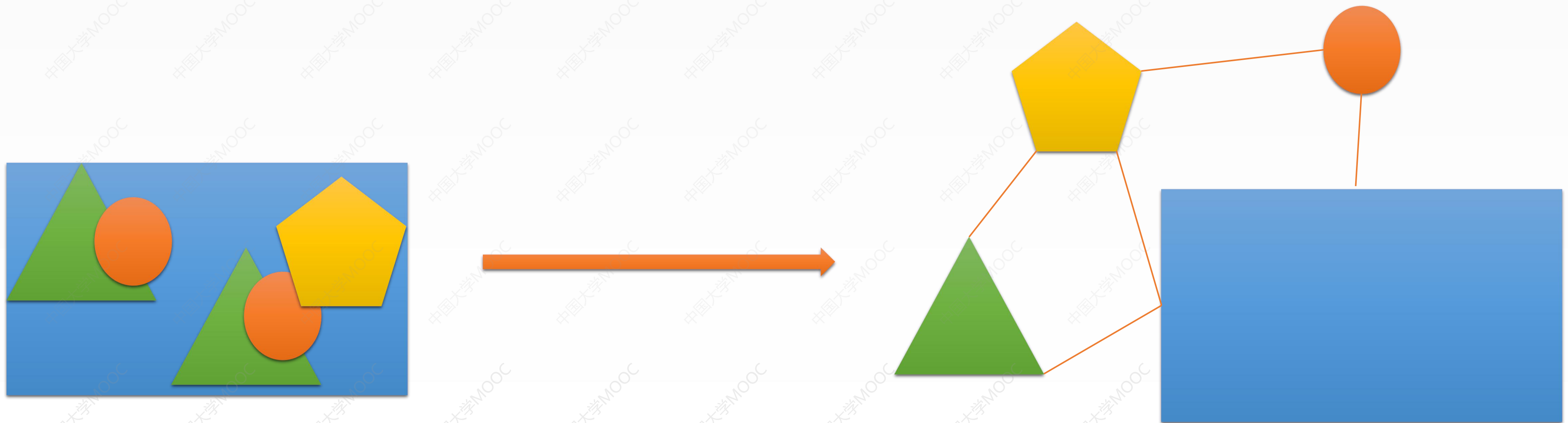


函数的**返回值**



函数的目的

- 函数是面向过程的体现：输入(参数) -> 处理流程 -> 输出(返回值)
- 目的：将重复发生的过程进行统一描述，实现代码复用与解耦
 - **代码复用**：避免重复写相同代码
 - **解耦**：避免发生改动时牵一发而动全身



函数声明与定义

函数的定义明确了函数：

1. 接受什么参数
2. 返回什么值
3. 具体进行什么操作（实现）

```
int addTwoInt(int a, int b) {  
    int c = a + b;  
    return c;  
}
```

函数的声明明确了函数：

1. 接受什么参数
2. 返回什么值

指明返回值类型

↑
`int addTwoInt(int a, int b);`
↓

指明参数列表

函数形参与实参

- **形式参数**：书写函数定义时用来描述函数体的变量
- **实际参数**：调用函数时，外部调用方实际传递给函数的变量

```
int addTwoInt(int a, int b) {  
    return a + b;  
}  
  
int m = 10, n = 20;  
cout << addTwoInt(m, n);
```


函数值传参与指针传参

- 值传参与指针传参实际上没有区别!
- Why? 回顾一个指针变量的值是什么? 是一个地址!
- 指针传参实际就是**指针的值传参**

// 交换两变量的值

```
void swap(int *pa, int *pb){
```

```
    int temp = *pa;
```

```
    *pa = *pb;
```

```
    *pb = temp;
```

```
}
```

```
int m = 10, n = 20;
```

```
int *pm = &m, *pn = &n;
```

```
swap(pm, pn);
```

实际上就是 **pm** 和 **pn** 对 **pa** 和 **pb** 的值传参(拷贝)

全因**指针的值是地址**而让指针传参显得特殊...

