测试框架

在test.h文件中实现了测试框架，命名空间为test，其中包含了两个类和大量的宏来封装测试过程。

## TestCase类

成员变量：

1. testcase\_name(const char\*类型) : 测试案例的名称
2. nTestResult(int类型) : 测试结果
3. nFailed(double) : 测试失败的数量
4. nPassed(double) : 测试成功的数量

成员函数：

1. 构造函数
2. Run函数(纯虚函数)

## UnitTest类

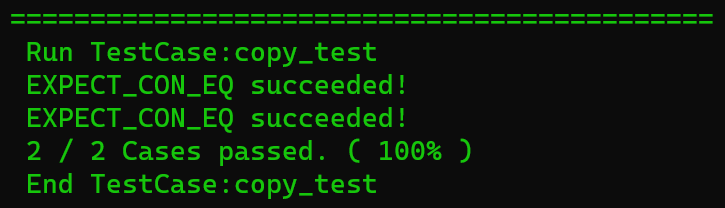
成员变量：

1. CurrentTestCase(TestCase\*) : 当前测试案例
2. nFailed(double) : 失败的案例数量
3. nPassed(double) : 成功的案例数量
4. \_TestCases(vector) : 保存所有的测试案例

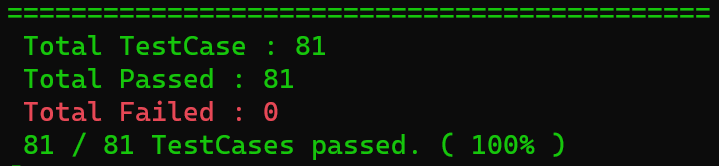
成员函数：

1. Static UnitTest\* GetInstance() : 获取一个测试案例
2. TestCase\* RegisterTestCase(TestCase\*) : 将测试案例加入到队列中
3. Run函数 : 对当前队列中的所有案例进行测试，并输出每个测试的结果，最后输出总的测试结果

单个测试输出格式:



总的输出格式：



# 宏

注意：

当有多条执行语句时，用do while(0)实现；

在宏的每行最后加上\；

1. TESTNAME\_CASE(testcase\_name) : 将测试案例的名字换成testcase\_name\_Test
2. TEST(testcase\_name) : 将每一个测试案例封装为一个继承TestCase的类

成员变量：

\_testcase(TestCase\*) : 直接调用RegisterTestCase

成员函数：

构造函数

Run函数(虚函数)

1. 真假断言
2. EXPECT\_TRUE(condition)
3. EXPECT\_FALSE(condition)
4. 条件断言(v1为期望值，v2为实际值，若断言为假则将两个参数都输出)

Note:

1. 参数应满足 EXPECT\_\*(Expect, Actual)的格式，左边是期望值，右边是实际值

2. 在断言失败时，会将期望值与实际值打印出来

3. 参数值必须是可通过断言的比较操作符进行比较的，参数值还必须支持 << 操作符来将值输入到 ostream 中

4. 这些断言可以用于用户自定义型别，但必须重载相应的比较操作符（如 == 、< 等）

5. EXPECT\_EQ 对指针进行的是地址比较。即比较的是它们是否指向相同的内存地址，而不是它们指向的内容是否相等。如果想比较两个 C 字符串(const char\*)的值，请使用 EXPECT\_STREQ 。特别一提的是，要验证一个 C 字符串是否为空(NULL)，请使用 EXPECT\_STREQ(NULL, c\_str)。但是要比较两个 string 对象时，应该使用 EXPECT\_EQ

1. EXPECT\_EQ(v1, v2)
2. EXPECT\_NE(v1, v2)
3. EXPECT\_LT(v1, v2)
4. EXPECT\_LE(v1, v2)
5. EXPECT\_GT(v1, v2)
6. EXPECT\_GE(v1, v2)
7. 字符串比较

Note:

1. 参数应满足 EXPECT\_STR\*(Expect, Actual)的格式，左边是期望值，右边是实际值

2. 该组断言用于比较两个 C 字符串。如果你想要比较两个 string 对象，相应地使用

EXPECT\_EQ、EXPECT\_NE 等断言

3. EXPECT\_STREQ 和 EXPECT\_STRNE 不接受宽字符串（wchar\_t\*）

4. 一个 NULL 指针和一个空字符串会不是一样的

1. EXPECT\_STREQ(v1, v2)
2. EXPECT\_STRNE(v1, v2)
3. 指针比较

Note:

