2020/05/06_C++_第2课_const、引用和默认参

笔记本:	C++
创建时间:	2020/5/6 星期三 15:03
作者:	ileemi
标签:	const, 默认参, 引用

bool

C语言并没有彻底从语法上支持“真”和“假”，只是用 0 和非 0 来代表。这点在 C++ 中得到了改善，C++ 新增了 bool 类型（布尔类型），它一般占用 1 个字节长度。bool 类型只有两个取值，true 和 false：true 表示“真”，false 表示“假”。

bool 是类型名字，也是 C++ 中的关键字，它的用法和 int、char、long 是一样的。

C语言使用布尔类型的方式，可以通过宏定义，设置TRUE、FALSE 的分别为1、0

C示例：

```
#include <stdio.h>
typedef int BOOL;
#define TRUE 1
#define FALSE 0
int main()
{
        BOOL bVal1 = TRUE;
        BOOL bVal2 = FALSE;
        //bVal1 = 5;   //可读性不好\
    return 0;
}
```

C++布尔类型示例：

```
#include <iostream>
using namespace std;
typedef int BOOL;
#define TRUE 1
```

```
#define FALSE 0

int main()
{
	bool bVal1 = true;		//1
	bool bVal2 = false;		//0 占用一个字节

	bVal1 = 9;					//可读性不好，编译可以通过，在内存地址上的值，仍然是1

	bool bVal2 = -1;			//布尔值的解释 非0为真true 0为假false

    // C++中的布尔类型， bool, true, false

    return 0;
}
```

在 C++ 中使用 cout 输出 bool 变量的值时还是用数字 1 和 0 表示，而不是 true 或 false

```
	bool bVal1 = true;
	bool bVal2 = false;

	cout << bVal1 << endl;  //1
	cout << bVal2 << endl;  //0
```

const的使用

无参宏是有缺陷的

1、编译预编译阶段做文本替换，编译器不做类型检查

```
#include <iostream>
using namespace std;
#define VALUE 10000
int main()
{
	char ch = VALUE;	//隐式转换 从“int”到“char”截断，截断常量值
	cout << (int)ch << endl;
}
```

2、调试程序时，

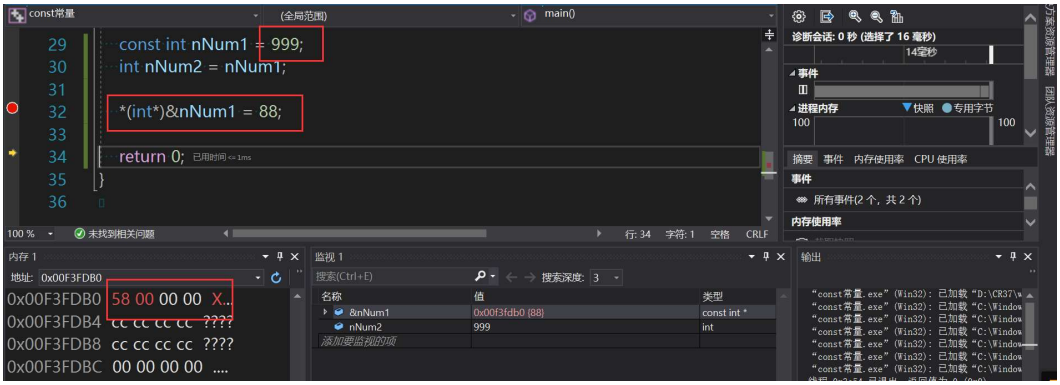
//DOTO

> const 是 constant 的缩写，本意是不变的，不易改变的意思。在 C++ 中是用来修饰内置类型变量，可自定义对象，成员函数，返回值，函数参数，C++ const 允许指定一个语义约束，编译器会强制实施这个约束，允许程序员告诉编译器某值是保持不变的。如果在编程中确实有某个值保持不变，就应该明确使用 const。

1、const修饰普通类型的变量

```
    const int nNum1 = 999;
    int nNum2 = nNum1;       //nNum1被定义为一个常量，可以将其值赋值给一个变量
    nNum1 = 88;              //不能再次对const修饰的变量进行赋值，编译失败
```

