# §.关于动态内存申请后越界访问的深度讨论

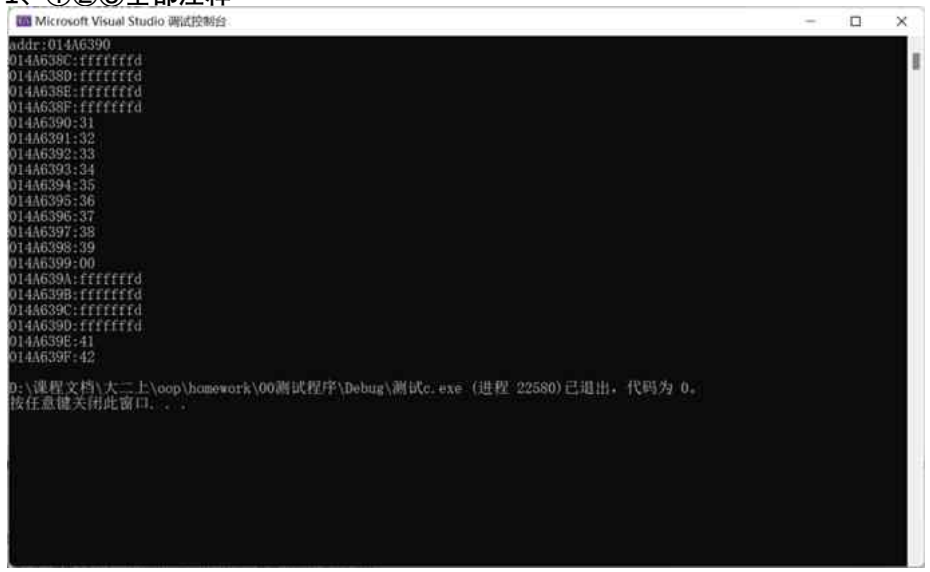
★ 如何判断动态申请越界（C方式，注意源程序后缀为.c）

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char *p;
    p = (char *)malloc(10 * sizeof(char));
    if (p == NULL)
        return -1;
    strcpy(p, "123456789");
①   p[10] = 'a';     //此句越界
    p[14] = 'A';     //此句越界
    p[15] = 'B';     //此句越界
②   p[10] = '\xfd'; //此句越界
    printf("addr:%p\n", p);
    for (int i = -4; i < 16; i++) //注意，只有0-9是合理范围，其余都是越界读
        printf("%p:%02x\n", (p+i), p[i]);
③   free(p);

