一、关于const的疑问

- 关于const常量的判别准则
  - 只有 `字面量` 初始化的 `const` 常量才会 `进入符号表`
  - 使用其它 `变量` 初始化的 `const` 常量仍然是 `只读变量`
  - 被 `volatile` 修饰的 `const` 常量 `不会进入符号表`，仅仅是只读变量，**这一条会影响第一条**

> 同时存在 `volatile` 和 `用字面量初始化const常量` 时，标识符仅仅是 `只读变量`

> 在 `编译器期间不能直接确定初始值` 的 `const标识符`，都被作为 `只读变量` 处理

- `const引用的类型` 与 `初始化变量的类型`，**注意是类型**
  - 相同时：初始化变量称为只读变量
  - 不同时：生成一个新的只读变量

```
char c = 'c';
const int& rc = c;
```

实验分析

```
#include <stdio.h>

int main(void)
{
    const int x = 1;
    const int& rx = x;
    int& nrx = const_cast<int&>(rx);

    nrx = 5;

    printf("x = %d\n", x);
    printf("rx = %d\n", rx);
    printf("nrx = %d\n", nrx);
    printf("&x = %p\n", &x);
    printf("&rx = %p\n", &rx);
    printf("&nrx = %p\n", &nrx);

    volatile const int y = 2;  // warning
    int* p = const_cast<int*>(&y);
    *p = 6;
```

```
21      printf("y = %d\n", y);
22      printf("p = %p\n", p);
23
24      const int z = y;  // read-only variable
25
26      p = const_cast<int*>(&z);
27      *p = 7;
28      printf("z = %d\n", z);
29      printf("p = %p\n", p);
30
31      // same type
32      char c = 'c';
33
34      const int& rc = c;
35      int& trc = const_cast<int&>(rc);
36
37      printf("c = %c\n", c);
38      printf("rc = %c\n", rc);
39      printf("trc = %c\n", trc);
40
41      trc = 'd';
42
43      printf("c = %c\n", c);
44      printf("rc = %c\n", rc);
45      printf("trc = %c\n", trc);
46
47      return 0;
48  }
```

输出

▶ 详情

二、关于引用得疑问

> 引用与指针有什么关系？如何理解 引用得本质就是指针常量

- 指针是一个变量
  - 值为一个内存地址，不需要初始化，可以保存不同的地址
  - 通过指针可以访问对应内存地址中的值
  - 指针可以被const修饰成为 常量 或 只读变量
- 引用只是一个变量的名字
  - 对引用的操作(赋值、取地址等)都会传递到代表的变量上
  - const引用使其代表的变量具有只读属性

    - 引用必须在定义时初始化，之后无法代表其它变量

- 从使用C++语言的角度【开发编写代码时】
    - 引用与指针没有任何关系
    - 引用是变量的名字，操作引用就是操作对应的变量

- 从C++编译器的角度【调试时】
    - 在编译器内部，使用指针常量实现 引用
    - 因此 引用在定义时必须初始化

下面的代码有问题?

```
int a = 1;
int main(void)
{

    int b = 2;
    int* pc = new int(3);
    int& array[] = {a, b};
}
```

> 有问题，C++不支持 引用数组，假设 `int& array[] = {a, b};` 是成立的，array数组里面的元素都是引用类型，
> 那么对元素进行取地址的时候，并不是元素的地址，而是引用所代表变量的地址，相邻元素的地址相减等于 元素的大小。
> 正常情况，数组中，每个元素的地址是相邻的，但是如果是 `&array[0]`，
> **得到的值是 变量a 的地址也就是全局数据区的地址**、 `&array[1]`，
> **得到的值是 变量b 的地址也就是栈区的地址**。相邻元素的地址相减不一定等于 元素的大小。
> 这就破坏了数组元素相邻的地址连续性。干脆C++就不支持。

`lesson_12-2.cpp`

```
#include <stdio.h>

struct C
{
    int& a;
    int& b;
    int& c;
};


int x = 1;

int main(void)
{
```

```
    int y = 2;
    int z = 4;

    struct C c = {x, y, z};

    printf("c.a = %p\n", &c.a);
    printf("c.b = %p\n", &c.b);
    printf("c.c = %p\n", &c.c);


    //int& arr[] = {a, b, c}; // error

    return 0;
}
```

小结

- 指针是一个变量
- 引用是一个变量的新名字
- `const`引用能够生成 `新的只读变量` （引用的类型与数据的类型不一致）
- 在编译器内部使用 `指针常量实现 引用`
- 在编译时不能直接确定初始值的 `const` 标识符都是 只读变量