逻辑运算符

1. 引言

- **问题引入**：
  妈妈许诺说：“如果你这次数学考到90分，**并且**英语也考到90分，我就给你买新的游戏机！”
  这里的关键是“**并且**”，它意味着两个条件必须**同时**满足。
  后来妈妈看你压力太大了，又说：好吧，要求降低一点。只要你数学考到90分，**或者**英语考到90分，我就带你去吃大餐！
  这里的关键是“**或者**”，它意味着两个条件只要满足**任意一个**就行。
  在C++中，我们如何表达这种“并且”、“或者”的复杂逻辑呢？
- **价值阐述**：逻辑运算符是构建程序决策的核心。它让我们可以将简单的判断（`a > b`，`c == 0`）组合成复杂的逻辑规则。从网站的用户登录（用户名**和**密码都正确）、游戏胜利条件（敌人数量为0**或**到达终点）、到复杂的AI决策，逻辑运算符无处不在。
- **学习目标**：
  1. 掌握 C++ 的三个逻辑运算符：`&&` (与), `||` (或), `!` (非)。
  2. 理解并能口述每个运算符的运算规则。
  3. 掌握“短路求值”这一重要特性，并理解其意义。
  4. 了解逻辑运算符、关系运算符和算术运算符之间的优先级关系。

2. 核心思想

- **思路铺垫**：
  我们已经知道，关系运算符（如 `>`、`==`）的运算结果是一个 `bool` 值（`true` 或 `false`）。逻辑运算符不做数值计算，它的工作对象就是这些 `bool` 值。它根据输入的 `true` 或 `false`，给出一个最终的 `true` 或 `false` 结果。
- **算法精讲**：
  C++ 主要有三个逻辑运算符，我们用来一一拆解。
  1. **逻辑与 `&&` (AND)**
     - **含义**：表示“并且”，要求左右两边的条件**都**为 `true` 时，最终结果才是 `true`。只要有一个是 `false`，结果就是 `false`。
     - **生活实例**：`数学 >= 90 && 英语 >= 90`。
     - **规则表 (真值表)**：

A	B	A && B
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

- **记忆口诀**：**全真才为真，一假即为假。**

2. **逻辑或 `||` (OR)**

   含义：**表示"或者"，只要左右两边的条件有**任意一个**为 `true`，最终结果就是 `true`。只有当两个都为 `false` 时，结果才是 `false`。**

   生活实例：`数学 >= 90 || 英语 >= 90`。

   规则表 (真值表)：

A	B	A || B
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

- **记忆口诀**：**一真即为真，全假才为假。**

3. **逻辑非 `!` (NOT)**

   含义：表示"取反"，它是一个单目运算符（只有一个操作数）。它会把 `true` 变成 `false`，把 `false` 变成 `true`。

   生活实例：妈妈说："只要你数学**不**低于70分就有奖励"。"不低于70分"等价于 `!(数学 < 70)`。

   规则表 (真值表)：

A	!A
`true`	`false`
`false`	`true`

- **记忆口诀**：**真变假，假变真。**
- **步骤拆解 (特性：短路求值)**
  计算机其实很聪明，它会用最快的方式得到逻辑运算的结果。
  - **对于 `&&` (与)**：`条件A && 条件B`
    1. 计算机先计算 `条件A`。
    2. 如果 `条件A` 的结果是 `false`，它会想："根据'一假即为假'的规则，无论 `条件B` 是什么，最终结果肯定是 `false` 了。"
    3. 于是，计算机**直接返回 `false`，根本不会去计算** `条件B`。这个行为就叫"短路"。
    - **回到例子**：如果你的数学考了88分 (`math >= 90` 为 `false`)，计算机根本不需要再看你的英语成绩，就知道你拿不到游戏机了。
  - **对于 `||` (或)**：`条件A || 条件B`
    1. 计算机先计算 `条件A`。
    2. 如果 `条件A` 的结果是 `true`，它会想："根据'一真即为真'的规则，无论 `条件B` 是什么，最终结果肯定是 `true` 了。"
    3. 于是，计算机**直接返回 `true`，根本不会去计算** `条件B`。
    - **回到例子**：如果你的数学考了100分 (`math >= 90` 为 `true`)，计算机根本不需要再看你的英语成绩，就知道你已经有大餐吃了。

