3_进化后的const

C语言中的const特性

1. `const` 修饰的变量是**只读的属性**，本质还是**变量**
2. `const` 修饰的局部变量在栈上分配空间
3. `const` 修饰的 **全局变量在只读存储区**
4. `const` 只在 **编译期有用**，在 **运行期无用**

> `const` 修饰的变量并不是真的常量，它只是告诉编译器，修饰的变量不能出现在赋值符号的左边
> `const` 不能定义真正意义上的常量

```c
#include <stdio.h>

int main(void)
{
    const int c = 0;
    int * pc = (int *)&c;

    printf("Begin...\n");

    *pc = 5;

    printf("c = %d\n", c);

    printf("*pc = %d\n", *pc);

    printf("End...\n");

    return 0;
}
```

> 以下经过 `gcc` 编译后输出的结果：

```
Begin...
c = 5
*pc = 5
End...
```

> 以下经过 `g++` 编译后输出的结果：

```
Begin...
c = 0
*pc = 5
End...
```

- C++在C的基础上对 `const` 进行了 **优化处理**
  - 当遇到 `const` 声明时 **在符号表中放入常量**
  - 编译过程中若发现使用了常量则 **直接以符号表中的替换**
  - 编译过程中若发现以下情况，则给对应的常量分配存储空间
    - const修饰的变量定义时，以及被其它变量初始化。
    - 对 `const` 常量使用了 `extern`
    - 对 `const` 常量使用了 `&` 操作符
- C++ 中的 `const` 常量 类似 于宏定义
  - const int c = 5; ≈ #define c 5
- C++ 中的 `const` 常量与宏定义不同
  - `const` 常量是由编译器处理
  - 编译器对 `const` 常量进行 **类型的检查** 和 **作用域的检查**
  - 宏定义由预处理器处理，单纯的文本替换

> **C++中的 const变量的使用**
>
> 当C++编译器遇到const修饰的变量时
>
> 1. 为变量分配内存空间
> 2. 然后将变量与对应的值加入一张符号表中
> 3. 后面当遇到使用该变量的时候直接从 符号表中取出对应变量的值进行替换。不会访问该空间的内容

const与宏的区别

```
#include <stdio.h>
void f()
{
    #define a 3
    const int b = 4;
}

void g()
{
    printf("a = %d\n", a);
    //printf("b = %d\n", b);
}

int main(void)
```

```
{
    const int A = 1;
    const int B = 2;
    int arr[A + B] = {0};
    int i = 0;

    for (; i < (A + B); i++)
        printf("arr[%d] = %d", i, arr[i]);

    f();
    g();


    return 0;
}
```

> `f函数的const b` 无法作用在 `g函数`
> `define` 作用于整个文件
> `const` 只作用在所在的作用域

小结

- C++ 的 `const` 是一个真正意义上的常量
- C++ **完全兼容** C语言中的 `const` 常量的语法特性