C++ `while` 循环 讲义

1. 课程引言

1.1 问题引入

任务：让计算机在屏幕上换行输出 1、2、3、4、5。
很简单，我们利用**顺序结构**很快就能写出这样的代码：

```
int main() {
    cout << 1 << endl;
    cout << 2 << endl;
    /* ... 略 ... */
    cout << 5 << endl;
    return 0;
}
```

“So easy!” 很多同学可能会这么想。但是，请看下一个任务：

升级版任务：让计算机换行输出 1 到 100 之间的每一个整数。
这时候，如果继续使用顺序结构，我们需要把 `cout` 语句复制粘贴 100 遍。如果要求是输出 1 到 10000 呢？这种简单、重复、有规律的工作，人类来做既枯燥又易错。

核心观点：计算机非常适合且极其擅长做大规模重复的事情！ 为了“解放”我们，我们需要一种新的程序结构——**循环结构**。

---

2. 核心思想：什么是 while 循环?

循环，顾名思义，就是让一段代码能够“转着圈”地反复执行。`while` 循环是其中最基础、最直观的一种。

2.1 生活中的 while 逻辑

大家对这句话一定不陌生，妈妈在你写作业前说：

“只要（while）你还没完成所有作业（循环条件），你就必须一直写（循环体）。”

这句话包含了 `while` 循环的核心要素：

- **循环条件**：还没完成所有作业。
- **循环体（重复的动作）**：一直写作业。

你每写完一门作业，都会检查一次条件（作业写完了吗？）。如果没有，继续写；直到条件不再成立（作业写完了），循环结束，你可以出去玩了。

2.2 if 与 while 的区别

- `if` **语句（分支）**：像一个岔路口，判断一次条件，满足就做，做完就接着往下走。
- `while` **语句（循环）**：在执行完“做某事”之后，会**返回到原点再次判断条件**，形成一个圈。只要条件为真，这个圈就会一直转下去。

2.3 while 循环的执行流程(重点)

```
while (条件表达式) {
    // 循环体：重复执行的代码
}
```

1. **第一步（判断）**：计算 `条件表达式` 的值。
2. **第二步（决策）**：若为 `true`（真），进入大括号执行代码；若为 `false`（假），直接跳过整个 `while` 结构。
3. **第三步（执行）**：从上到下执行循环体内的所有语句。
4. **第四步（返回）**：执行完毕后，**自动返回第一步**，重新判断条件。

---

3. 代码实战：编写 while 循环的模板

如何编写一个正确的 `while` 循环：

- **第一步：初始化循环控制变量。** 在循环开始前，设立一个“标记变量”（如计数器）。
- **第二步：设置循环继续的条件。** 写下让循环“继续下去”的规则。
- **第三步：更新控制变量。** 在循环体内部改变“标记变量”的值，使其朝着“不满足条件”的方向发展。

3.1 解决 输出 1 到 100 的任务

```
int main() {
    // 初始化：创建一个循环控制变量 i，让它从 1 开始
    int i = 1;

    // 条件：只要 i 小于或等于 100，就继续循环
    while (i <= 100) {
        // 循环体
        cout << i << endl; // 重复的任务：打印当前数字

        // 更新：让 i 的值加 1，为下一轮判断做准备
        i++;
    }
    return 0;
}
```

---

4. 案例剖析

案例一：乘地铁回家

场景：小六乘坐地铁回家，世纪大道站上车，梦站下车。他想知道途经了多少站。
逻辑梳理：

- **初始化**：上车，计数器设为 0。
- **条件**：`while (还没有到梦站)`
- **循环体**：再坐一站路；计数器 +1。

- **结束**：下车，输出计数器。

代码思路：

```
上车;
计数器设置为0;
while( 没有到梦站 ){
    再做一站路;
    计数器+1;
}
下车;
输出计数器;
```

案例二：一尺之棰（数学计算 - 书本 P51 例 4-4）

题目：《庄子》曰：“一尺之棰，日取其半，万世不竭”。输入木棍初始长度 a，每天锯掉一半（向下取整），求第几天木棍长度会变为 1？
分析：这题很适合 `while` 循环，因为我们**不知道具体要循环多少次**，我们只知道**循环结束的条件。**

```cpp
int a;
cin >> a;
int day = 1; // 第一步：初始化天数

while (a > 1) {  // 第二步：循环条件，只要长度还大于1就继续砍
    a = a / 2;   // 第三步：更新变量1（木棍变短）
    day++;       // 第三步：更新变量2（天数增加）
}

cout << day << endl; // 循环结束后输出结果
```