    return 0;
}
```

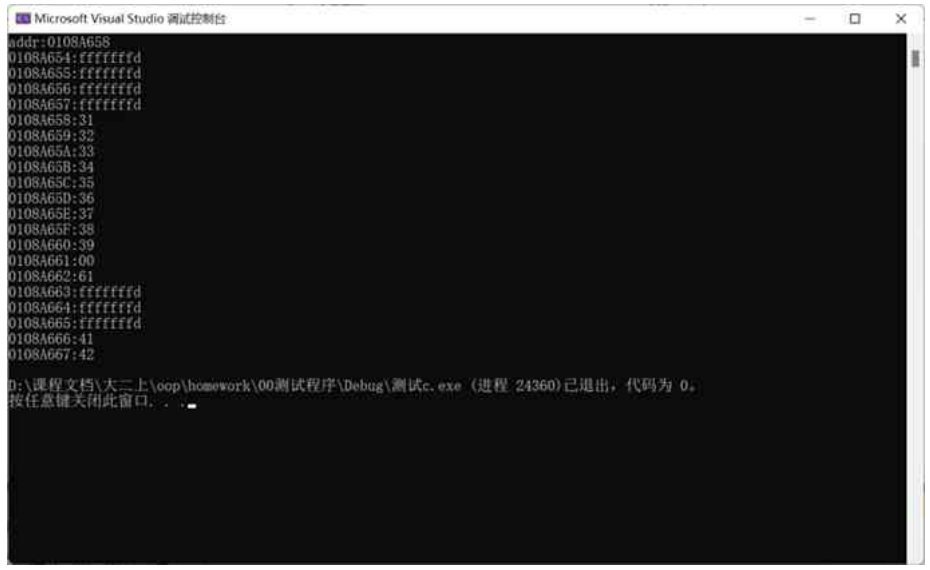
在VS2022的x86/Debug模式下运行：
1、①②③全部注释，观察运行结果
2、①放开，②③注释，观察运行结果
3、①③放开，②注释，观察运行结果
4、①②③全部放开，观察运行结果
结论：VS的Debug模式是如何判断
动态申请内存访问越界的？

再观察下面四种环境下的运行结果：
VS2022 x86/Release
Dev 32bit-Debug
Dev 32bit-Release
Linux
每种讨论的结果可截图+文字说明，
如果几种环境的结果一致，用一个
环境的截图+文字说明即可(可加页)

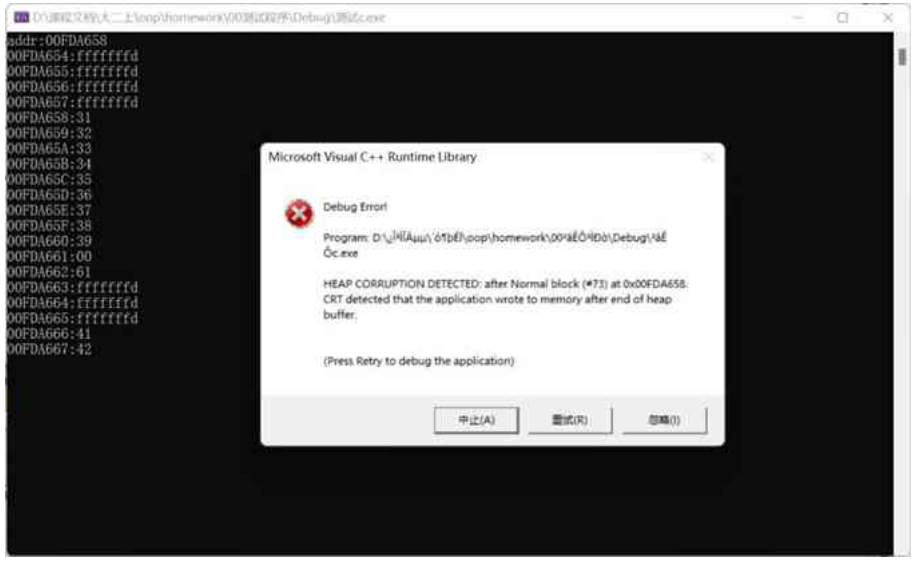
1、①②③全部注释



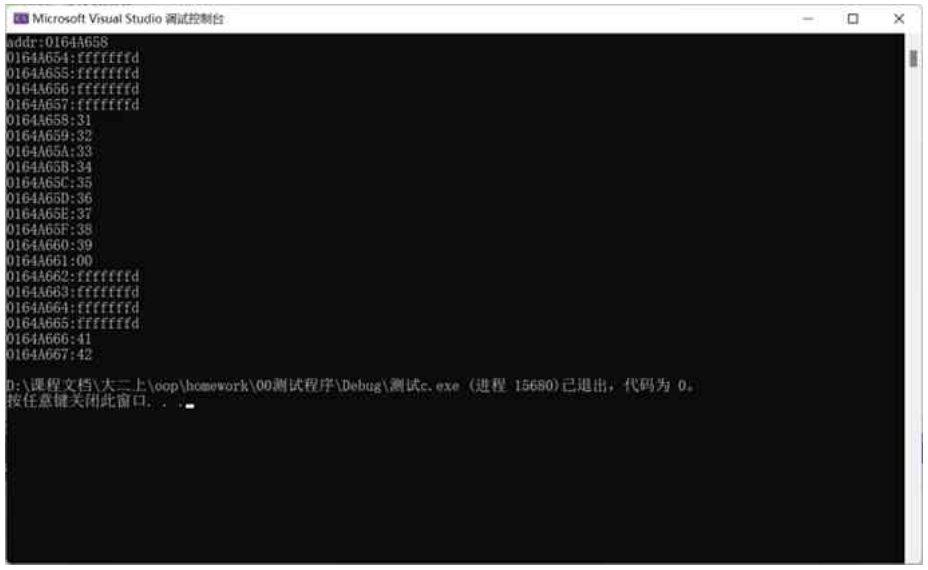

2、①放开，②③注释




3、①③放开，②注释



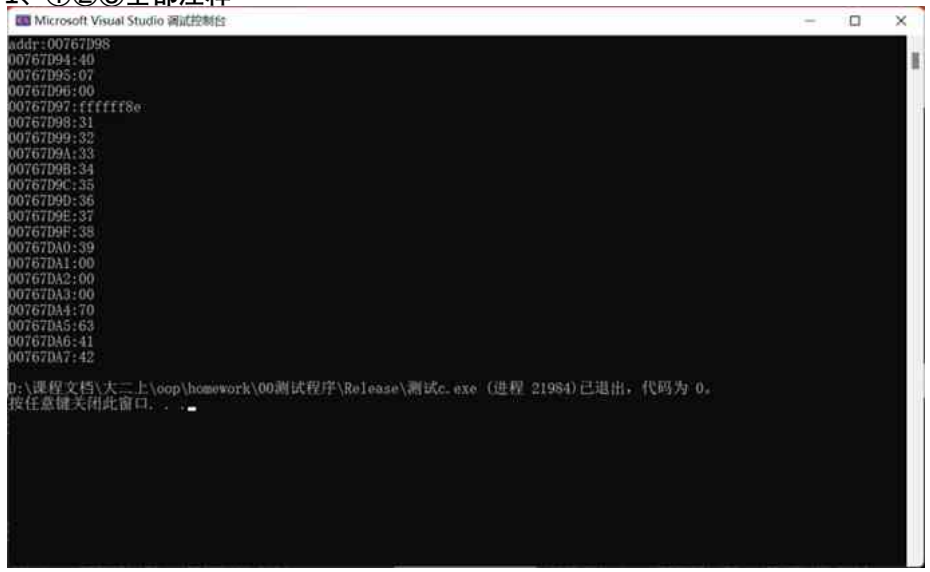

4、①②③全部放开



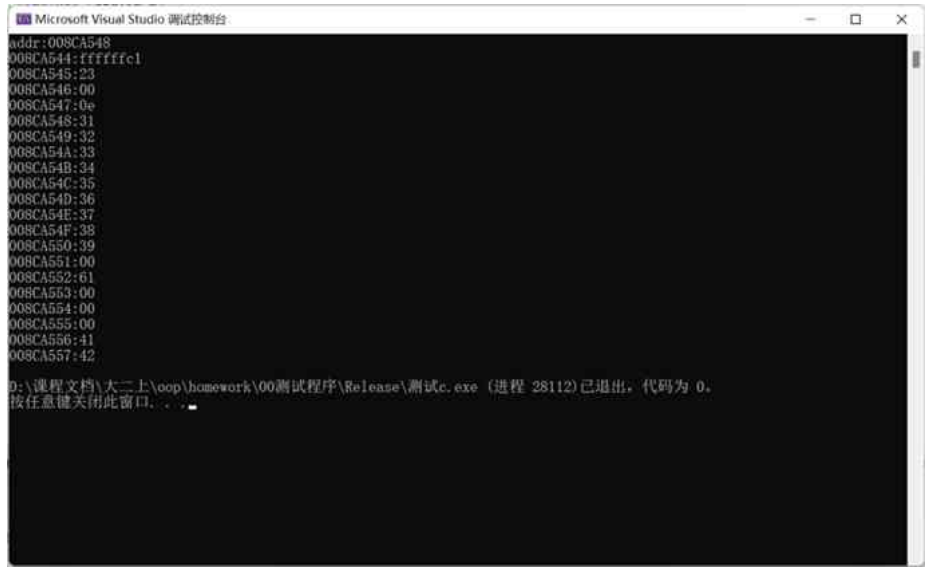

VS2022的x86/Debug

结论：VS的Debug模式通过释放内存(free)时与申请的内存最后紧连的一块未申请内存是否有改变来进行判定内存访问越界

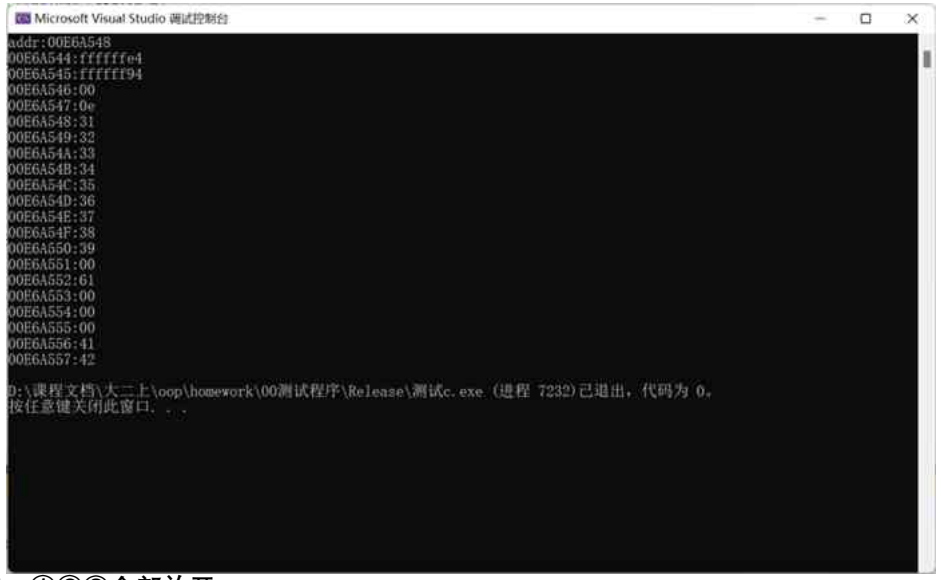
1、①②③全部注释



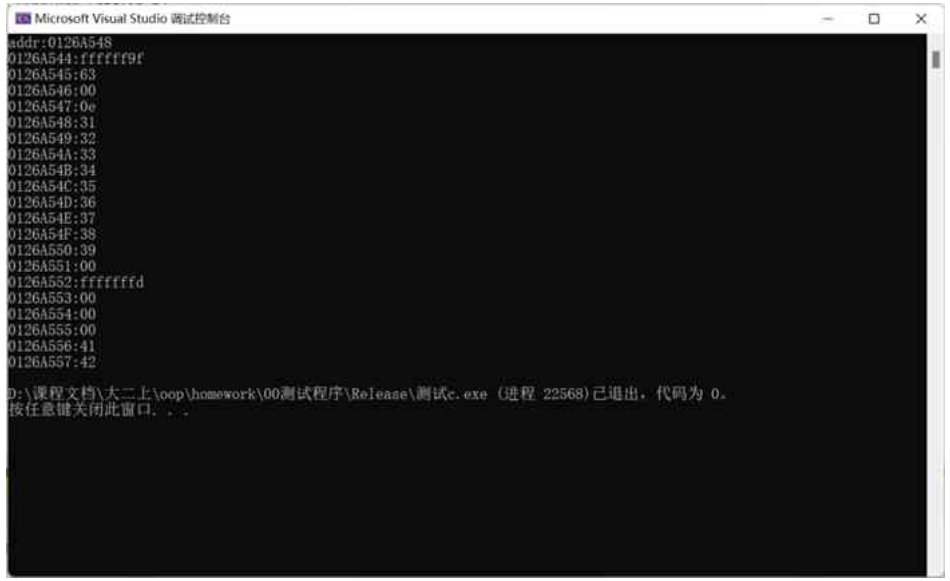

2、①放开，②③注释




3、①③放开，②注释

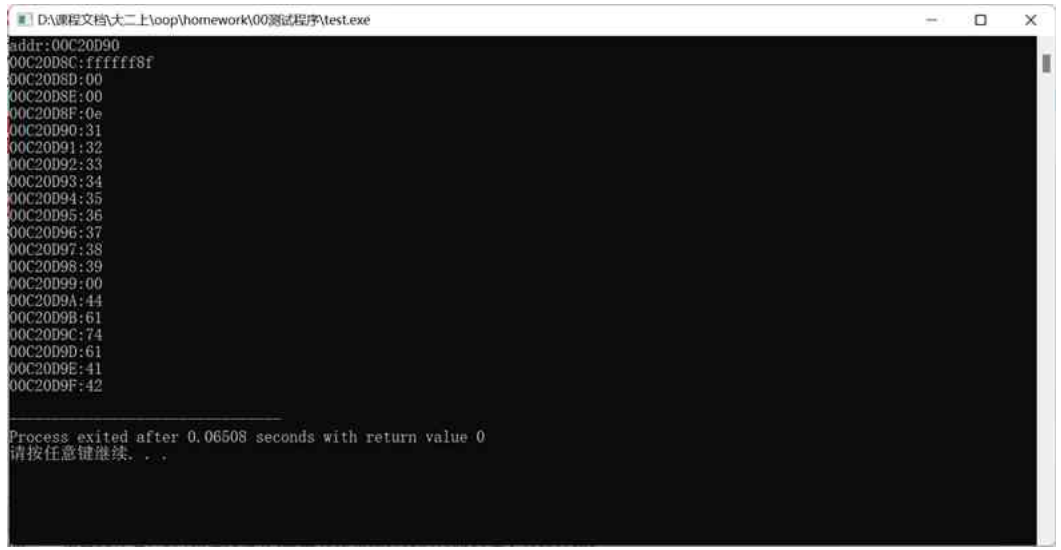



4、①②③全部放开

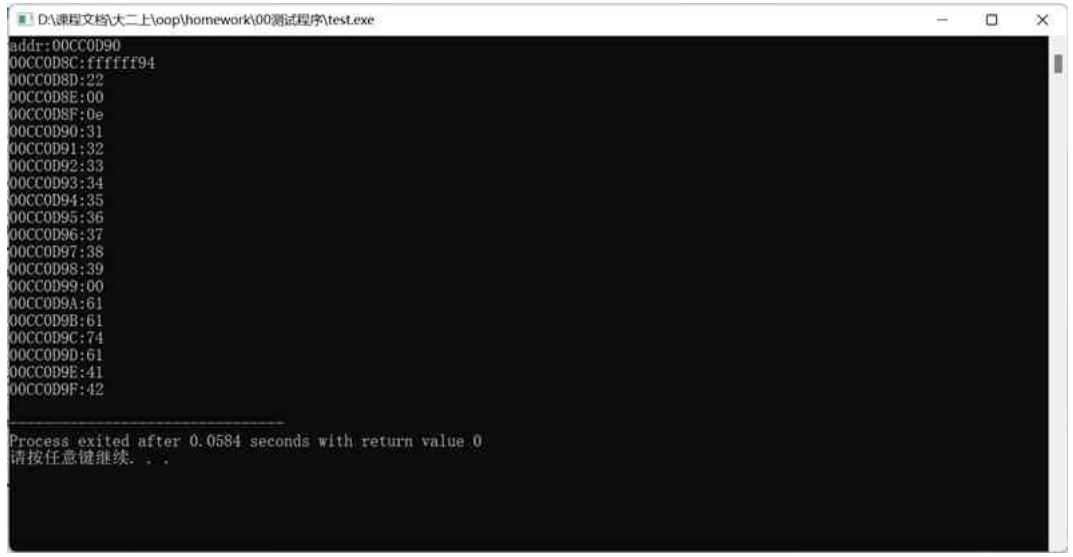



VS2022 x86/Release

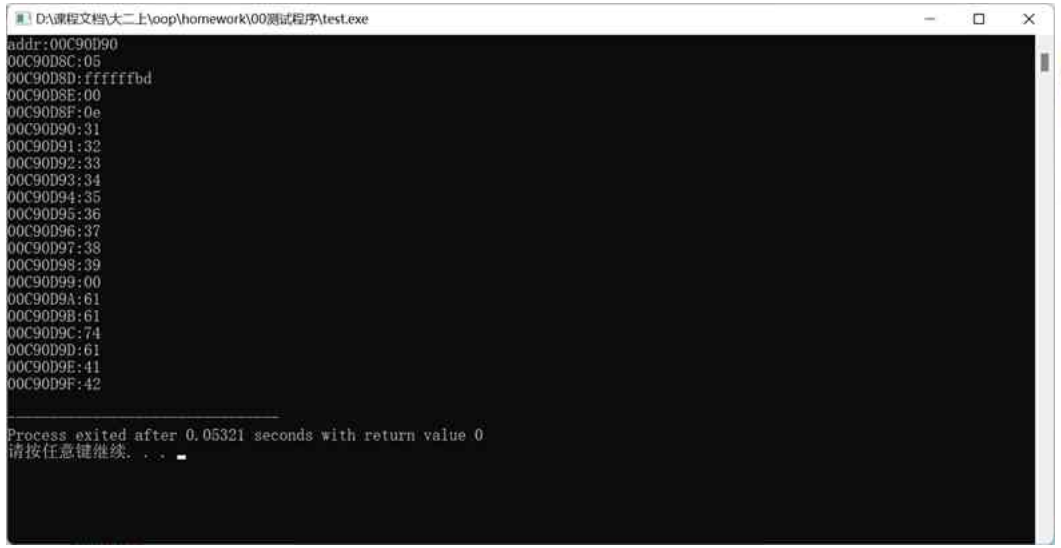
1、①②③全部注释



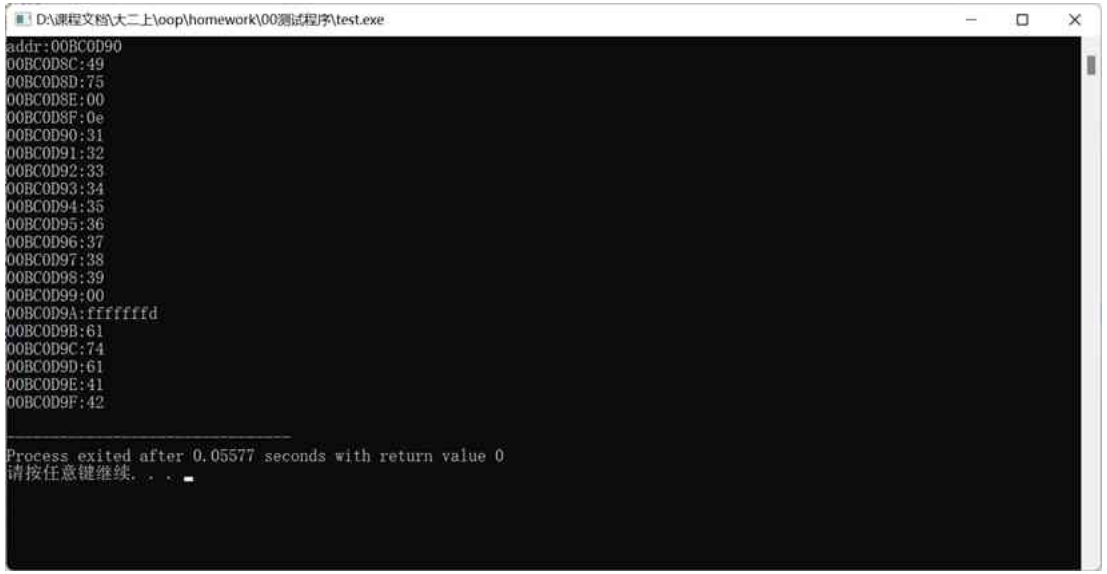

2、①放开，②③注释




3、①③放开，②注释

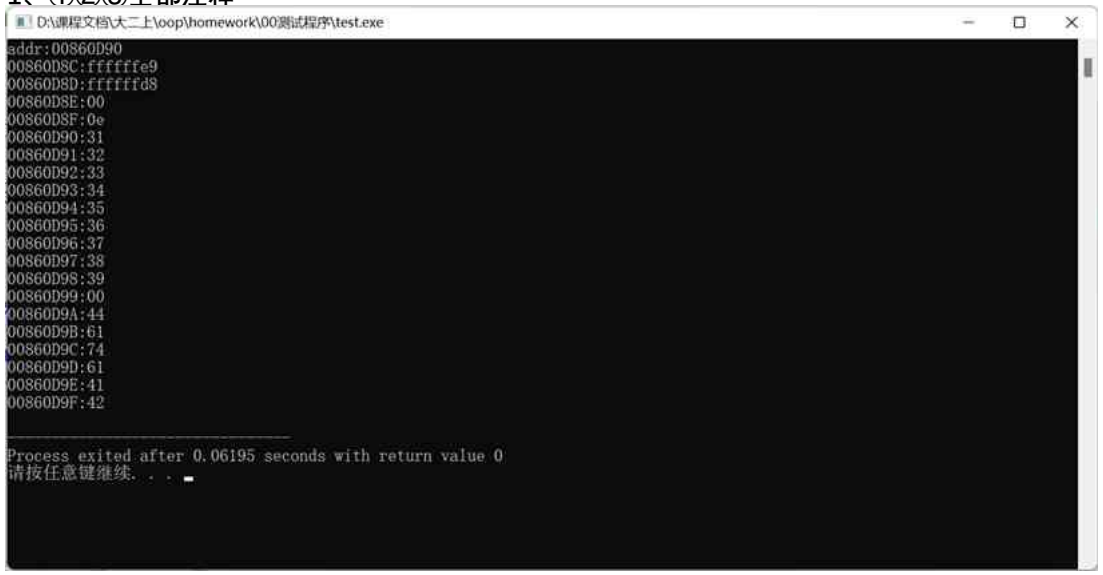



4、①②③全部放开

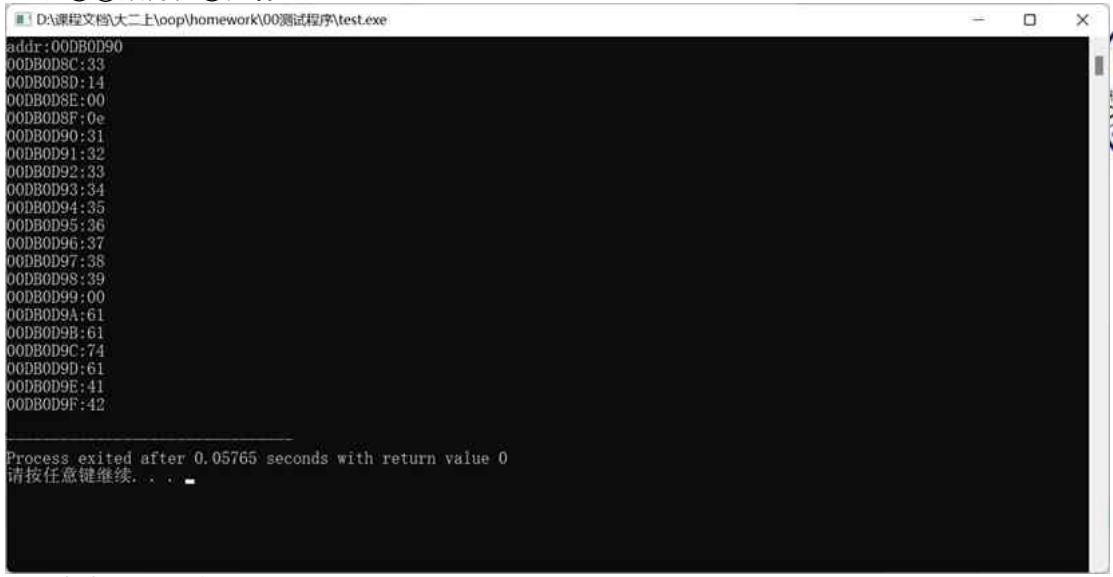



Dev 32bit-Debug

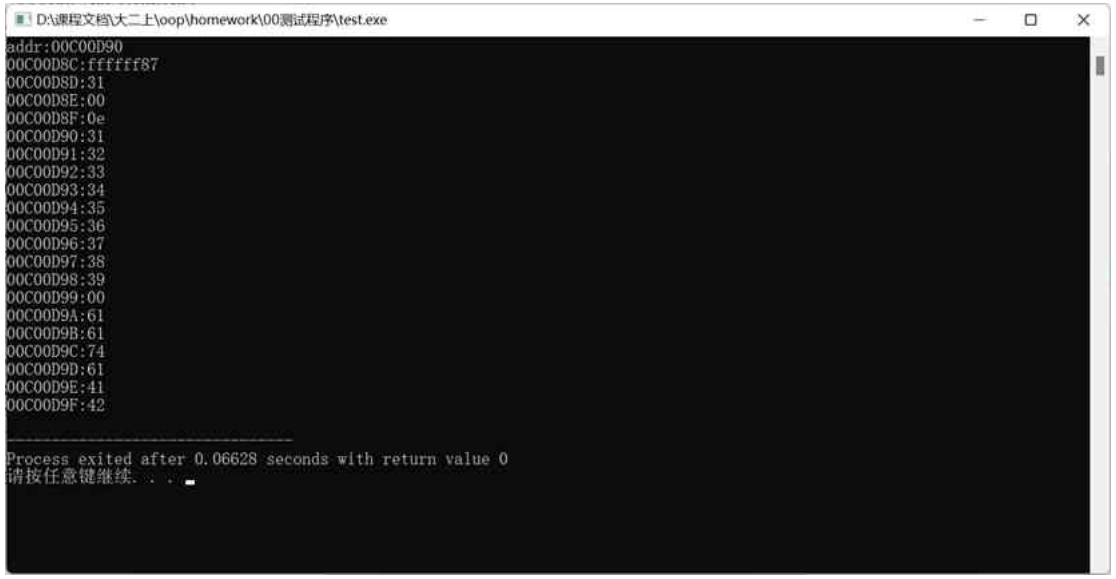
1、①②③全部注释



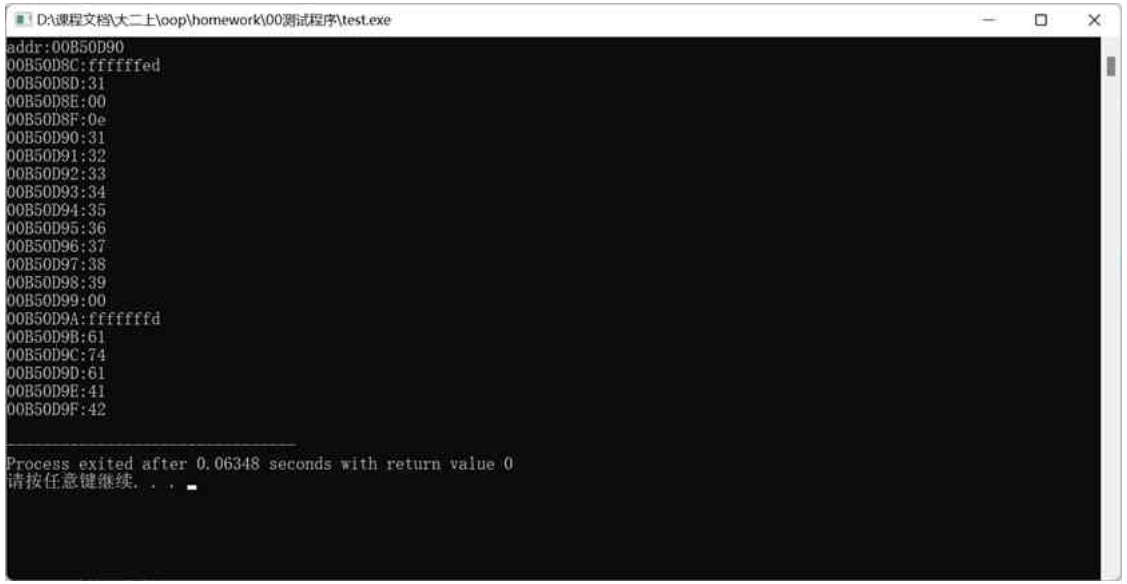

3、①③放开，②注释




2、①放开，②③注释




4、①②③全部放开




Dev 32bit-Release

1、①②③全部注释