复习函数

函数默认参数

引用与引用传参

函数重载

内联函数 (inline)

默认值(default)的概念

- 回顾：计算机程序必要因素之**确定性**
- 程序不怕做错事，怕无法确定自己到底要做什么
- 默认值：告诉程序当数据/指令缺失时，默认用什么来代替，避免失去确定性

```
int power(int n, int x = 2);  
  
int power(int n, int x) {  
    int ans = 1;  
    for (int i = 0; i < x; i++) {  
        ans *= n;  
    }  
    return ans;  
}  
  
int main() {  
    cout << power(5);  
}
```

→ 没给形参x传实参，可以吗？

函数默认参数

默认参数必须在形参列表的结尾！避免歧义

必须在函数**声明**中声明默认参数！

函数声明就是函数的**身份证**

外部调用方不看定义只看声明

```
int power(int n, int x = 2);

int power(int n, int x) {
    int ans = 1;
    for (int i = 0; i < x; i++) {
        ans *= n;
    }
    return ans;
}

int main() {
    cout << power(5);
    cout << power(4, 3);
}
```

没传实参 -> 用默认值

传了实参 -> 用传进来的

复习函数

函数默认参数

引用与引用传参

函数重载

内联函数 (inline)

引用：其实就是别名

- 百变的我？

- 学生口中的 “小谢老师”
- 同事口中的 “Harold”
- 父母口中的 “小宝”
- 好友口中的 “大头”
- ...

其实指的都是同一个实体：我

- 这些别名都在 “引用” 我

- 引用：**同一个实体**(变量)的**别名**

引用：其实就是别名

已定义的

- 特指**左值引用**，即给一个已经有名字的变量起别名，所以不可存在**空引用**！

```
int main() {
```

```
    int a = 10;
```

```
    int &b = a, &c = b;
```

```
    cout << b;
```

```
    c++;
```

```
    cout << a;
```

```
}
```

应当理解为一个整体，即变量b和c的类型是 “**int &**”
中文名为 “**整型引用**”

输出10

输出11

因为c是a的一个引用（“别名”）c自增就是a自增

函数的引用传参

- 思考：指针传参解决了什么问题？
 - 避免按值传参发生的拷贝，实现了**原地**改动调用方传入参数的功能
- 指针传参还有什么问题？
 - 代码里夹杂着间接引用符号 '*' 难写难看难读，还有符号优先级的问题！
- 这个问题怎么解决？
 - 如果形参就是实参的一个别名，岂不是...
- 这个方法就是**引用传参**

```
void swap(int& a, int& b){  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int main() {  
    int m = 1, n = 99;  
    swap(m, n);  
    cout << m << '\t' << n;  
}
```

输出： 99

1



复习函数

函数默认参数

引用与引用传参

函数重载

内联函数 (inline)

over: 重复

load: 装载

函数重载 **overload**

- 思考：函数解决了什么问题？
 - 避免重复书写相同的过程/流程代码，实现代码复用和解耦
- 函数还有什么问题？
 - C++是强类型语言，同一套操作若要用在**不同类型**/数量的参数上，则需要编写**不同函数**！
- 所以呢？
 - 调用方需针对不同的实参写不同的函数调用代码，if-else增多，需判断类型信息
 - 函数名称不同，有多少种参数列表就需要多少个函数名
 - ...
- 解决思路：允许多个**同名函数**存在，分别处理不同类型/数量的参数...
- 这就是**函数重载**

函数重载 overload

- 多个函数的**名字**相同，**参数列表(数量、类型)不同**

```
int add(int a, int b) {  
    return a + b;  
}
```

```
int add(int a, int b, int c) {  
    return a + b + c;  
}
```

```
int add(int a, int b, int c, int d) {  
    return a + b + c + d;  
}
```

```
cout << add(1, 2);  
cout << add(1, 2, 3);  
cout << add(1, 2, 3, 4);
```



都能正确运行

编译器会根据实参类型/数量

自动**匹配**调用哪个函数

函数重载 overload

- 多个函数的**名字**相同，**参数列表(数量、类型)**不同

```
string myAdd(int a, int b) {  
    return std::to_string(a) + std::to_string(b);  
}  
  
string myAdd(int a, string s) {  
    return std::to_string(a) + s;  
}  
  
string myAdd(string s, int a) {  
    return s + std::to_string(a);  
}  
  
string myAdd(string s1, string s2) {  
    return s1 + s2;  
}
```

```
cout << myAdd(111, 999);  
cout << myAdd(111, "abc");  
cout << myAdd("abc", 111);  
cout << myAdd("abc", "def");
```



都能正确运行

编译器会根据实参类型/数量

自动**匹配**调用哪个函数

over: 重复

load: 装载

函数重载 **overload**

- 所以overload解决了什么问题?
 - 减少了函数调用方的代码冗余，现在调用方对不同类型/数量的实参可以写完全一样的代码了！
- 别激动，overload也存在问题
 - 最重要的一环：通过调用时的实参列表，和多个重载函数中的一个进行**匹配**
 - 匹配是编译器自动完成的，但如果你的调用代码有可能产生歧义...

