C++ `switch` 多选择语句 讲义

1. 引言

- **问题引入：多分支选择的困境**

  在之前的学习中，我们掌握了 `if-else if-else` 结构，它能帮助我们处理各种复杂的判断。例如，设计一个“自动贩猫机”，根据投入的金额，输出对应的猫咪：

  ```
  if (n == 9) {
      cout << "小白";
  } else if (n == 8) {
      cout << "小黑";
  } else if (n == 6) {
      cout << "小棕";
  } else if (n == 4) {
      cout << "小灰";
  } else {
      cout << "无";
  }
  ```

  这段代码功能正确，但当选项非常多时，一长串的 `else if` 会显得有些重复和臃肿。我们不禁会想：对于这种**只针对同一个变量进行等值判断**的场景，有没有更清晰、更结构化的写法呢？

- **解决方案：`switch` 语句**

  答案是肯定的。C++ 为这类“多路等值选择”场景，提供了一个专门的解决方案——`switch` 语句。它能让我们的代码意图一目了然，极大地提高代码的可读性。

- **学习目标**：

  1. 理解 `switch` 语句与 `if-else if` 在逻辑上的核心区别。
  2. 掌握 `switch`, `case`, `break`, `default` 四个关键字的语法和作用。
  3. 深刻理解并规避 `switch` 语句最常见的陷阱——“case穿透”。
  4. 学会利用 `switch` 的特性合并相似逻辑，编写简洁高效的代码。

---

2. 核心思想

`if-else if` 和 `switch` 的工作方式有着本质的不同：

- **`if-else if` 的逻辑是“顺序比较”**：
  它像一个哨兵，拿着一个值，从第一个条件开始逐一盘问：“你是9吗？”、“你是8吗？”、“你是6吗？”... 直到找到匹配的条件或者走到最后的 `else`。这是一个**串行**的过程。

- **`switch` 的逻辑是“直接跳转”**：
  它更像一个“对号入座”系统。`switch` 会先看一下变量的值（比如是8），然后直接在内部的“情况列表（`case`）”中找到值为8的标签，并**直接跳转**到该位置开始执行代码。这是一个**直接**的过程，在选项很多时通常效率更高。

---

3. 语法详解 (构成与规则)

我们来看一下 `switch` 语句的完整、规范的结构：

```
switch (表达式) {
    case 常量值1:
        // 当'表达式'的值等于'常量值1'时，执行此处的代码
        break;
    case 常量值2:
        // 当'表达式'的值等于'常量值2'时，执行此处的代码
        break;
    // ...可以有任意多个case...
    default:
        // 当'表达式'的值与上面所有case都不匹配时，执行此处的代码
        break;
}
```

- **关键字剖析**：
  - `switch (表达式)`：这里的"表达式"是我们要判断的变量或计算结果，它的值必须是**整数类型**（如 `int`, `char`, `bool` 等）。
  - `case 常量值:`：每个 `case` 都是一个"路牌"或"标签"。`case` 后面必须跟一个**常量**（如 `8`, `'A'`），不能是变量。程序会用 `switch` 表达式的值与这些常量值进行匹配。
  - `break;`：**这是 `switch` 语句的刹车**。当一个 `case` 的代码块执行完毕后，`break` 会立即终止并跳出整个 `switch` 结构。**如果缺少 `break`，将会引发"case穿透"**。
  - `default:`：这是一个**可选**的"默认"或"其他"分支。如果表达式的值与所有 `case` 都不匹配，程序就会执行 `default` 下的代码块。它功能上类似于 `if-else if` 链条中最后的那个 `else`。

---

4. 关键机制与技巧

陷阱："Case穿透"

让我们回到"贩猫机"的例子，如果我们在写 `switch` 时，忘记了写 `break`，会发生什么？

```
// 错误的示例 - 忘记写 break
int n = 8;
switch(n) {
    case 9:
        cout << "小白";
    case 8:
        cout << "小黑"; // 程序会从这里开始执行
    case 6:
        cout << "小棕";
    case 4:
        cout << "小灰";
    default:
        cout << "无";
}
// 最终输出: 小黑小棕小灰无
```

- **执行分析**：
  1. `switch(n)` 发现 `n` 的值是 `8`。

2. 程序直接**跳转**到 `case 8:` 标签处。
3. 执行 `cout << "小黑";`。
4. **由于 `case 8` 后面没有 `break`，程序不会停止！** 它会无视下面的 `case 6:`、`case 4:` 等标签，像一辆失控的汽车一样，继续向下执行所有代码，直到遇到一个 `break` 或者 `switch` 结构的末尾。
5. 因此，它会接着执行 `cout << "小棕";`、`cout << "小灰";` 和 `cout << "无";`。

结论：忘记写 `break` 是 `switch` 语句中最常见的逻辑错误。**请养成写完一个 `case` 的逻辑后，立即补上 `break` 的好习惯。**

合并相似分支：有意利用"穿透"

虽然"穿透"通常是 Bug 的来源，但我们也可以巧妙地利用它来简化代码。当多个不同的 `case` 需要执行完全相同的操作时，我们可以将它们堆叠在一起。

- **案例**：判断输入的字母是元音还是辅音。

```
char letter = 'i';
switch (letter) {
    // 对于 a, e, i, o, u 这五种情况，都执行相同的逻辑
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        cout << "是元音字母" << endl;
        break; // 执行完后跳出
    default:
        cout << "是辅音字母" << endl;
        break;
}
// 输出: 是元音字母
```

在这个例子中，当 `letter` 是 `i` 时，程序跳转到 `case 'i':`，因为前面没有任何 `break`，它会一直"穿透"下去，直到 `case 'u':` 之后，执行 `cout << "是元音字母" << endl;`，然后 `break`。这种写法清晰地表达了"对于这一组情况，执行相同的操作"的意图。

---

5. 使用场景与限制 (`switch` 和 `if-else if`)

`switch` 并非 `if-else if` 的完全替代品，它有自己的使用领域。

特性	`switch` 语句	`if-else if-else` 语句
比较类型	只能进行**等于 (==)** 的比较	可以进行任意逻辑条件的比较 (`>`, `<`, `!=`, `&&`, `||`)
判断变量	必须是**同一个**变量或表达式	每次判断的条件可以完全不同
`case` 值	必须是**整型或字符型常量**	条件中的值可以是变量，类型不限

特性	`switch` 语句	`if-else if-else` 语句
适用场景	**等值判断(就是判断是否与某个值相等)**	**区间判断**、**复杂逻辑组合**、所有通用判断场景
可读性	在处理多个等值选择时，结构更清晰，可读性**更高**	分支过多或条件复杂时，可读性可能下降

简单总结：当你需要判断一个变量**等于**哪个**具体常量值**时，`switch` 是更优的选择。而在进行**范围判断**（如 `score > 90`）或**复合条件判断**时，必须使用 `if-else if`。

---

6. 总结

1. **功能定位**：`switch` 是处理**多路等值选择**的专用结构，比 `if-else if` 更清晰、高效。
2. `break` ：务必为每个独立的 `case` 分支添加 `break`，以防止意外的“case穿透”。
3. **合并** `case`：当多个情况执行相同逻辑时，可以利用“穿透”特性将它们分组，以简化代码。
4. `default` **保底**：总是提供一个 `default` 分支来处理所有未预料到的情况，这能让你的程序更加健壮。
5. **明确限制**：记住 `switch` 只能用于整型/字符型变量的等值判断，不能用于范围和复杂逻辑。