```
[u2152041@oop ~]$ ./test
addr:0x60eeb0
0x60eeac:00
0x60eead:00
0x60eeae:00
0x60eeaf:00
0x60eeb0:31
0x60eeb1:32
0x60eeb2:33
0x60eeb3:34
0x60eeb4:35
0x60eeb5:36
0x60eeb6:37
0x60eeb7:38
0x60eeb8:39
0x60eeb9:00
0x60eeba:00
0x60eebb:00
0x60eebc:00
0x60eebd:00
0x60eebe:41
0x60eebf:42
```

2、①放开，②③注释

```
[u2152041@oop ~]$ ./test
addr:0x8f8eb0
0x8f8eac:00
0x8f8ead:00
0x8f8eae:00
0x8f8eaf:00
0x8f8eb0:31
0x8f8eb1:32
0x8f8eb2:33
0x8f8eb3:34
0x8f8eb4:35
0x8f8eb5:36
0x8f8eb6:37
0x8f8eb7:38
0x8f8eb8:39
0x8f8eb9:00
0x8f8eba:61
0x8f8ebb:00
0x8f8ebc:00
0x8f8ebd:00
0x8f8ebe:41
0x8f8ebf:42
```

3、①③放开，②注释

```
[u2152041@oop ~]$ ./test
addr:0x2165eb0
0x2165eac:00
0x2165ead:00
0x2165eae:00
0x2165eaf:00
0x2165eb0:31
0x2165eb1:32
0x2165eb2:33
0x2165eb3:34
0x2165eb4:35
0x2165eb5:36
0x2165eb6:37
0x2165eb7:38
0x2165eb8:39
0x2165eb9:00
0x2165eba:61
0x2165ebb:00
0x2165ebc:00
0x2165ebd:00
0x2165ebe:41
0x2165ebf:42
```

4、①②③全部放开

```
[u2152041@oop ~]$ ./test
addr:0x1f5eeb0
0x1f5eeac:00
0x1f5eead:00
0x1f5eeae:00
0x1f5eeaf:00
0x1f5eeb0:31
0x1f5eeb1:32
0x1f5eeb2:33
0x1f5eeb3:34
0x1f5eeb4:35
0x1f5eeb5:36
0x1f5eeb6:37
0x1f5eeb7:38
0x1f5eeb8:39
0x1f5eeb9:00
0x1f5eeba:fffffffd
0x1f5eebb:00
0x1f5eebc:00
0x1f5eebd:00
0x1f5eebe:41
0x1f5eebf:42
```

# §.关于动态内存申请后越界访问的深度讨论

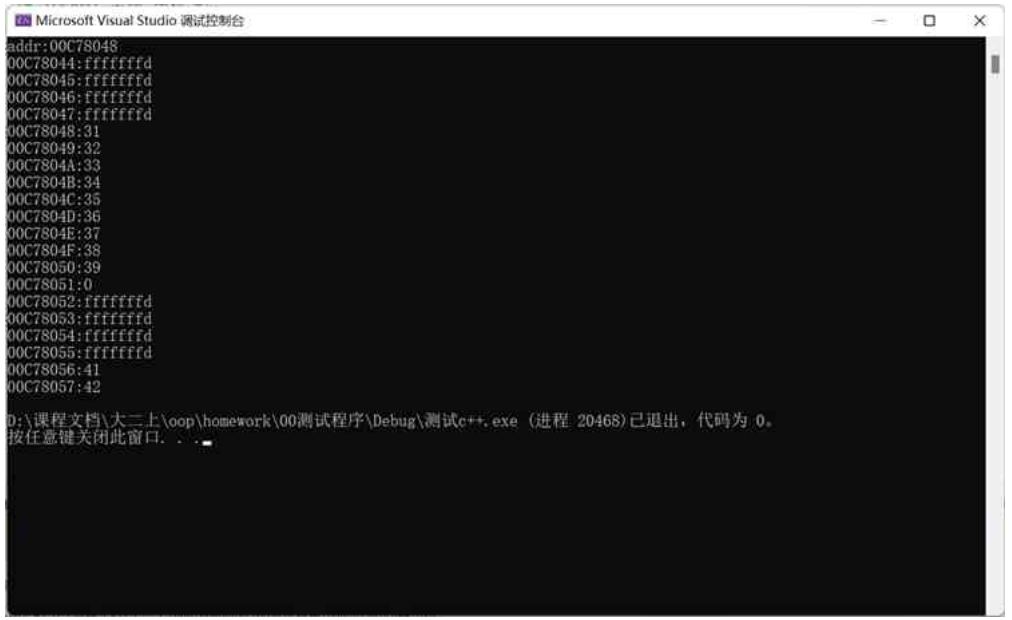
★ 如何判断动态申请越界（C++方式，注意源程序后缀为.cpp）

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char *p;
    p = new(nothrow) char[10];
    if (p == NULL)
        return -1;
    strcpy(p, "123456789");
①   p[10] = 'a';     //此句越界
    p[14] = 'A';     //此句越界
    p[15] = 'B';     //此句越界
②   p[10] = '\xfd'; //此句越界
    cout << "addr:" << hex << (void *)(p) << endl;
    for (int i = -4; i < 16; i++) //注意，只有0-9是合理范围，其余都是越界读
        cout << hex << (void *)(p + i) << ":" << int(p[i]) << endl;
③   delete[]p;

    return 0;
}
```