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

```
int m = 10, n = 20;  
cout << func(m, n);
```


避免 overload 歧义

- 如果调用时实参能匹配多个(> 1个)重载函数, 则编译器遇到歧义, 产生编译错误

```
int func(int a, int b) {  
    return a + b;  
}  
  
int func(int& a, int& b) {  
    return a + b;  
}  
  
int func(int a, int b, int c = 2) {  
    return a + b + c;  
}  
  
string func(int a, int b) {  
    return std::to_string(a) + std::to_string(b);  
}
```

```
int m = 10, n = 20;  
cout << func(m, n);
```

编译器: “好像都可以呢TT”

```
<< func(m, n);
```

func
还有 3 个重载
[联机搜索](#)

有多个重载函数 "func" 实例与参数列表匹配:
函数 "func(int a, int b)" (已声明 所在行数:23)
函数 "func(int &a, int &b)" (已声明 所在行数:27)
函数 "func(int a, int b, int c = 2)" (已声明 所在行数:44)
参数类型为: (int, int)

[联机搜索](#)

避免 overload 歧义

- 不允许**仅有返回值不同**的函数重载：重载是针对参数列表的！

```
int func(int a, int b) {  
    return a + b;  
}  
  
int func(int& a, int& b) {  
    return a + b;  
}  
  
int func(int a, int b, int c = 2) {  
    return a + b + c;  
}  
  
string func(int a, int b) {  
    return std::to_string(a) + std::to_string(b);  
}
```

→ overload和返回值无关，只要满足：
声明时：①名字相同 ②参数列表不同
调用时：③不产生匹配歧义

复习函数

函数默认参数

引用与引用传参

函数重载

内联函数 (inline)

“额外开销”的概念

- 一位学生的每日计划:

- 8:00 - 9:00 坐公车去图书馆 1小时
- 9:00 - 11:00 学习 2小时
- 11:00 - 12:00 坐公车回家 1小时
- 12:00 - 12:30 午餐 0.5小时
- 12:30 - 13:30 坐公车去图书馆 1小时
- 13:30 - 16:30 学习 3小时
- 16:30 - 17:30 坐公车回家 1小时
- 17:30 - 18:00 晚饭 0.5小时
- 18:00 - 19:00 坐公车去图书馆 1小时
- 19:00 - 21:00 学习 2小时
- 21:00 - 22:00 坐公车回家 1小时

他勤奋吗?
不可谓不勤奋。
但科学吗?