对于const 修饰的变量nNum1，可以取变量nNum1的地址并将其转换车工指向对应类型的指针，然后再通过间接运算符 * 取对应地址上的值，进行更改，编译阶段进行值替换。




```
#include <iostream>
using namespace std;
const int   VALUE = 10000;              //使用const 定义一个常量
int main()
{
        char ch = VALUE;  //隐式转换 从“int”到“char”截断，截断常量值
        cout << (int)ch << endl;

        //VALUE = 999;     //error，VALUE”：不能给常量赋值
        const int nVal = 10;
        *(int*)&nVal = 9;  //C++会把const定义的常量，编译阶段进行值替换
        cout << nVal << endl;
```

```
    //const int nVal;//定义一个常量需要为其赋初值，不然就没有意义
}
```

C语言的中的const是一个假const，定义的常量值可以被修改
C++中的const是一个真const，定义的常量值不能修改

2、const 修饰函数参数

```
#include <iostream>
using namespace std;
void ShowData(const int nTest)
{
	//nTest++;				//编译错误，nTest为常量，不能被改变
	cout << nTest << endl;
}
int main()
{
	ShowData(999);		//输出999
	return 0;
}
```

3、 函数参数为指针使用const进行修饰，可以防止指针指向的值不被意外篡改

```
#include <iostream>
using namespace std;
void ShowAryData(const int* nAry, int nArySize)
{
    //nAry[2] =
999;                                                    //编译失败，
不能给常量赋值
    *(int*)&nAry[2] = 999;                              //通过间接访问地
址上的值，可以修改
    for (int i = 0; i < nArySize; i++)
    {
        cout << nAry[i] << endl;
    }
}
int main()
{
    int nAry[] = { 1,2,3,5,6,7,8,9 };
    int nArySize = sizeof(nAry) / sizeof(nAry[0]);
    ShowAryData(nAry, nArySize);
	return 0;
}
```

const type *p 指针本身可以被修改，指针指向的内容不能被修改

```
        int nNum1 = 999;
        int nNum2 = 888;
        const int* p = &nNum1;
        p = &nNum2;
        //*p = 777;                                    //编译失败，不能给常量赋
值
```

type const *p 和 const type *p 是一样的

```
        int nNum1 = 999;
        int nNum2 = 888
        int const* p = &nNum1;
        p = &nNum2;
        //*p = 777;                                    //编译失败，不能给常量赋
值
```

type * const p 指针本身不可以被修改，指针指向的内容可以被修改

```
        int nNum1 = 999;
        int nNum2 = 888;
        int* const p = &nNum1;
        //p = &nNum2;    //编译失败，不能给常量赋值
        *p = 777;
```

const type * const p 指针本身不可以被修改，指针指向的内容不能被修改

```
        int nNum1 = 999;
        int nNum2 = 888;
        const int* const p = &nNum1;
        //p = &nNum2;          //编译失败，不能给常量赋值
        //*p = 777;                        //编译失败，不能给常量赋值
```

const 修饰的指针之间的转换 --限制少的向限制多的进行转换，限制多的就不可以向限制少的进行转换

```
        //const 修饰的指针之间的转换 --限制少的向限制多的进行转换，限制
多的就不可以向限制少的进行转换
        const int* p1 = nullptr;
        int* const p2 = p1;                    //不可以 无法从“const int
*”转换为“int *”
        const int* const p3 = p2;
        int* const p4 = p2;                    //类型一致
        int* const p5 = p3;                    //不可以 无法从“const int
```

```
*const ”转换为“int *”


    //const修饰pNum，pNum 是一个指针 是const
    char* const pCh = "abc";
    *pCh = 'C'; //ok
    pCh++;  //error

    //*pCh1 是const，指针指向的内容是const
   const char* pCh1 = "ABC";
   *pCh1 = "BCD"; //error
   //看const 在 * 号的前面还是后面，const在*前，指针指向的内容是const，
在*后，指针是const
```

看const离谁最近，离谁进，谁就不能被修改
const修饰的指针间的转换

引用

向函数传递参数的引用调用方法，把**引用的地址复制给形式参数。**在函数内，该引用用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。