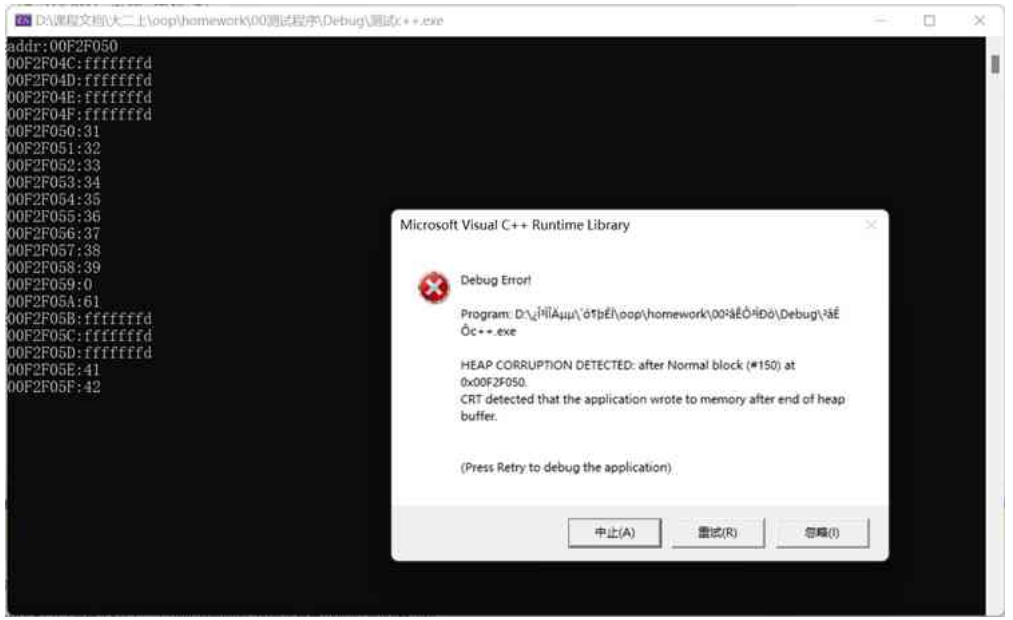
在VS2022的x86/Debug模式下运行：
1、①②③全部注释，观察运行结果
2、①放开，②③注释，观察运行结果
3、①③放开，②注释，观察运行结果
4、①②③全部放开，观察运行结果
结论：VS的Debug模式是如何判断
动态申请内存访问越界的？

再观察下面四种环境下的运行结果：
VS2022 x86/Release
Dev 32bit-Debug
Dev 32bit-Release
Linux
每种讨论的结果可截图+文字说明，
如果几种环境的结果一致，用一个
环境的截图+文字说明即可(可加页)

