C++ `do-while` 循环 讲义

1. 引言

1.1 问题引入：书本例 4-5 猜数游戏

想象一下，让机器人在心里想一个1到100之间的整数，然后让大家来猜。

- 你每猜一个数，会给你提示：“太大了 (Too large)” 或 “太小了 (Too small)”。
- 如果你猜中了，它会说：“猜对了 (You are right!!)”。
- 只要没猜对，你就要一直猜下去。

这个“一直猜下去”的过程，在编程中应该如何实现呢？

使用我们**循环**！但用我们之前学过的 `while` 循环，会遇到一个小小的问题。

`while` 循环的特点是“**先判断，后执行**”。它的逻辑是：
`while (猜的数 != 答案) { // 让你继续猜 }`

这里问题就来了：在第一次进入循环，准备进行判断时，你**还根本没有猜过任何数字**！`猜的数` 这个变量里没有有效的值，这个判断也就失去了意义。

这就引出了我们今天要学习的 `do-while` 循环。它专门用来解决这类“**无论如何，都得先做一次**”的问题。除了猜数游戏，还有很多类似场景：

- **ATM取款**：你必须先输入一次密码，银行系统才能去验证它是否正确。
- **游戏菜单**：程序至少要先显示一次菜单（开始游戏、加载存档、退出），你才能进行选择。
- **输入验证**：程序至少要先让你输入一次，才能判断你的输入是否合规。

---

2. 核心思想

2.1 `while` 循环

我们先尝试用 `while` 循环来解决猜数问题，看看会发生什么。

```
int main() {
    int ans = 42; // 假设答案是42
    int guess;    // 准备存放你猜的数

    // 问题：第一次判断时，guess 是多少？
    while (guess != ans) {
        cout << "请输入你猜的数字: ";
        cin >> guess;
        // ...后续判断大小的逻辑...
    }
    return 0;
}
```

在进入 `while` 的那一刻，`guess` 变量没有被初始化，它里面是一个不确定的值。用一个不确定的值去和 `ans` 比较，结果是完全不可预测的。

为了让 `while` 循环能跑起来，我们需要做一些妥协，比如在循环外先写一次输入代码，像这样：

```
// ...
int ans = 42;
int guess;

cout << "请输入你猜的数字: ";
cin >> guess; // 在循环外先猜一次

while (guess != ans) {
    // ...判断大小的逻辑...
    cout << "请再猜一次: ";
    cin >> guess; // 在循环内又写了一遍
}
// ...
```

但是这样做的话，`cin >> guess;` 这句代码重复了，对于这种先执行，后判断的流程，C++有更加优雅的方式完成。

2.2 `do-while`

`do-while` 循环就是“**先执行，后判断**”。它保证循环体内的代码**至少会被执行一次。**

它的语法结构如下：

```
do {
    // 循环体
    // 这里的代码保证，无论如何都会先被执行一次。
} while (循环条件); // 注意这里的分号
```

- `do`：一个关键字，代表循环体开始，并且不管三七二十一，先把 `{}` 里面的代码做一遍！
- `{ ... }`：循环体，和 `while` 循环一样，是需要重复执行的代码。
- `while (循环条件);`：在循环体执行**之后**，才在这里进行条件判断。如果条件为真，就跳回到 `do` 的位置，开始下一次循环；如果为假，则循环结束，继续执行后面的代码。

2.3 步骤拆解

`do-while` 的执行流程我们可以拆解为以下几步：

1. **无条件执行**：程序遇到 `do`，先直接进入 `{}` 执行循环体内的所有语句。
2. **事后判断**：循环体执行完毕后，来到 `while(条件)` 这一行，计算括号内条件表达式的真假。
3. **决策**：
    - 如果条件为 **真 (true)**，程序跳转回 `do` 关键字处，准备开始新一轮的循环（重复步骤1和2）。
    - 如果条件为 **假 (false)**，循环正式结束。程序会越过这个循环结构，去执行 `while(...);` 后面的代码。

---

3. 实例

3.1 具体案例

假设小洛机器人心里想的数字是 **42**，那么我们就可以通过`do-while`写出如下的代码。

代码片段:

```
int ans = 42;
int guess;
do {
    cout << "猜一个1-100的数: ";
    cin >> guess;
    if (guess > ans) { cout << "太大了!\n"; }
    if (guess < ans) { cout << "太小了!\n"; }
} while (guess != ans);

cout << "猜对了!!\n";
```

---

4. 代码实现

现在，我们来完成一个功能完整的“猜数游戏”。输入数字，来判断玩家猜测的数字是否正确，并且给出提示。

注意在真实的游戏中，答案不应该是固定的，而应该是一个随机数。

4.1 完整代码

```
#include <iostream>
#include <cstdlib> // 用于 rand() 和 srand()
#include <ctime>   // 用于 time()

int main() {
    // 设置随机数种子
    // 使用当前时间作为种子，确保每次运行程序时随机数都不同
    srand(time(0));

    // 生成一个 1 到 100 之间的随机数
    // rand() % 100 的范围是 [0, 99]，加1后范围变为 [1, 100]
    int ans = rand() % 100 + 1;

    int guess; // 用来存储玩家猜测的数字


    // 3. 使用 do-while 循环让玩家猜测
    do {
        cin >> guess;

        if (guess > ans) {
            cout << "太大了!\n";
        } else if (guess < ans) {
            cout << "太小了!\n";
        }

    } while (guess != ans); // 只要猜的数不等于答案，就继续循环
```

```
    // 4. 循环结束，说明猜对了
    cout << "恭喜你，猜对了!\n";

    return 0;
}
```

4.2 代码解读

- `#include <cstdlib>` 和 `<ctime>`：这两个头文件是生成“随机数”的关键。`<cstdlib>` 包含了 `rand()`（生成伪随机数）和 `srand()`（种子），`<ctime>` 包含了 `time()`（获取当前时间）。
- `srand(time(0));`：这是非常关键的一行。计算机生成的随机数其实是“伪随机”，它由一个“种子”开始，通过一个固定算法算出一串数。如果种子不变，那串数也永远不变。`time(0)` 能获取从1970年1月1日到现在的秒数，这个数时刻在变，是作为种子的绝佳选择。**这行代码通常在程序开始时只执行一次。**
- `rand() % 100 + 1`：这是一个生成特定范围随机数的经典公式。
  - `rand()` 生成一个从 0 到 `RAND_MAX`（一个很大的数）的整数。
  - `% 100` 对100取余，结果一定是 0 到 99 之间。
  - `+ 1` 将 `[0, 99]` 的范围整体平移，得到 `[1, 100]`。
- `do { ... } while (guess != ans);`：整个“输入-判断-提示”的逻辑被包裹在 `do` 的循环体中，确保玩家至少能猜一次。然后 `while` 在每次猜测后检查是否猜中，决定是继续还是结束。

5. 总结

在什么时候用 `do-while`，什么时候用 `while`？

- 标准就是：**循环体是否需要无条件地至少执行一次？**
  - 如果**是**，比如菜单显示、输入验证等，`do-while` 是最佳选择。
  - 如果**否**，也就是可能一次都不需要执行，那么 `while` 更合适。

对比：`while` vs `do-while`

特性	`while` 循环	`do-while` 循环
核心原则	**先判断，后执行**	**先执行，后判断**
执行保证	可能执行 **0** 次或多次	**至少**执行 **1** 次
语法结构	`while (条件表达式) { ... }`	`do { ... } while (条件表达式);`