1. 参数应满足 EXPECT\_PTR\_\*(Expect, Actual)、

EXPECT\_PTR\_RANGE\_\*(Expect, Actual, len)的格式，

即参数表中期望值在实际值左边

2. EXPECT\_PTR\_EQ 比较的是指针所指元素的值，如果要比较

指针指向的地址是否相等，请用 EXPECT\_EQ

3. EXPECT\_PTR\_RANGE\_\* 比较的是从 p1，p2 开始，

长度为 len 的区间，请确保区间长度有效

1. EXPECT\_PTR\_EQ(v1, v2)
2. EXPECT\_PTR\_NE(v1, v2)
3. EXPECT\_PTR\_RANGE\_EQ(v1, v2, len) : 调用std的equal函数
4. EXPECT\_PTR\_RANGE\_NE(v1, v2, len) : 调用std的equal函数
5. 容器比较

Note:

1. 容器可以是 STL 容器，自定义的容器，或者数组，但不可以是指针

2. 容器的数据类型要能够进行比较，类型一致或可以发生隐式转换

3. EXPECT\_CON\_EQ 测试失败时，会打印首次不相等的两个值

1. EXPECT\_CON\_EQ(c1, c2)
2. EXPECT\_CON\_NE(c1, c2)
3. 不同情况的数量级

#if defined(\_DEBUG) || defined(DEBUG)

#define LEN1 10000

#define LEN2 100000

#define LEN3 1000000

#else

#define LEN1 100000

#define LEN2 1000000

#define LEN3 10000000

#endif

#define SCALE\_LLL(N) (N \* 20)

#define SCALE\_LL(N) (N \* 10)

#define SCALE\_L(N) (N \* 5)

#define SCALE\_M(N) (N)

#define SCALE\_S(N) (N / 5)

#define SCALE\_SS(N) (N / 10)

#define SCALE\_SSS(N) (N / 20)

#define WIDE 14

1. 输出通过提示

#define PASSED std::cout << "[ PASSED ]\n"

1. 遍历输出容器

注意：在define中使用#可以让变量转换为字符串

例如：

#define toStr(a) \

string name = #a;\

COUT(container)



STR\_COUT(str)

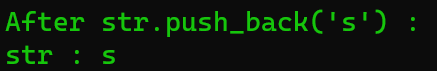


1. 输出容器函数调用后的结果

FUN\_AFTER(con, fun)



STR\_FUN\_AFTER(str, fun)



1. 输出容器调用函数的值

FUN\_VALUE(fun)



1. 输出测试数量级

TEST\_LEN(len1, len2, len3, width)

用一个std::stringstream对象将三个size\_t类型len以空格分隔的形式读入，再输出到三个string对象中，实现size\_t类型到string类型的转换

然后调整并输出：

str1 += " |";

std::cout << std::setw(wide) << str1;

str2 += " |";

std::cout << std::setw(wide) << str2;

str3 += " |";

std::cout << std::setw(wide) << str3 << "\n";

1. 常用测试性能的宏

FUN\_TEST\_FORMAT1(mode, fun, arg, count)

FUN\_TEST\_FORMAT2(mode, fun, arg1, arg2, count)

LIST\_SORT\_DO\_TEST(mode, count)

MAP\_EMPLACE\_DO\_TEST(mode, con, count)

注意：mode的值为std或者mystl；在调用函数和声明容器时需要注意(可以改名为NameSpace)

CLOCKS\_PER\_SEC是标准c的time.h头函数中宏定义的一个常数，表示一秒钟内CPU运行的时钟周期数，要计算运行某程序所需的时间只需要利用clock()函数得到运行此程序所消耗的钟计时单元数,然后再除以CLOCKS\_PER\_SEC即可。

利用snprintf将n写进char数组buf中，然后再转换为string类型

然后调整并输出：

t += "ms |";

std::cout << std::setw(WIDE) << t;

1. 重构重复代码

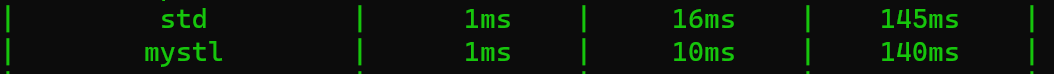
CON\_TEST\_P1(con, fun, arg, len1, len2, len3)

CON\_TEST\_P2(con, fun, arg1, arg2, len1, len2, len3)

LIST\_SORT\_TEST(len1, len2, len3)

MAP\_EMPLACE\_TEST(con, len1, len2, len3)

利用std和mystl区别mode，在每个模式下分别调用三个不同数量级参数的宏完成测试



1. 运行所有测试案例

RUN\_ALL\_TESTS()

1. 是否开启性能模式

PERFORMANCE\_TEST\_ON

默认开启

1. 是否开启大数据量测试

LARGER\_DATA\_TEST\_ON

默认不开启