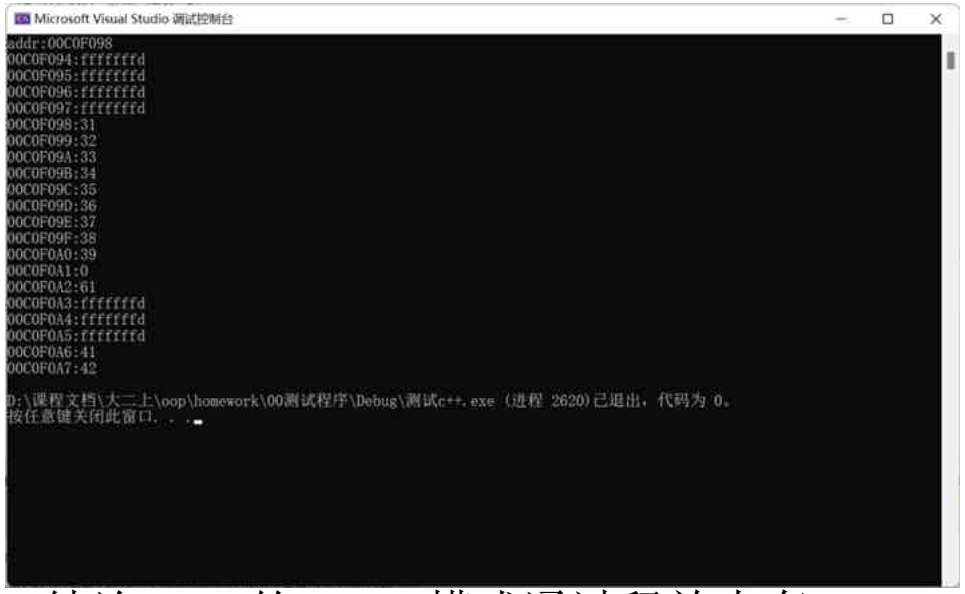
1、①②③全部注释



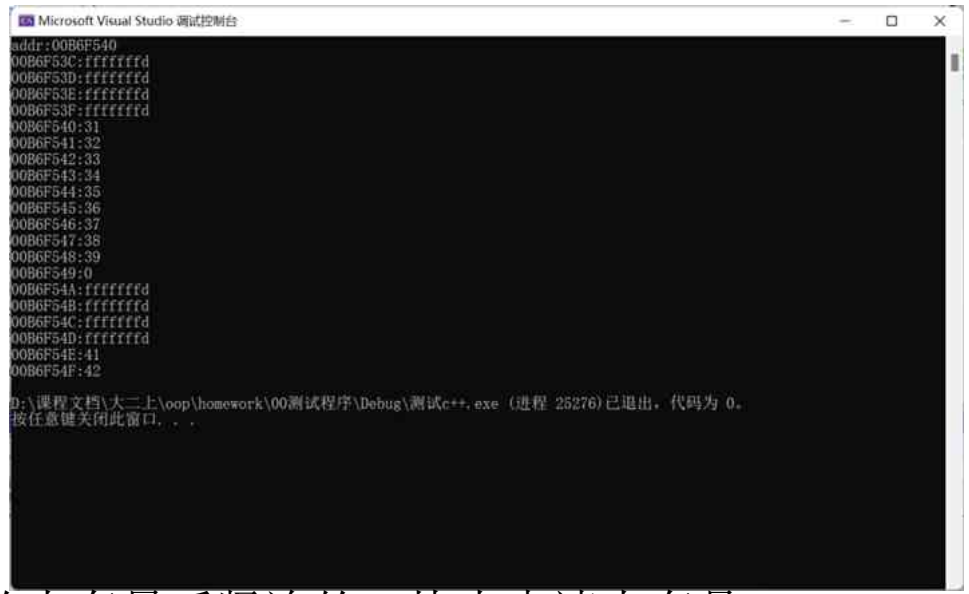

3、①③放开，②注释




2、①放开，②③注释

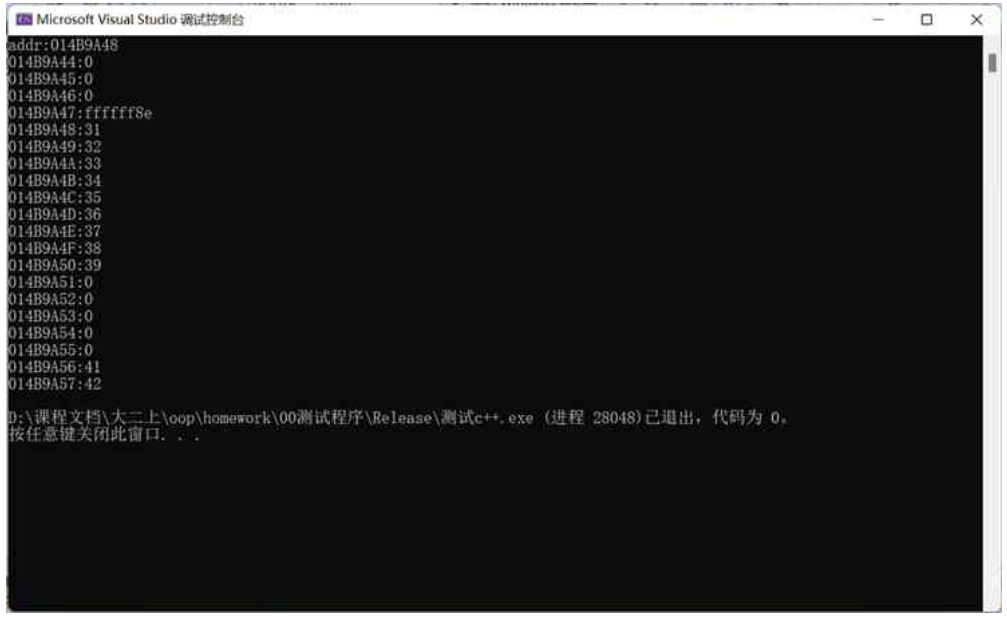



4、①②③全部放开



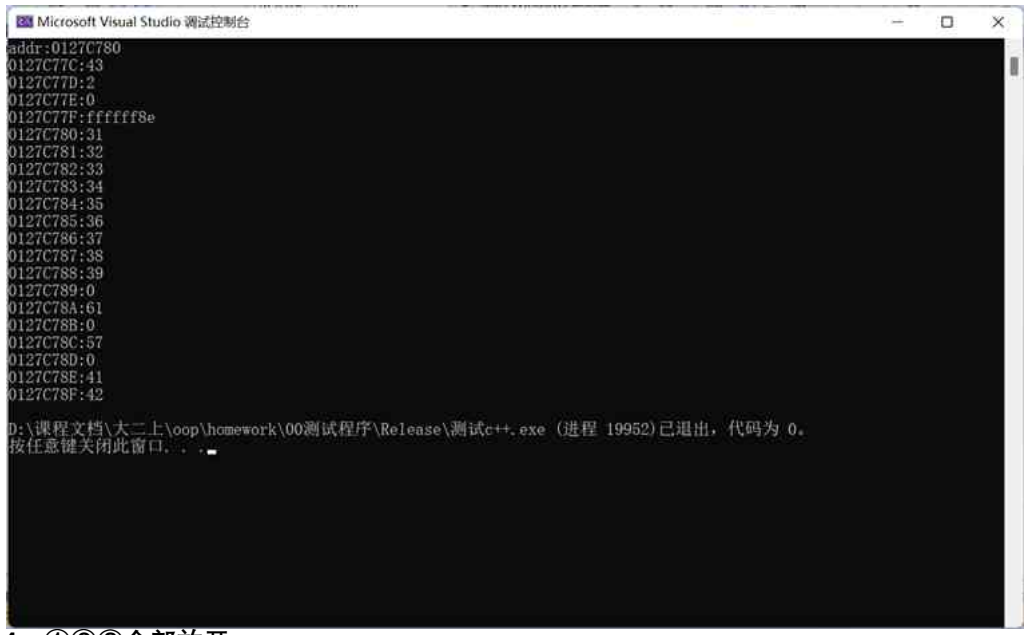

结论：VS的Debug模式通过释放内存(delete)时与申请的内存最后紧连的一块未申请内存是否有改变来进行判定内存访问越界　　VS2022的x86/Debug

1、①②③全部注释

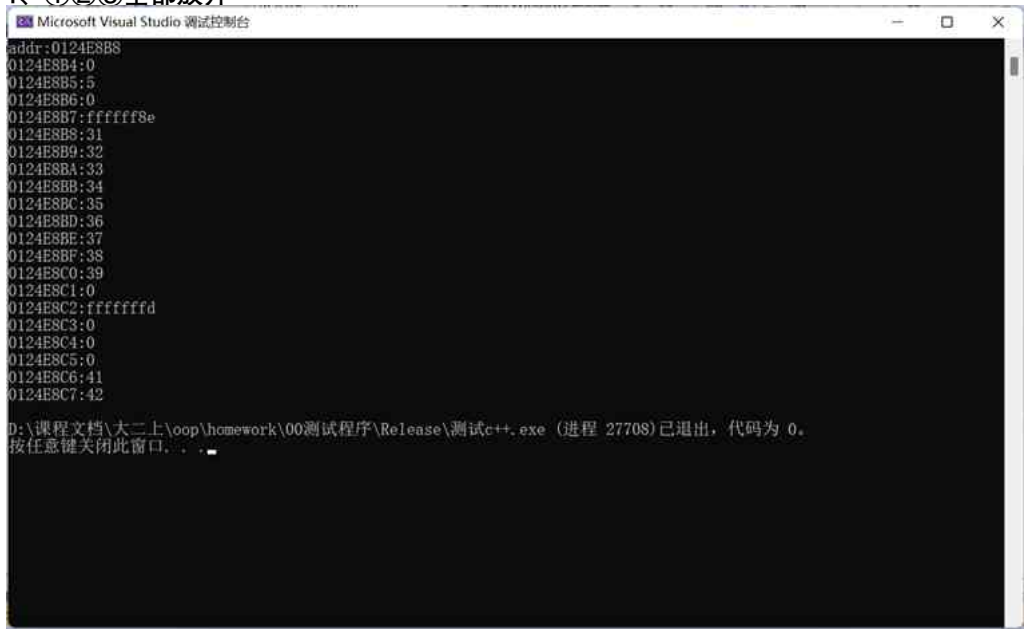



2、①放开，②③注释


3、①③放开，②注释

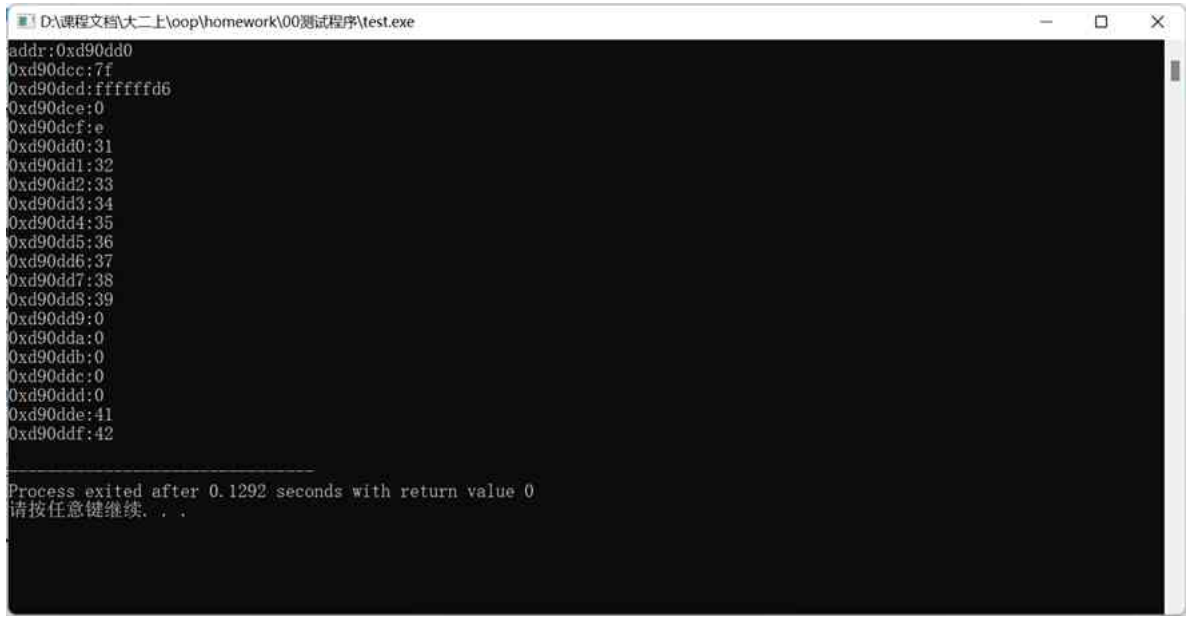



4、①②③全部放开

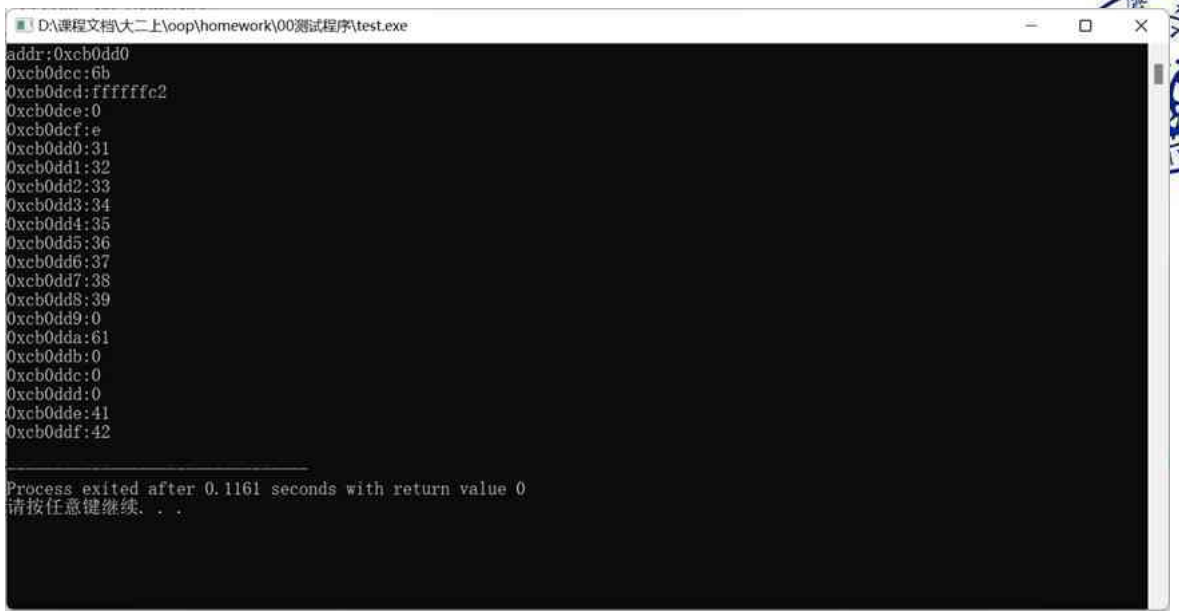



VS2022 x86/Release

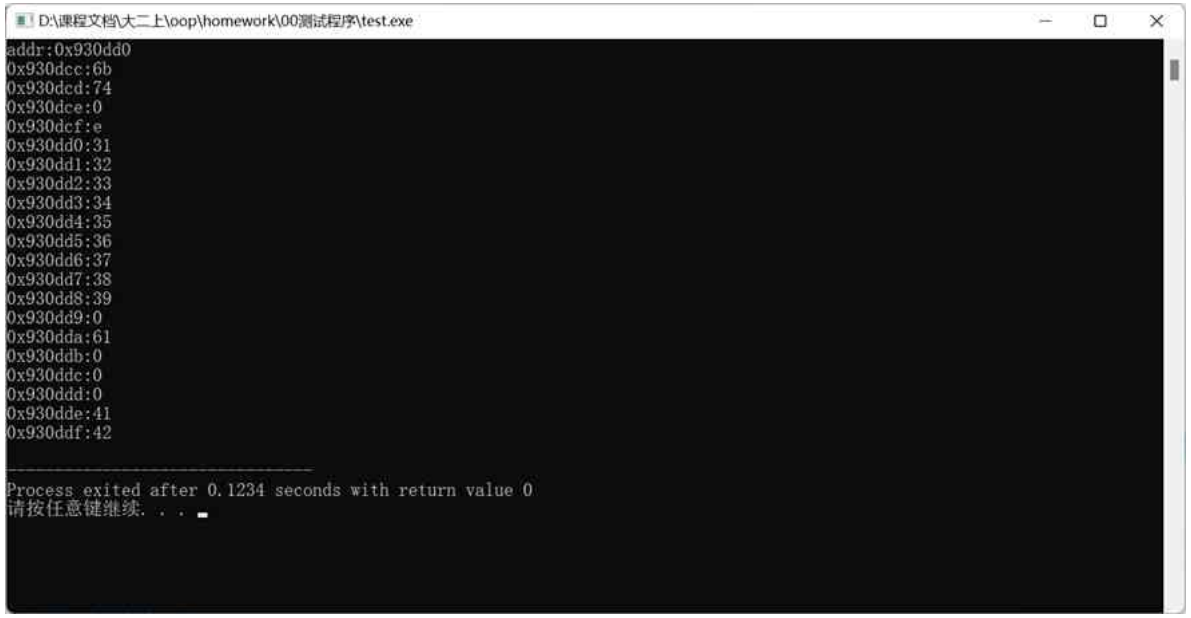
1、①②③全部注释



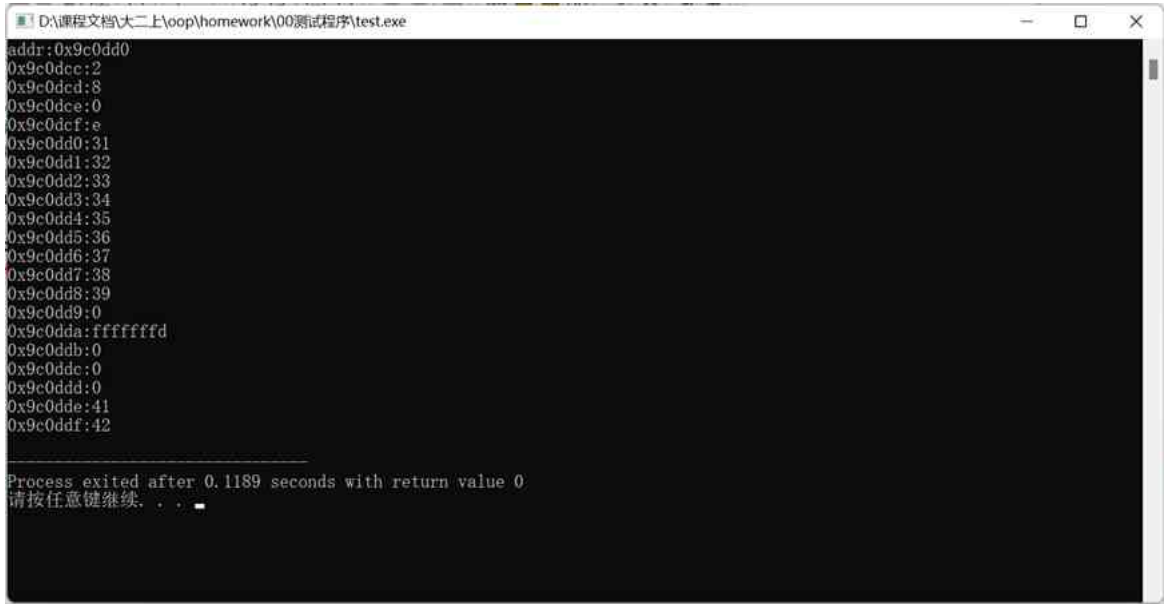

3、①③放开，②注释




2、①放开，②③注释

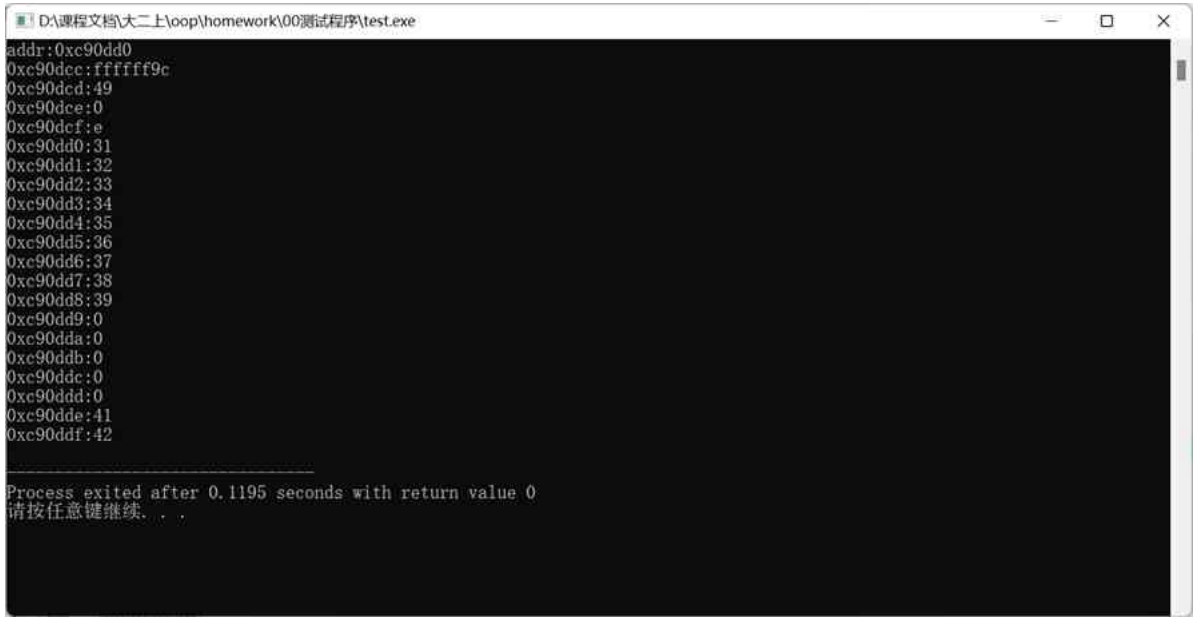



4、①②③全部放开

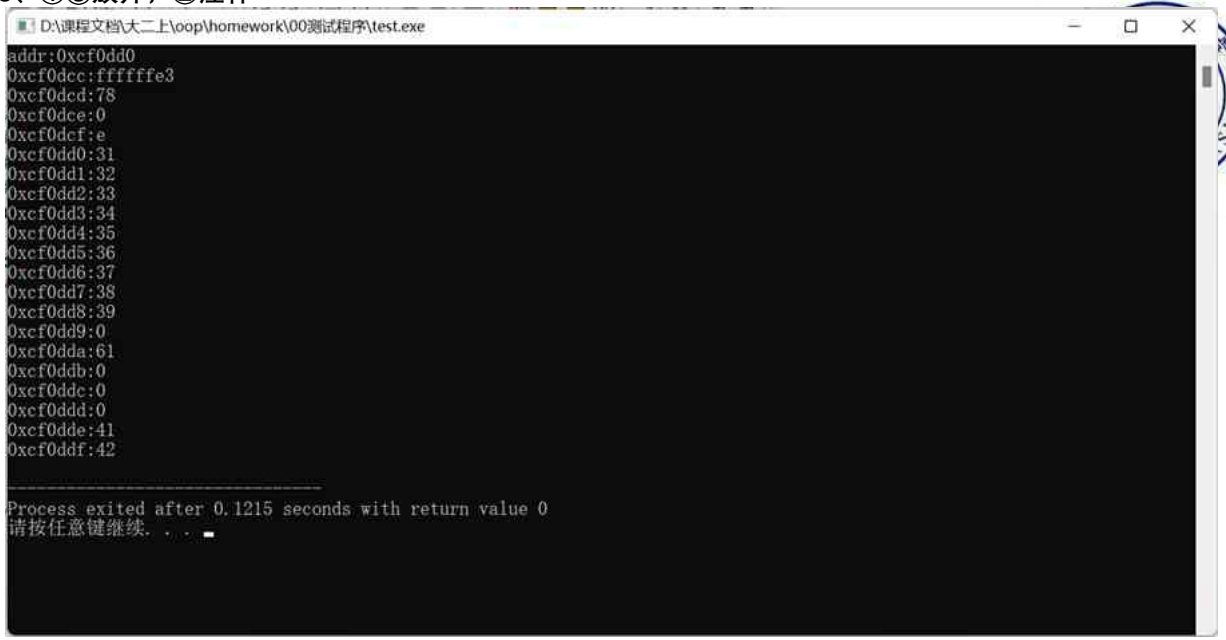



Dev 32bit-Debug

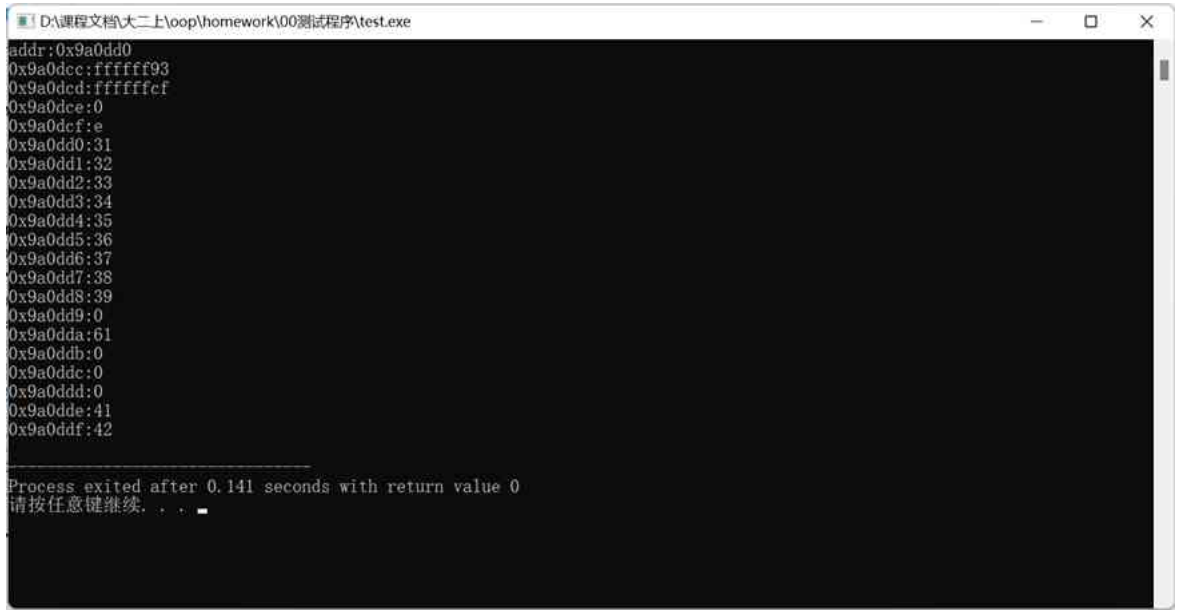
1、①②③全部注释



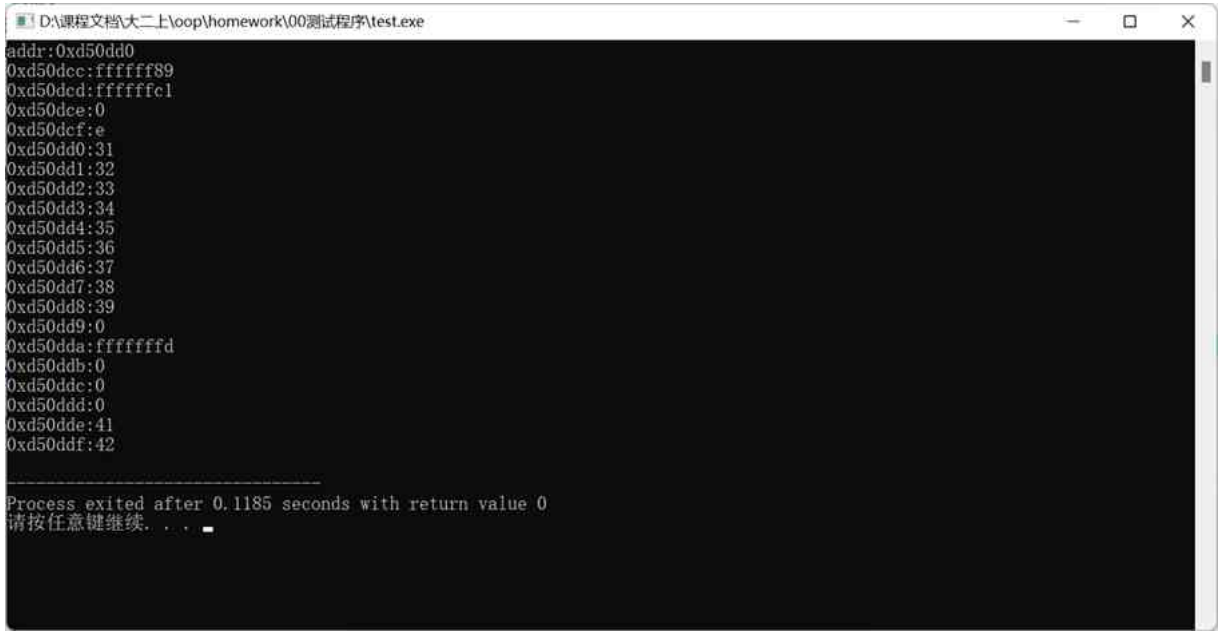

3、①③放开，②注释




2、①放开，②③注释




4、①②③全部放开




Dev 32bit-Release

1、①②③全部注释

```
[u2152041@oop ~]$ ./test
addr:0xc69eb0
0xc69eac:0
0xc69ead:0
0xc69eae:0
0xc69eaf:0
0xc69eb0:31
0xc69eb1:32
0xc69eb2:33
0xc69eb3:34
0xc69eb4:35
0xc69eb5:36
0xc69eb6:37
0xc69eb7:38
0xc69eb8:39
0xc69eb9:0
0xc69eba:61
0xc69ebb:0
0xc69ebc:0
0xc69ebd:0
0xc69ebe:0
0xc69ebf:42
```

2、①放开，②③注释

```
[u2152041@oop ~]$ ./test
addr:0x11f9eb0
0x11f9eac:0
0x11f9ead:0
0x11f9eae:0
0x11f9eaf:0
0x11f9eb0:31