3. 实例

- **具体案例**：
  假设 `int score = 85;`，我们来分析表达式 `score > 80 && score < 90` 的真假。
- **分步演示**：
  1. **拆分表达式**：这个逻辑表达式由两个关系表达式 `score > 80` 和 `score < 90`，以及一个逻辑运算符 `&&` 组成。
  2. **计算左侧**：`score > 80` -> `85 > 80` -> 结果为 `true`。
  3. **判断短路**：因为是 `&&` 运算，左侧为 `true` 还不能确定最终结果，所以**不能短路**，必须继续计算右侧。
  4. **计算右侧**：`score < 90` -> `85 < 90` -> 结果为 `true`。
  5. **最终结果**：`true && true` -> 根据 `&&` 的规则，最终结果为 `true`。

4. 代码实现

- **问题**：输入一个学生的数学和英语成绩，判断他是否能拿到游戏机（两门都>=90），是否能吃到大餐（任意一门>=90）。
- **完整代码**：

```
#include <iostream>
using namespace std;

int main() {
    int math_score, english_score;
    cout << "请输入数学成绩和英语成绩: ";
    cin >> math_score >> english_score;

    // 判断是否能拿到游戏机 (&&)
    bool can_get_game = (math_score >= 90) && (english_score >= 90);

    // 判断是否能吃到大餐 (||)
    bool can_get_dinner = (math_score >= 90) || (english_score >= 90);

    // (了解即可)C++中，boolalpha 可以让我们直接输出 true 或 false 而不是被计算机转化的 0 和 1
    // 如果不添加下面这一行 cout << boolalpha; 那么输出 true 或 false 就会被计算机转为 1 和 0
    cout << boolalpha;

    cout << "能拿到游戏机吗? " << can_get_game << endl;
    cout << "能吃到大餐吗? " << can_get_dinner << endl;

    return 0;
}
```

- **代码解读**：
  - `bool can_get_game = (math_score >= 90) && (english_score >= 90);`
    - 我们把整个逻辑表达式的结果存入一个 `bool` 类型的变量 `can_get_game`。

        - 括号 `()` 的使用让代码的可读性变得非常好，我们一眼就能看出是在对两个条件进行 `&&` 操作。
    - `cout << boolalpha;`
        - 这是一个输出格式控制器。加上它之后，`cout` 在打印 `bool` 值时，会打印出 "true" 或 "false"，而不是 "1" 或 "0"，这样结果更直观。

5. 深化

- **运算符优先级**
  当一个复杂的表达式里什么运算符都有的时候，计算顺序是怎样的？
  口诀：非算关与或（非 > 算术 > 关系 > 与 > 或）

  1. **最高**：`!`（逻辑非）和 `()`（括号）
  2. **次高**：算术运算符（`*`，`/`，`+`，`-`）
  3. **中等**：关系运算符（`>`，`<`，`==`，`!=`）
  4. **次低**：`&&`（逻辑与）
  5. **最低**：`||`（逻辑或）

  黄金法则：如果你不确定优先级，或者想让代码意图更清晰，**那就毫不犹豫地使用括号 `()`！**例如 `(a + b > 5) && (c == 0)`。

- **常见问题**：
  1. `0 || 0` **的结果是什么？**
     - 我们知道“非零即为真”，所以 `0` 在逻辑判断中被当作 `false`。
     - 表达式相当于 `false || false`。根据 `||` 的“全假才为假”规则，结果为 `false`（会输出 0）。
  2. `!(-3)` 的结果是什么？
     - `-3` 是一个非零数，所以它被当作 `true`。
     - 表达式相当于 `!(true)`。根据 `!` 的取反规则，结果为 `false`（会输出 0）。

6. 总结

- **要点回顾**：
  1. **逻辑与 `&&`**：全真才为真。左侧为假时发生短路。
  2. **逻辑或 `||`**：一真即为真。左侧为真时发生短路。
  3. **逻辑非 `!`**：取反，真变假，假变真。
  4. **优先级**：`!` > 算术 > 关系 > `&&` > `||`。**括号是王道，口诀：非算关与或。**
  5. **短路求值**：一种优化策略，可以避免不必要的计算。