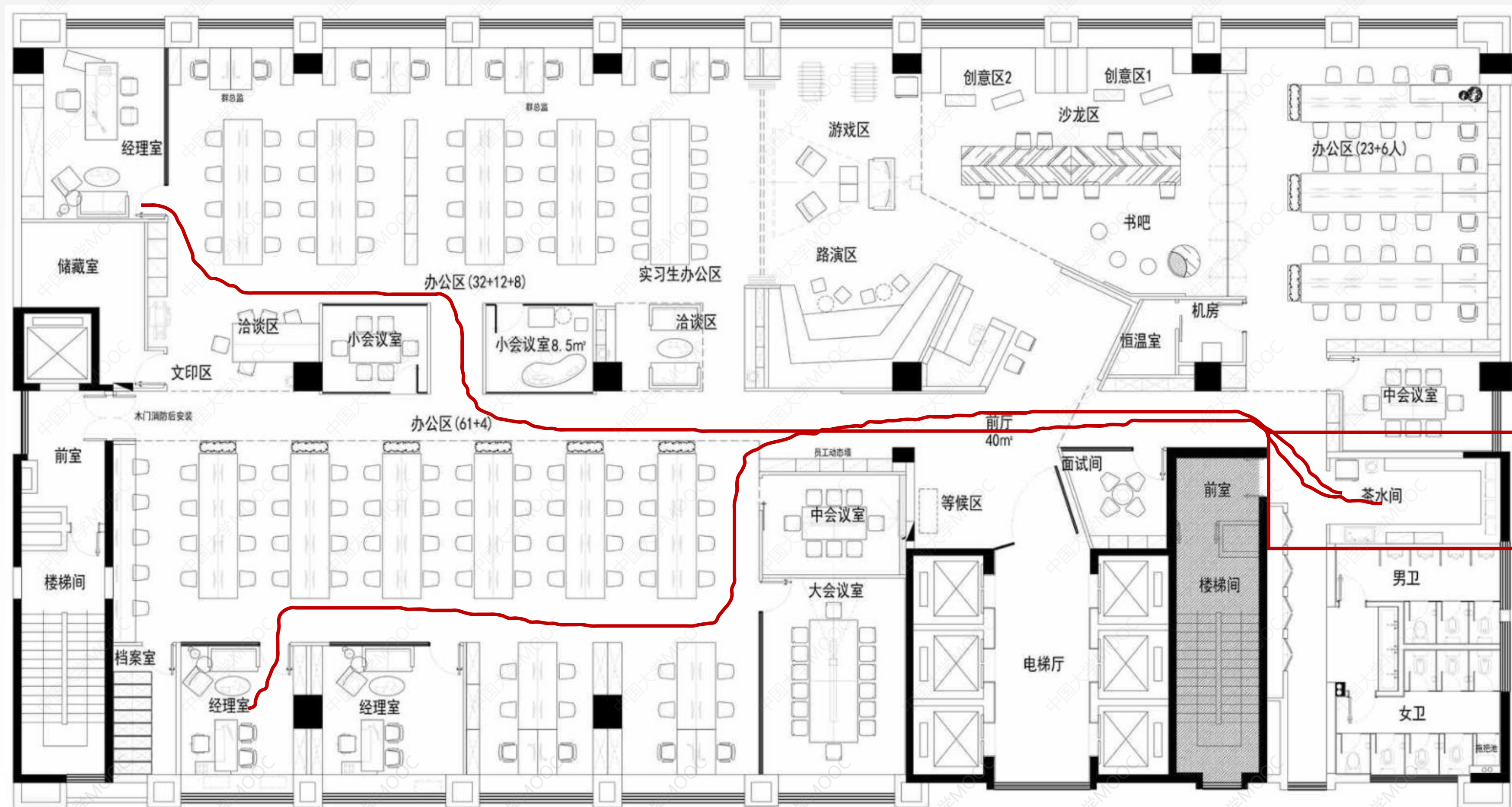
为学习做**准备/善后**工作的时间
花给学习的总时间 = $\frac{6\text{小时}}{(6 + 7)\text{小时}} = 46\%$

诸位学习时是否也有大量的
准备时间、磨蹭时间、分心时间?

“额外开销”的概念

“咱有钱了一定**每个**办公室配一台饮水机。”