0x11f9eb1:32
0x11f9eb2:33
0x11f9eb3:34
0x11f9eb4:35
0x11f9eb5:36
0x11f9eb6:37
0x11f9eb7:38
0x11f9eb8:39
0x11f9eb9:0
0x11f9eba:61
0x11f9ebb:0
0x11f9ebc:0
0x11f9ebd:0
0x11f9ebe:41
0x11f9ebf:42
```

3、①③放开，②注释

```
[u2152041@oop ~]$ ./test
addr:0x16f5eb0
0x16f5eac:0
0x16f5ead:0
0x16f5eae:0
0x16f5eaf:0
0x16f5eb0:31
0x16f5eb1:32
0x16f5eb2:33
0x16f5eb3:34
0x16f5eb4:35
0x16f5eb5:36
0x16f5eb6:37
0x16f5eb7:38
0x16f5eb8:39
0x16f5eb9:0
0x16f5eba:61
0x16f5ebb:0
0x16f5ebc:0
0x16f5ebd:0
0x16f5ebe:41
0x16f5ebf:42
```

4、①②③全部放开

```
[u2152041@oop ~]$ ./test
addr:0x1ff5eb0
0x1ff5eac:0
0x1ff5ead:0
0x1ff5eae:0
0x1ff5eaf:0
0x1ff5eb0:31
0x1ff5eb1:32
0x1ff5eb2:33
0x1ff5eb3:34
0x1ff5eb4:35
0x1ff5eb5:36
0x1ff5eb6:37
0x1ff5eb7:38
0x1ff5eb8:39
0x1ff5eb9:0
0x1ff5eba:fffffffd
0x1ff5ebb:0
0x1ff5ebc:0
0x1ff5ebd:0
0x1ff5ebe:41
0x1ff5ebf:42
```

# §.关于动态内存申请后越界访问的深度讨论

★ 如何判断普通数组的越界访问（C++方式，注意源程序后缀为.cpp）

```
/* 2152041 王浩 计科 */
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    char p[10];
    strcpy(p, "123456789");
① p[10] = 'a';     //此句越界
    p[14] = 'A';     //此句越界
    p[15] = 'B';     //此句越界
② p[10] = '\xfd'; //此句越界
    cout << "addr:" << hex << (void *)(p) << endl;
    for (int i = -4; i < 16; i++) //注意，只有0-9是合理范围，其余都是越界读
        cout << hex << (void *)(p + i) << ":" << int(p[i]) << endl;

    return 0;
}
```

