C++ 变量与常量 讲义

一、 为什么我们需要变量?

例 1-8：存钱问题（参考书本 P9 1.3 节）
Uim 的银行账户里原有 100 元。经过了以下操作：

1. 往里面存了 10 元；
2. 购物花掉了 20 元；
3. 把里面的钱全部取出。
   需求：在每次操作后输出账户余额。

分析： 这个问题无法用一个简单的算式一次性解决。我们需要在每一步操作后，把算出来的**中间结果**保存下来，留给下一步继续计算。在编程中，用来**存储计算中间结果**的工具，就叫做**变量** 。

二、 计算机里的容器

在生活中，我们用**容器**（如碗、盘子）来存放物品。在计算机中，我们用**变量**来存放数据。

做菜的比喻（参考书本 P15 第一自然段）：

- **原材料** = 我们要处理的数据（数字、字符等）。
- **容器** = 变量（用来装数据）。
- **操作台** = 计算机的内存。

内存的物理形态：
计算机的数据存放于内存中。内存就像一排排的小抽屉，这样的小抽屉就是一个**字节 (Byte)**，它是计算机中最小的存储单位。不同的数据类型（比如一张照片和一个单纯的数字），占用的抽屉数量是不一样的。

三、 声明变量：挑个柜子写上自己的名字

如果你想占用内存里的“抽屉”，必须先向计算机**声明，否则别人（其他程序）可能也会去用这个抽屉，导致数据大乱斗。

声明变量的格式： `数据类型 + 变量名;`

1. 数据类型（选多大的柜子）

参考书本 P16 表 2-1
做菜时，装高汤不能用菜篓子。编程时，不同的数据也要选不同类型的柜子：

分类	类型名称	关键字	说明 / 存储范围
整型 (整数)	短整型	`short`	-32768 ~ 32767
	基本整型	`int`	**-2147483648 ~ 2147483647 (最常用)**

分类	类型名称	关键字	说明 / 存储范围
	长整型	`long long`	存非常大的整数
浮点型 (小数)	单精度型	`float`	精确到小数点后 6~7 位
	双精度型	`double`	**精确到小数点后 15~16 位 (最常用)**
字符型	字符型	`char`	单个字符，例如 `A`，`!`，`?`，`b` 等
布尔型	布尔型	`bool`	只有真 (`true` 正确的) 和假 (`false` 错误的) 两种状态

2. 变量名命名规则（贴什么名字）

参考书本 P11 第三自然段
变量名必须做到**见名知意**（具有自解释性，比如用 `balance` 表示余额，用 `age` 表示年龄）。

- **强制法则：** 只能由字母、数字和下划线（`_`）组成。
- **数字禁忌：** 数字**绝对不能**出现在第一个位置上！（`num1` 正确，`1num` 错误）。
- **避开关键字：** 不能使用 C++ 自带的指令词（如 `include`，`return`，`int`）。
- **不可重复：** 同一作用域（范围）内，名字不能重复，你不能命名两个柜子都叫做 `pingguo`。
- **语言限制：** 英文不好可以用拼音（如 `pingguo`），但**严禁使用中文**！

四、 赋值与初始化：往柜子里装东西

声明了柜子后，我们如何把数据放进去呢？这就需要用到**赋值符 (`=`)**。
注意： 在 C++ 中，`=` 不叫“等于”，它叫“赋值”。它的动作是：**将等号右边的数值，粘贴（覆盖）给左边的变量。**

1. 变量的初始化

初始化指的是**在定义变量的同时就给它赋值**，避免柜子里残留以前的垃圾数据。比方说之前有人往这个柜子里放了化学试剂，如果你直接往里放食材的话，肯定就会有危险！

```
int num; // 未初始化，如果直接去读取里面的数据，会读取到随机的垃圾数字
int a = 10; // 安全：定义整数 a，并初始化为 10
double b = 3.123; // 安全：定义小数 b，并初始化为 3.123
```

2. 连续定义与赋值（补充知识，请记在书本 P15 页）

如果我们需要同时申请多个同类型的变量，可以用逗号 `,` 连写：

```
int a, b; // 同时申请两个空柜子 a 和 b
int x = 10, y = 20; // 同时申请两个柜子 x 和 y 并装入初始值
```