指针和引用的区别
指针的缺点：空指针、野指针（未初始化的指针）
使用指针间接访问交换两个整数的值：

```
void Swap(int* p1, int * p2)
{
   int nTmp = *p1;
   *p1 = *p2;
   *p2 = nTmp;
}
int main()
{
    int nNum1 = 999;
    int nNum2 = 888;
    Swap(&nNum1, &nNum2);
    printf("%d %d\r\n", nNum1, nNum2);
    return 0;
}
```

使用C++引用交换两个整数值，按引用传递值，参数引用被传递给函数，就像传递其他值给函数一样。因此相应地，在下面的函数 Swap() 中，您需要声明函数参数为引用类型，该函数用于交换参数所指向的两个整数变量的值

```
void Swap(int& nNum1, int& nNum2)
{
    int nTmp = nNum1;      //保存地址 nNum1 的值
    nNum1 = nNum2;         //把 nNum2 赋值给 nNum1
    nNum2 = nTmp;          //把 nNum1 赋值给 nNum2
}
int main()
{
     int nNum1 = 999;
     int nNum2 = 888;
     Swap(nNum1, nNum2);
     cout << nNum1 << " " << nNum2 << endl;
     return 0;
}
```

为变量取别名，使用不同的名字可以操作同一块内存

```
     int nNum = 999;
     int& nNums = nNum;       //给变量 nNum 取别名
     nNums = 8;
```

引用必须进行初始化

```
        int& nTest;                  // 必须初始化引用
```

引用不存在二级引用，三级引用...

```
     int nNum = 999;
     int&& nNums = nNum;       //没有二级引用
```

引用时，只能引用相同类型的变量

```
        int nNum = 999;
        int& nNums = nNum;
        nNums = 8;
        //int& nTest;                  // 必须初始化引用

        float fltTest = 6.3;
        //int& nRef = fltTest;//只能引用相同类型的变量，无法从“float”转换为“int &”
```

对指针做引用

```
    char ch = 'A';
    char* p = &ch;
    char*& pRef = p;  // 给指针取个别名
    *pRef = 'B';          //相当于 *p = 'B'
    char ch2 = 'C';
    pRef = &ch2;//修改pRef,同样会修改p
```

小坑
对常量进行引用，不能更改其值

```
    int nNum = 8;
    const int& nValRef = nNum;
    //const int& nValRef1 = 3;
    //nValRef1 = 9; //VS2019->//error C3892: “nValRef1”: 不能给常量赋
值
    //int& nValRef2 = 6; //错误
    //nValRef2 = 7;
    //6 = 7;
```

编译器在处理引用的时候，内部仍然是使用指针进行操作

默认参

语法：形参后跟 = 默认实参
使用默认参的规则：
1、可以给参数提供一个默认参数，当程序调用函数的时候，如果没有提供实参，就是用默认参，如果提供实参，则使用实参

2、只能从右向左进行填写，中间不能断开

```
void Test0(int nTest1, int nTest2 = 3, int nTest3 = 5, int nTest1 = 6)
```

3、默认参必须写到头文件(声明中)，而不能写到cpp(实现中)

默认参数只能放在函数声明处或者定义处，能放在声明处就放在声明处，
如果某个参数是默认参数，那么它后面的参数必须都是默认参数

- 大部分情况，别人调用你的代码只能看到函数声明，如果你写在定义处，别人根本不知道你的默认参数是什么。
- 如果你是在定义处写的默认参数，那么你在使用该函数前就需要把函数定义放在前面。不然编译阶段通过无默认参数的函数声明无法确定这个函数是带默认参数的。 所以，默认参数写在声明处啊。

示例：

```cpp
#include <iostream>
using namespace std;
void SayHi(const char* szPeopleName, const char* szHi = "Hello")
{
    cout << szPeopleName << "\t" << szHi << endl;
}

int main()
{
    SayHi("Tom: ");
    SayHi("Jack: ");
    SayHi("小张: ");
    SayHi("小李: ");
    SayHi("小宋: ");
    return 0;

    /*
    输出结果:
    Tom:    Hello
    Jack:   Hello
    小张:   Hello
    小李:   Hello
    小宋:   Hello
    */
}
```