数组为char a[10]形式

在理解P.1/P.2的情况下，自行构造相似的程序，来观察数组越界后的内存表现，并验证与动态申请是否相似

要求：

1、数组用 char a[10]; 形式

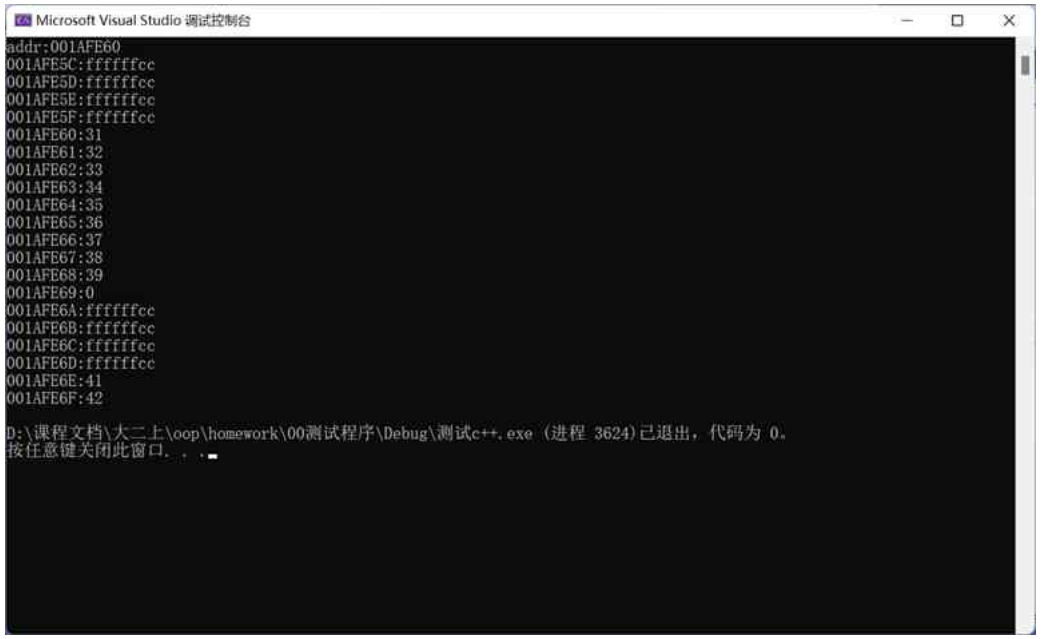
2、数组用 int a[10]; 形式

3、测试程序在下面五种环境下运行

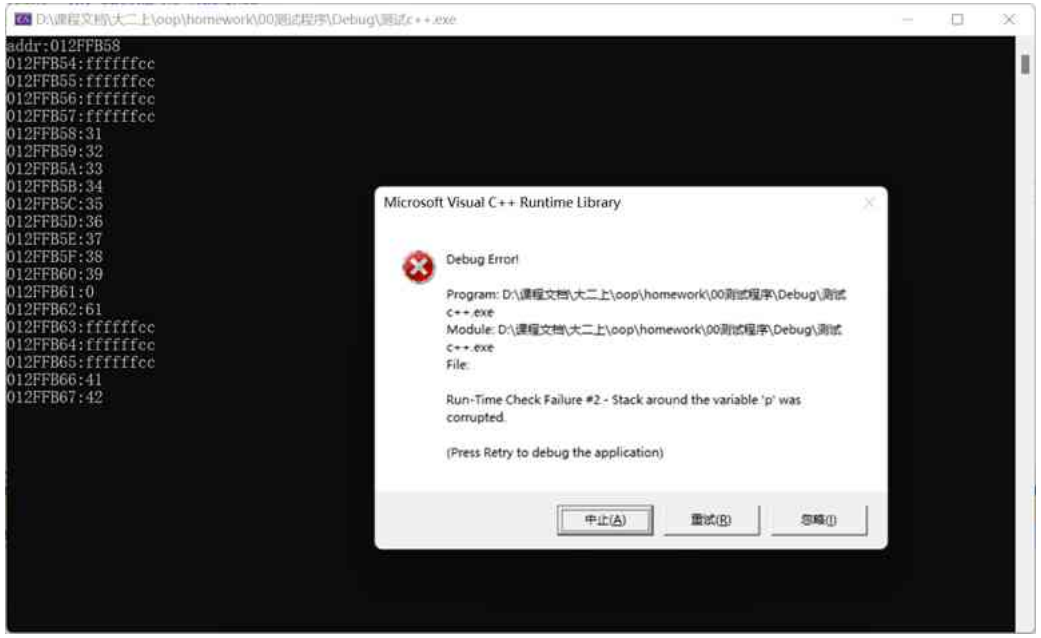
- VS2022 x86/Debug
- VS2022 x86Release
- Dev 32bit-Debug
- Dev 32bit-Release
- Linux

4、每种讨论的结果可截图+文字说明，如果几种环境的结果一致，用一个环境的截图+文字说明即可(可加页)