3. 赋值的覆盖特性与累加

对变量的赋值是一个**覆盖**过程。新数据装进去，旧数据就彻底消失了。

变量的自我更新：

```
int num1 = 1;
num1 = num1 + 7; // 此时 num1 变成了 8
```

简单写法（书本 P10 第六自然段）：
为了让代码更简洁，程序员发明了简写形式：

- `num1 += 7;` 完全等同于 `num1 = num1 + 7;`
- `a += b;` 完全等同于 `a = a + b;`

五、 常量

例 1-9（P10 页）：求圆的周长面积和体积

在计算圆周率 π 时，我们需要一个容器来装 `3.14159`。但是，圆周率是一个自然界的真理，在程序运行期间我们**不允许它被意外修改。**

这时，我们需要使用关键字 `const` 来声明一个**常量。**因为在实际的工程中，一个软件是由很多程序员开发的，定义常量就能让别的程序员知道我定义的这个变量不能修改，改了会出问题。

常量的规则：

1. **必须在声明时初始化**，否则编译直接报错。
2. **终生不可更改**，不能再通过赋值符 `=` 修改它。

```
// 声明一个双精度浮点数常量 PI
const double PI = 3.14159;
// PI = 3.0; // 如果写这行代码，计算机会报错！
```

六、 综合实战演练

练习：理解并练习书本例 1-10（P11 页），猴子吃桃问题

题目描述： 猴子第一天吃了所有桃子的一半又多吃了一个；第二天又吃了剩下的一半多一个；第三天同理。第四天发现只剩下一个了。问原来有几个？

解题思维（逆向）：
正向推导较为复杂，但如果运用倒推法，问题就能较为简单的解决。

- 第四天：剩下 1 个。
- 第三天：把第四天的数量加上它贪嘴多吃的那 1 个（吐出来），然后再翻倍。
- 第二天、第一天：重复同样的动作。
  我们可以利用变量来持续覆盖、更新当前的桃子数。

代码思路：

- **第一步：** `int peach = 1;`

- 计算机在内存里找了一个整数大小的空抽屉，在抽屉外面贴上了一个标签，名字叫 `peach`（桃子）。然后，把第四天最后剩下的数字 **1**，放进了抽屉里。 *【此时 peach =1】*

- **第二步：** `peach = (peach + 1) * 2;`
  - 这行代码在数学上是不成立的，但在编程里，我们要**先看等号右边，再看等号左边。** 计算机先执行右边的计算：它拉开名为 `peach` 的抽屉，发现里面是 1。于是它在后台的大脑里计算 `(1 + 1) * 2`，算出来的结果是 4。 接着，执行等号（赋值）的动作：计算机地把抽屉里原来的 1 扔掉，把刚刚算出来的 **4** 装了进去！这代表第三天开始时有 4 个桃子。 *【此时 peach = 4】*

- **第三步：** `peach = (peach + 1) * 2;`
  - 代码跟第二步长得一模一样，但结果完全不同！ 计算机再次拉开 `peach` 抽屉，此时里面装的已经是 4 了。计算机在大脑里计算 `(4 + 1) * 2`，得到 10。 然后再次执行覆盖：把 4 扔掉，把 **10** 装进抽屉。这代表第二天开始时有 10 个桃子。 *【此时 peach = 10】*

- **第四步：** `peach = (peach + 1) * 2;`
  - 最后一次倒推。拉开抽屉拿出 10，计算 `(10 + 1) * 2`，得到 22。 扔掉 10，把 **22** 放进抽屉里。这代表第一天小猴子刚买回来的桃子总数。 *【此时 peach = 22】*

- **第五步：** `cout << peach;`
  - 任务完成！我们 `cout` 输出 `peach` 抽屉，把它现在装的最终数字展示在屏幕上。屏幕上将打印出：**22**。

我们全程只用了一个名叫 `peach` 的抽屉，但通过反复使用 `=`（赋值符），我们让抽屉里的数据根据题意进行了变化和覆盖。

程序代码:

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

int main() {
    // 1. 申请抽屉并放入第四天的数量
    int peach = 1;

    // 2. 倒推第三天的数量
    peach = (peach + 1) * 2;

    // 3. 倒推第二天的数量
    peach = (peach + 1) * 2;

    // 4. 倒推第一天（刚买回来时）的数量
    peach = (peach + 1) * 2;

    // 5. 展示抽屉里最终的结果
    cout << peach << endl;

    return 0;
}
```