- 一家公司平面图：



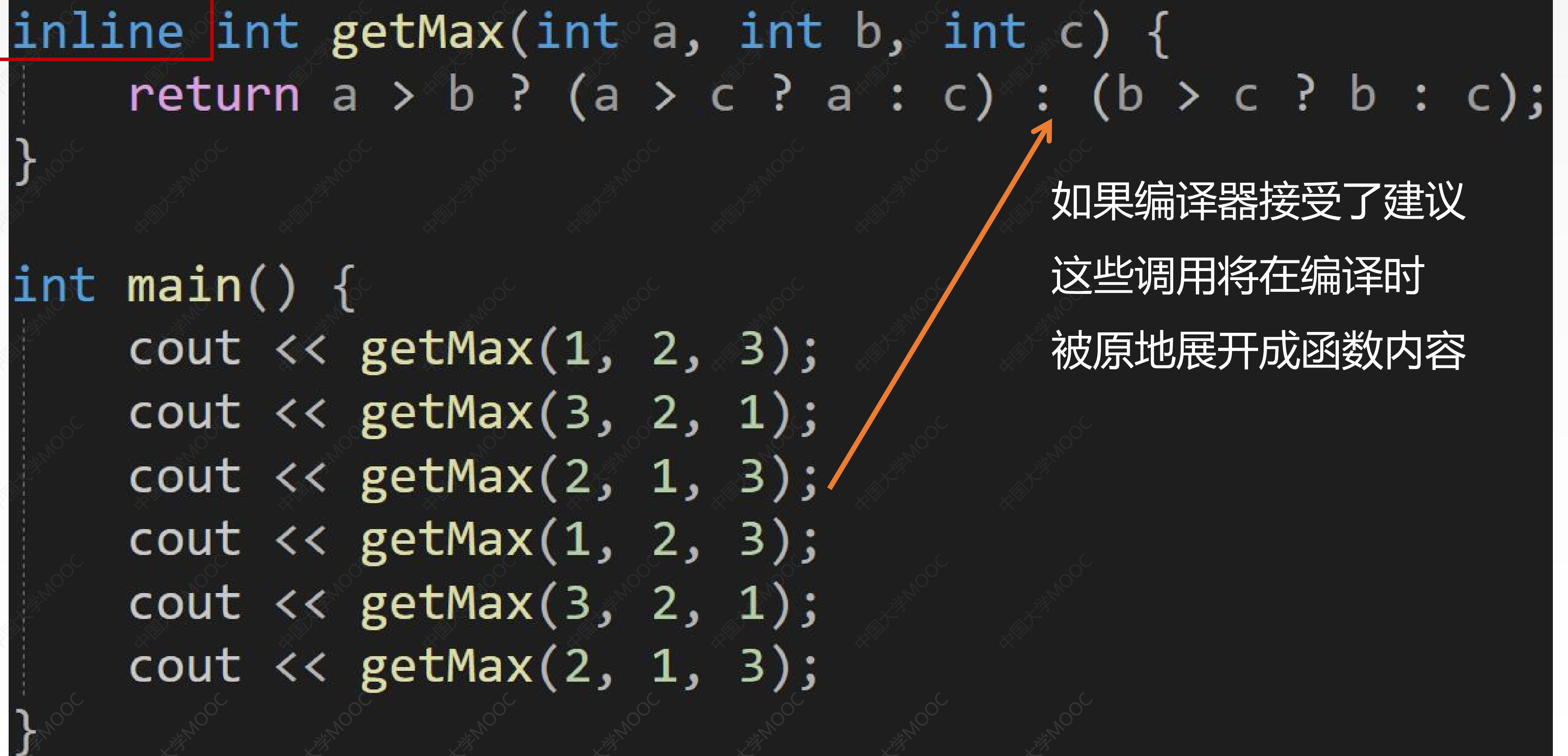
内联(inline)函数

- 函数的目的：将重复发生的流程统一起来，实现代码复用和解耦
- 但是！如果某个函数的功能**非常简单**，又被**反复调用**的话，存在什么问题？
 - 提示：“函数调用”这个行为本身也是有一定开销的（调用栈）
- 那么调用这个函数的“额外开销”占比就很大了
- 此时可以**建议**编译器在编译时将函数直接在调用处**展开**，避免函数调用行为的额外开销
- 这就是**函数内联**

本想饮水机复用 → 结果走去茶水间的开销大于接一杯水的收益 → 不如每个办公室配一台饮水机

内联(inline)函数

- 内联函数：指**建议**编译器编译时将某个函数在**调用处**直接**展开**，避免运行时调用开销

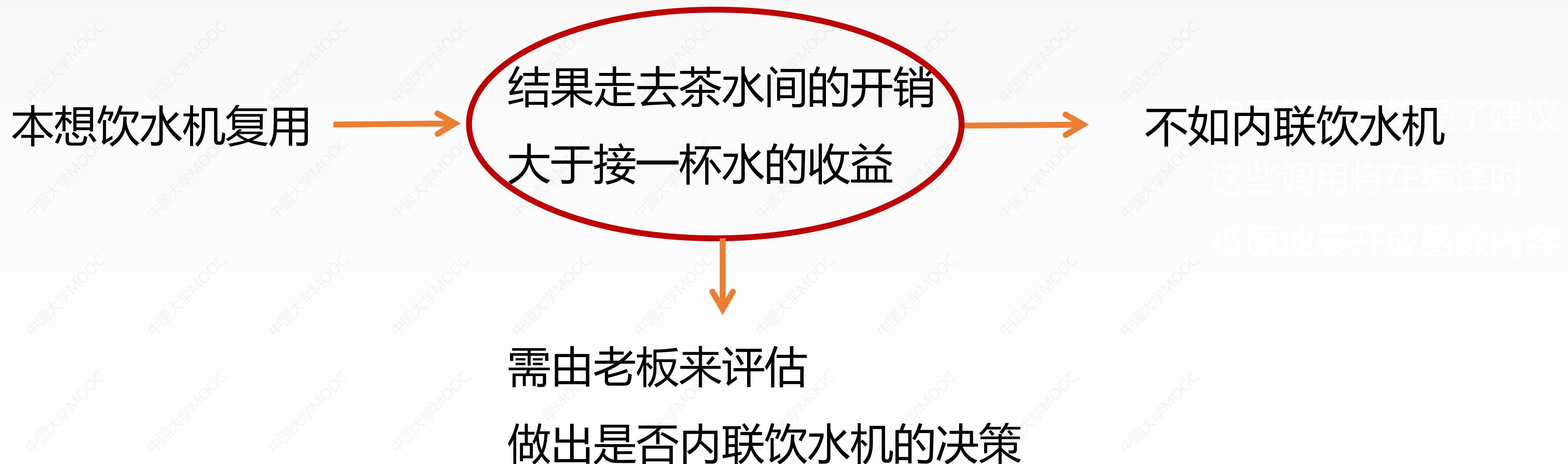


```
inline int getMax(int a, int b, int c) {  
    return a > b ? (a > c ? a : c) : (b > c ? b : c);  
}  
  
int main() {  
    cout << getMax(1, 2, 3);  
    cout << getMax(3, 2, 1);  
    cout << getMax(2, 1, 3);  
    cout << getMax(1, 2, 3);  
    cout << getMax(3, 2, 1);  
    cout << getMax(2, 1, 3);  
}
```