如果缺了 `a = a / 2;`，木棍永远那么长，永远砍不完；如果缺了 `day++;`，时间就静止了天数永远都是1。

案例三：累加

3.1 题目描述

编写一个程序，让用户输入一个正整数 `n`，然后计算 `s = 1 + 2 + 3 + … + n` 的值，并输出最终结果 `s`。

输入/输出样例：

- 如果用户输入 `3`，程序应该计算 `1 + 2 + 3`，最终输出 `6`。
- 如果用户输入 `5`，程序应该计算 `1 + 2 + 3 + 4 + 5`，最终输出 `15`。
- 如果用户输入 `100`，程序应该计算 `1 + 2 + ... + 100`，最终输出 `5050`。

3.2 思路分析

这个问题我们无法用一行代码直接得出答案。我们先想一想，如果不用程序，我们的大脑是如何计算 `1 + 2 + 3 + 4 + 5` 的？

过程很可能是这样的：

1. 先算 `1 + 2`，在脑中记下结果 `3`。
2. 再用刚才记下的 `3`，去加上 `3`，得到 `6`，并用这个新结果 `6` 更新我们脑中的记忆。

3. 接着，用记下的 `6`，去加上 `4`，得到 `10`，再次更新记忆。
4. 最后，用记下的 `10`，去加上 `5`，得到最终答案 `15`。

在这个过程中关键的一点是：我们的大脑里始终**“记着”**一个**“当前的和”**。这个“当前的和”在计算过程中是不断变化的、滚雪球式增长的。

程序也需要这样一个能“记事儿”的角色。在编程中，我们给它起了一个非常形象的名字——**累加器**。

3.3 累加器的运作流程

基于上面的分析，我们可以把核心思路拆解成以下几个步骤：

1. **准备一个“累加器”**：我们需要一个整型变量，专门用来存放累加的和。我们叫它 `sum`。在计算开始前，这个“和”是空的，所以 `sum` 的初始值必须是 **0**。
2. **准备一个“计数器”**：我们需要另一个变量，依次代表我们要加的数：1, 2, 3, ..., n。这个我们很熟悉了，就是循环控制变量，我们叫它 `i`。它应该从 `1` 开始。
3. **建立循环**：我们需要一个 `while` 循环，来驱动计数器 `i` 从 `1` 稳定地增长到 `n`。循环的条件自然就是 `i <= n`。
4. **执行累加动作**：在循环的每一次中，我们都执行最核心的累加动作：`sum = sum + i;`。这行代码的含义是：取出 `sum` 的当前值，和 `i` 的当前值相加，得到的**新和**再存回给 `sum` 变量自己，完成一次自我更新。
5. **更新计数器**：在累加动作之后，要让 `i` 的值加1（`i++;`），这样下次循环才能累加下一个数。

这个过程就像滚雪球，`sum` 是那个雪球，`i` 是每一捧新雪。循环一次，雪球就裹上一层新雪，变得更大。当 `i` 遍历完所有数，雪球的最终大小就是我们所求的总和。

3.4 代码实现与解读

```
int main() {
    // 让用户输入目标数字 n
    int n;
    cin >> n;

    // 初始化
    int sum = 0; // 累加器 sum 初始值为 0
    int i = 1;   // 计数器 i 从第一个要加的数 1 开始

    // 循环条件：只要 i 还没有超过 n，就继续累加
    while (i <= n) {
        // 将当前的数 i 累加到 sum 中
        sum = sum + i;  // 也可以简写为: sum += i;
        // 让 i 增加 1，准备加下一个数
        i++;
    }

    // 循环结束后，sum 中存放的就是 1 到 n 的最终累加结果
    cout << sum << endl;
    return 0;
}
```

代码解读：

- `int sum = 0;`：如果 `sum` 没有被初始化，它的值将是随机的，最终结果自然也是错误的。而 `0` 是加法运算的起点。
- `while (i <= n)`：循环的边界条件是 `i <= n`，确保从 `1` 到 `n`（包括 `n` 本身）的每一个数都被加进了 `sum` 中。
- `sum = sum + i;`：它读取 `sum` 的旧值，进行计算，然后用新值覆盖旧值。
- `cout << sum << ...;`：这行代码位于 `while` 循环的**外部**。因为我们只关心最终的那个总和，而不是中间过程的每一个“当前和”。如果你把这行代码错误地放进了循环体内部，会发生什么呢？可以动手试一试，区别一下“循环内外”到底有什么不同。