如果编译器接受了建议
这些调用将在编译时
被原地展开成函数内容

inline只是建议

- 并不是写了 inline 关键字就一定会被内联，只是提出**建议**，由编译器决定是否采纳
- 内联这个动作发生在编译时，提升运行时的效率



★ **Key 1**: 为了避免歧义，默认参数应当放在形参列表的**最后面**

★ **Key 2**: 默认参数应当在函数声明里设置

1. 在C++中，下列关于函数参数默认值的描述中正确的是：**C**

- A. 设置参数默认值时，应当全部设置
- B. 设置参数默认值后，调用函数不能再对参数赋值
- C. 设置参数默认值时，应当从右向左设置
- D. 只能在函数定义时设置参数默认值

2. 以下代码中，编写带默认参数的函数正确的是：**B**

- | | |
|--|--|
| A. <code>int func(int a, int b = 2);</code> | B. <code>int func(int a, int b = 2) {...}</code> |
| <code>int func(int a, int b = 1) {...}</code> | |
| C. <code>int func(int a, int b = 2, int c);</code> | C. <code>int func(int a, int b);</code> |
| <code>int func(int a, int b, int c) {...}</code> | <code>int func(int a, int b = 2) {...}</code> |

分析：函数声明对外表明了函数的名字、返回值与参数列表，它就是函数的身份证。B选项是声明同时定义。

★ **Key 3**: 引用的本质是已定义变量的别名，因此不可存在空引用

★ **Key 4**: 除函数形参外，其他引用定义时必须赋初始值

1. 以下程序段中a和b两处空行应分别填入：**D**

```
void swap(____a____) {  
    temp = a;  
    a = b;  
    b = temp;  
}  
  
// ... 省略
```

```
int m = 10, n = 20;  
swap(____b____);
```

类型& 变量名 是定义一个该类型的引用
&变量 是对变量取地址，得到一个指针

A. int a, int b 和 m, n

C. int *a, int *b 和 &m, &n

B. int &a, int &b 和 &m, &n

D. int &a, int &b 和 m, n

★ **Key 5**: 函数重载三要素: ①名称相同 ②参数列表不同 ③调用不产生匹配歧义

★ **Key 6**: 仅有返回值不同不能构成重载!

1. 函数重载的目的是: **B**

- A. 减少函数提供方的代码冗余
- B. 方便调用方编写代码, 提高可读性
- C. 减少程序运行时的内存占用
- D. 提高程序运行效率

2. 以下哪项不能与 `int func(int, int)` 构成重载函数: **C**

- A. `int func(int&, int&);`
- B. `int func(int, int, int);`
- C. `string func(int, int);`
- D. `string func(int, string);`

★ Key 7: 若一函数功能简单, 则函数调用的额外开销占比较高

★ Key 8: inline 关键字只是**建议**编译器将函数内联, 是否内联由编译器自行决定

★ Key 9: 函数内联发生在编译时, 提高的是运行时效率

1. 下列哪个类型函数不适合声明为内联函数: **A**

- A. 函数体语句较多
- B. 函数体语句较少
- C. 函数执行时间较长
- D. 函数执行时间较短。

2. 在内联函数内允许使用的是: **D**

- A. if-else 语句
- B. switch语句
- C. 赋值语句
- D. 以上都允许

3. 关于下列函数, 说法正确的是: **D**

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

- A. 将在预编译阶段进行内联展开
- B. 将在编译时进行内联展开
- C. 将在运行时进行内联展开
- D. 不确定其是否会进行内联展开



中国大学MOOC
搜索: C++不挂科