---

5. 深化理解

5.1 死循环

如果我们在输出 1 到 100 的代码中漏掉了 `i++;` 会怎样？

```
int i = 1;
while (i <= 100) {
    cout << "救命！" << endl;
}
```

运行结果：`i` 的值永远是 1，`1 <= 100` 永远成立。屏幕会被“救命！”一直刷屏，程序永不停止。
所以务必确保循环体内部有代码能让循环条件最终变为 `false`！

5.2 C++ 中的“真”与“假”

在 C++ 中，**0 为假（`false`），其他所有非 0 数都为真（`true`）。**

- `if (0)` 或 `while (0)`：条件为假，里面的代码**绝对不会**执行。
- `while (1)` 或 `while (999)`：条件永远为真，这是一个标准的无限循环（死循环）写法，通常需要配合内部的 `break` 语句来跳出（后续课程学习）。

5.3 如何实现循环 n 次？

我们要输出 n 个 `*`，有许多种写法（例如从 n 递减到 1），但最通用、最标准的“约定”是：**从 0 开始，循环 n 次。**

```
int n;
cin >> n;

int i = 0;          // 从 0 开始
while (i < n) {     // 条件为小于 n（注意不是小于等于）
    cout << "*" << endl;
    i++;
}
```

为什么推荐这种写法？

1. **与数组契合**：C++ 中的数组索引是从 0 开始的，习惯 `0` 到 `n-1` 的模式，未来学习数组时可以进行衔接。

2. **逻辑清晰**：`[0, n)` 是一个左闭右开的区间（从 `0~n-1` ），区间内元素的个数正好是 n，不多也不少。

---

6. 遍历 + 筛选 模式

掌握了基础循环，我们来看一个解决大量编程问题的核心基础框架：**遍历与筛选。**

任务：输入区间起点 $n1$ 和终点 $n2$，打印出这个区间内的**所有偶数。**

用一个生活中的例子想象一下，你现在推着购物车在超市的货架通道里走。你的任务是：**“在第 1 排到第 10 排货架之间，把所有打半价的商品挑出来。”**

在这个场景中：

- **货架通道的范围（第 1 排到第 10 排）**：这就是我们的**区间**（$n1$ 到 $n2$）。
- **你顺着通道一排排走过，目光扫过每一件商品**：这个动作就叫做【遍历】。不管买不买，你都必须逐一“看”过去，不能漏掉。在代码中，这就是 `while (i <= n2)` 和不断向前看的 `i++`，而你当前正在看的这个物品，就相当于是循环变量 `i`。
- **你拿起一件商品，看一眼标签上的价格**：这个动作叫做【筛选】。如果是半价，放进购物车（执行操作）；如果不是半价，放回原处（什么都不做）。在代码中，这就是 `if` 判断。

```
int n1, n2;
cin >> n1 >> n2;

int i = n1; // 初始化循环变量 i，因为 n1 是起点，故初始化为 n1 让循环变量从起点开始

while (i <= n2) { // 步骤1：【遍历】只要循环变量 i 没越过终点，就挨个检查

    if (i % 2 == 0) { // 步骤2：【筛选】判断当前数字是否为偶数
        cout << i << endl; // 满足条件才处理（输出）
    }

    i++; // 步骤3：更新，无论是不是偶数，都要前进到下一个数字
}
```

为什么说它是“解决大量编程问题的核心基础框架”？因为掌握了这个模式，你只需要修改 `if` 里面的**鉴定条件**，就能解决成百上千种不同的问题，而外层的 `while` 遍历框架**完全不需要改动**！

如果把刚才代码里的 `if` 条件换一下，我们可以实现：

- **找奇数**：`if (i % 2 != 0)` （除以 2 余数不为 0）
- **找 5 的倍数**：`if (i % 5 == 0)`
- **找个位数是 7 的数字**：`if (i % 10 == 7)` （任何正整数对 10 取余，结果就是它的个位数）

遍历 + 筛选 模式思路步骤：

1. **遍历**：使用 `while` 循环，借助自增循环变量 `i`，挨个访问想要查找区间内的每一个元素。
2. **筛选**：在循环体内部，嵌套一个 `if` 条件语句，提取出我们感兴趣的数据（如偶数、素数、最大值等）。