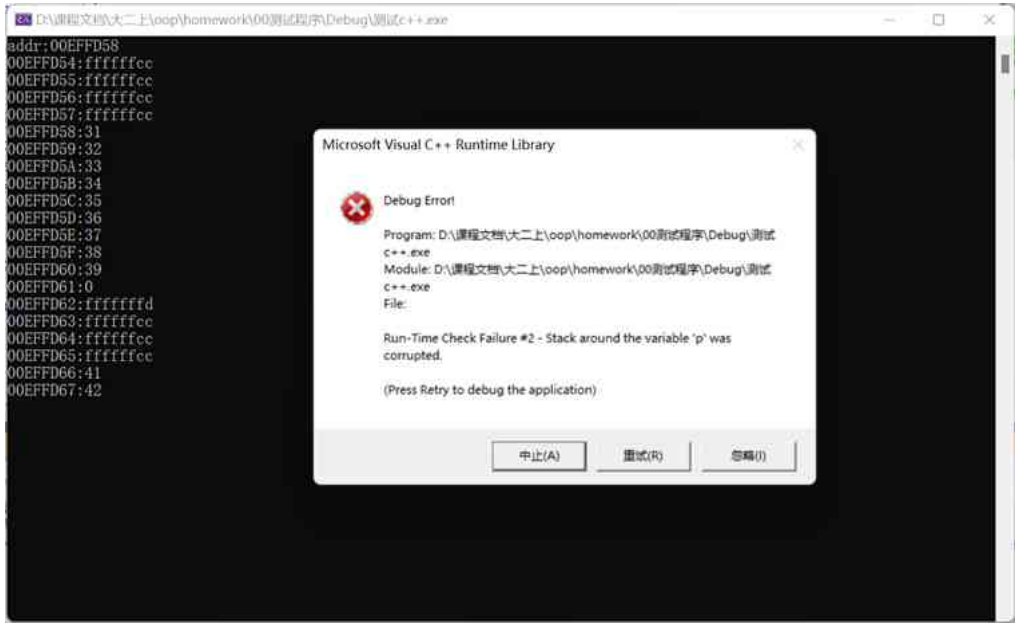
1、①②全部注释



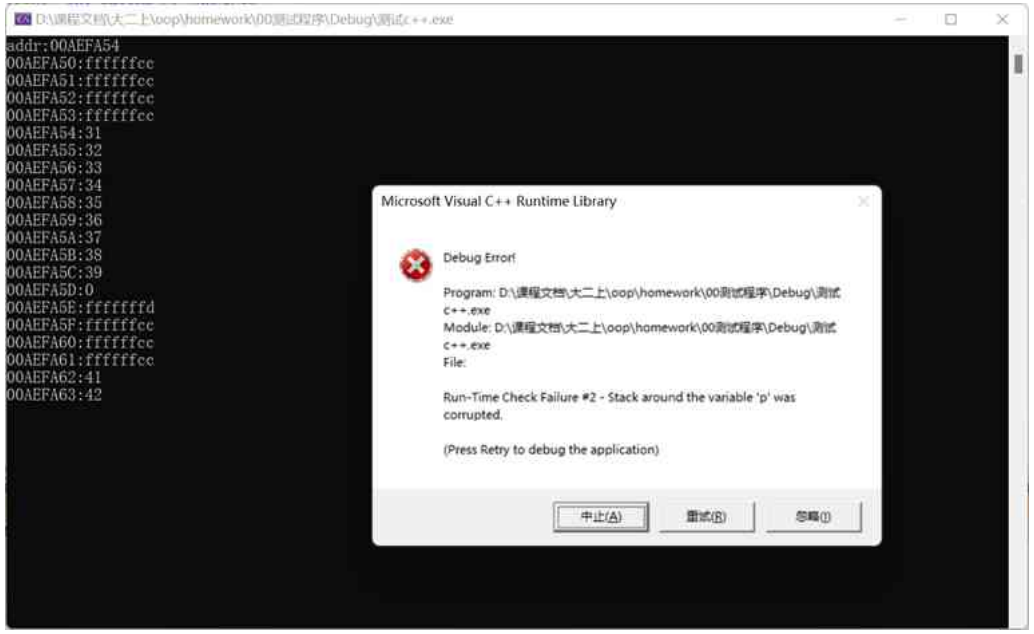

2、①放开，②注释




3、 ②放开， ①注释

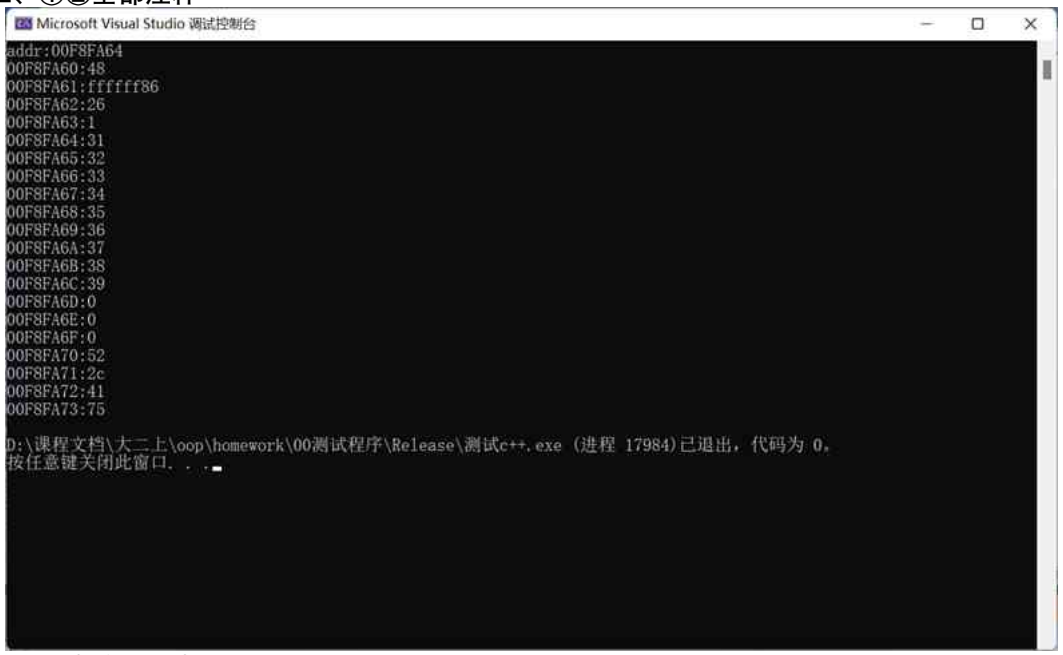



4、①②全部放开

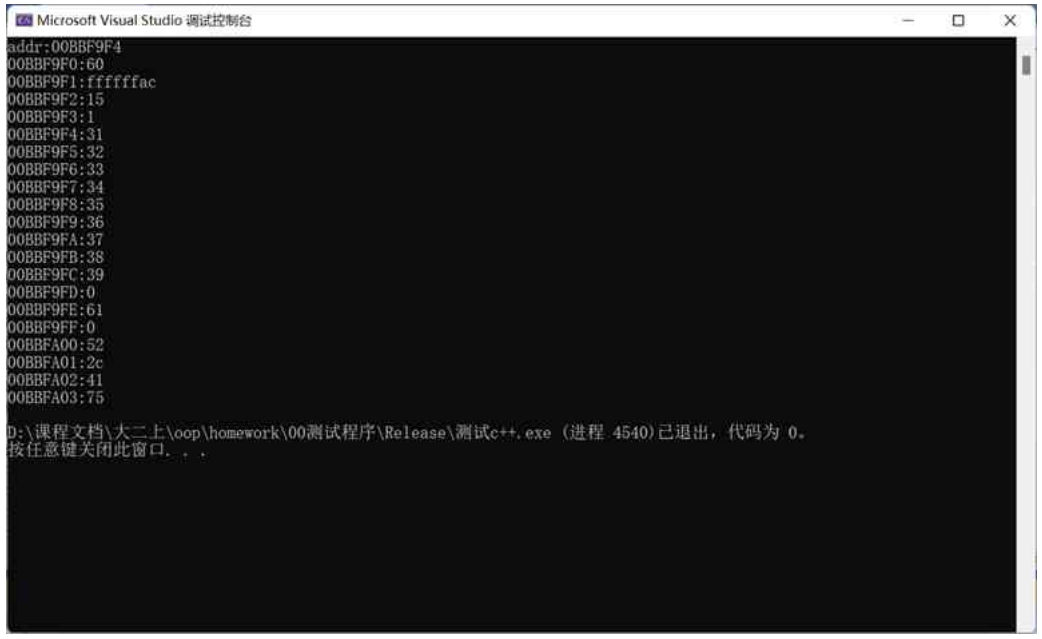



VS2022 x86/Debug

1、①②全部注释

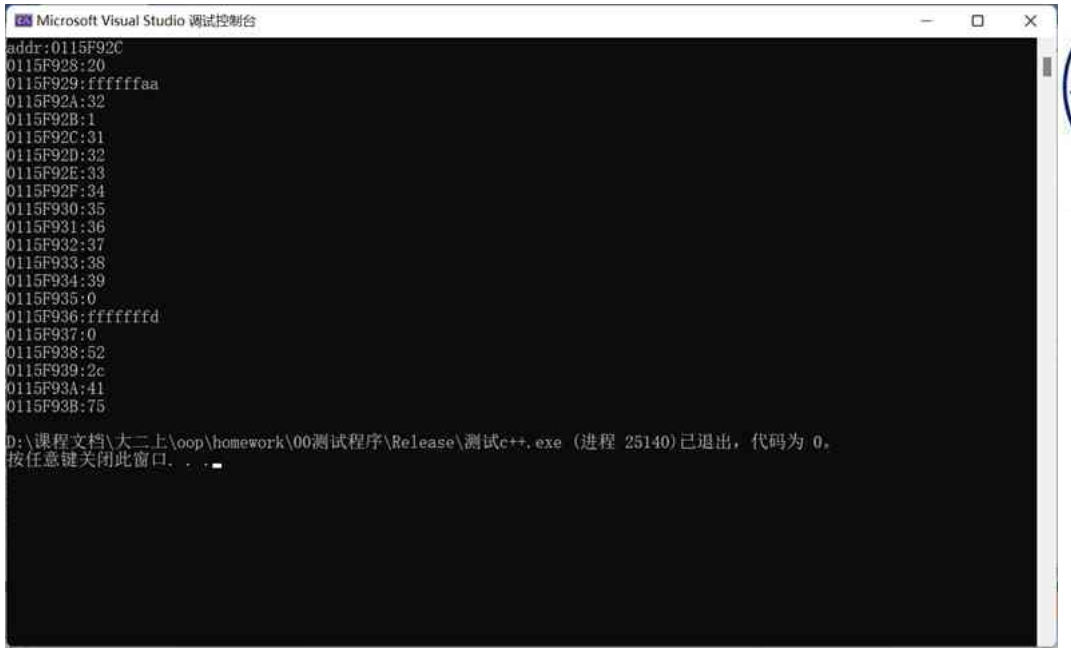



2、①放开，②注释

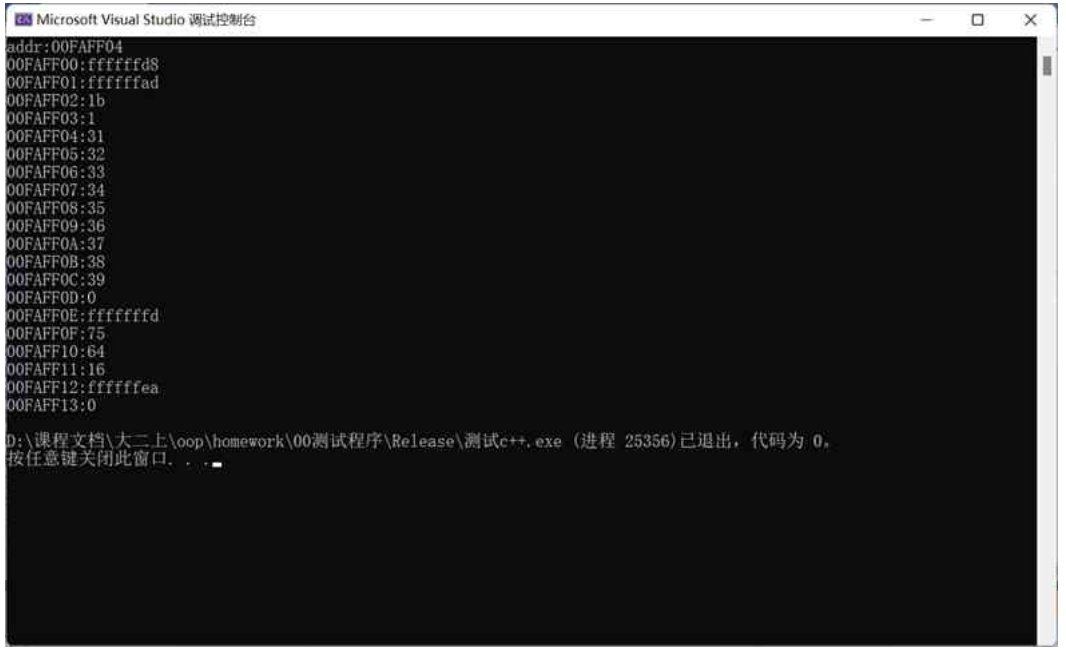



VS2022 x86/Release

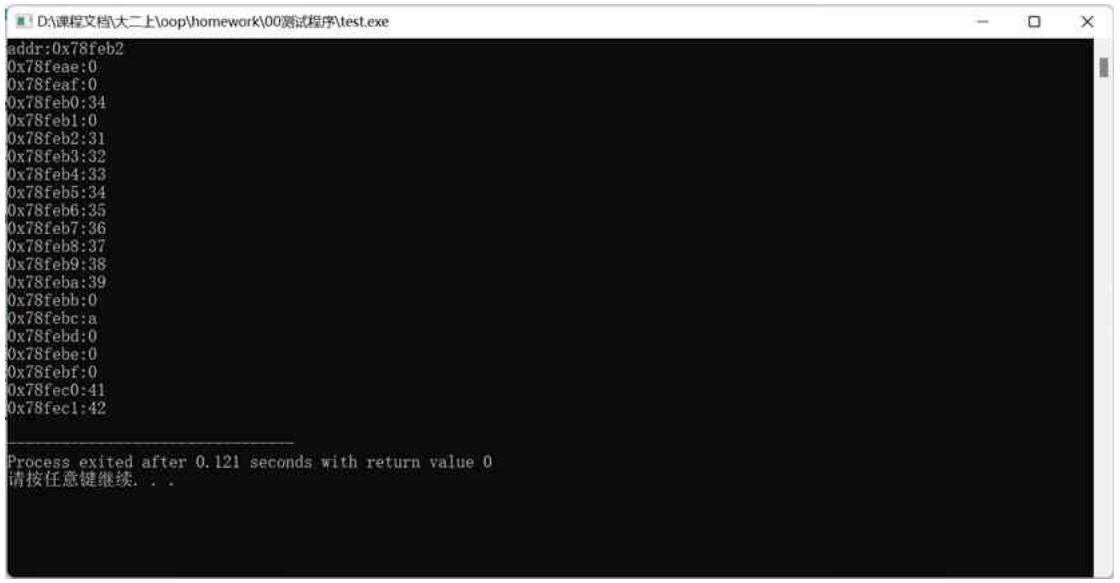
3、 ②放开， ①注释



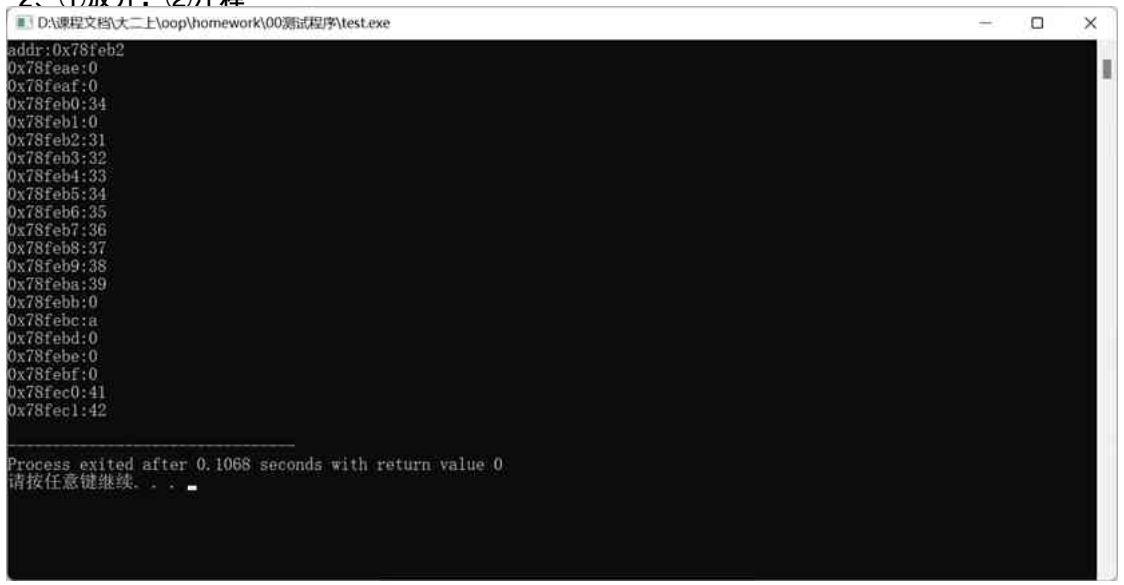

4、①②全部放开



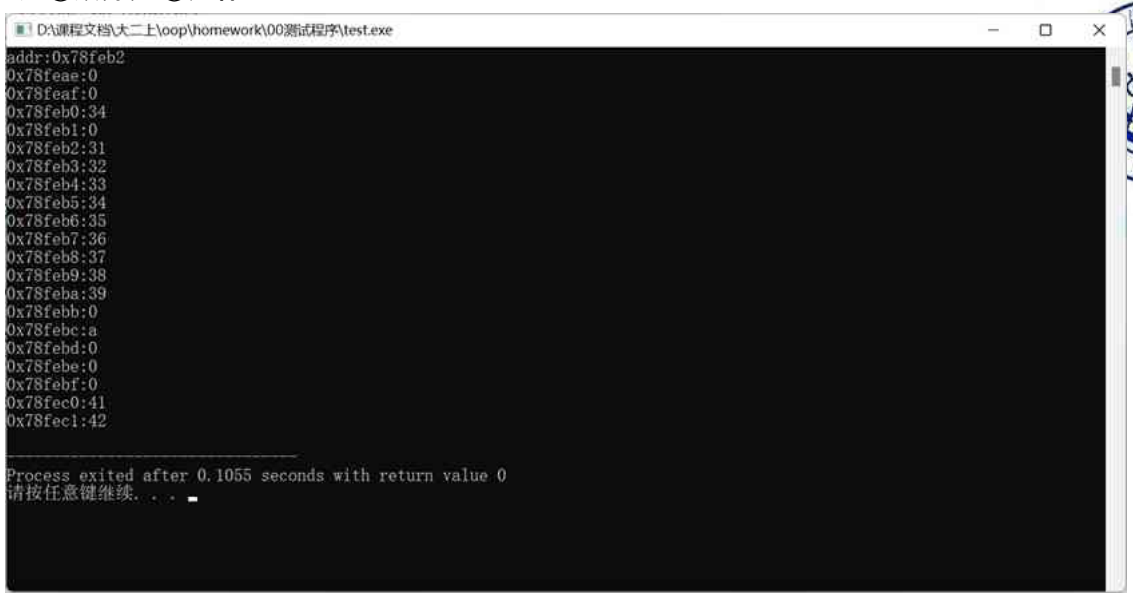
1、①②全部注释



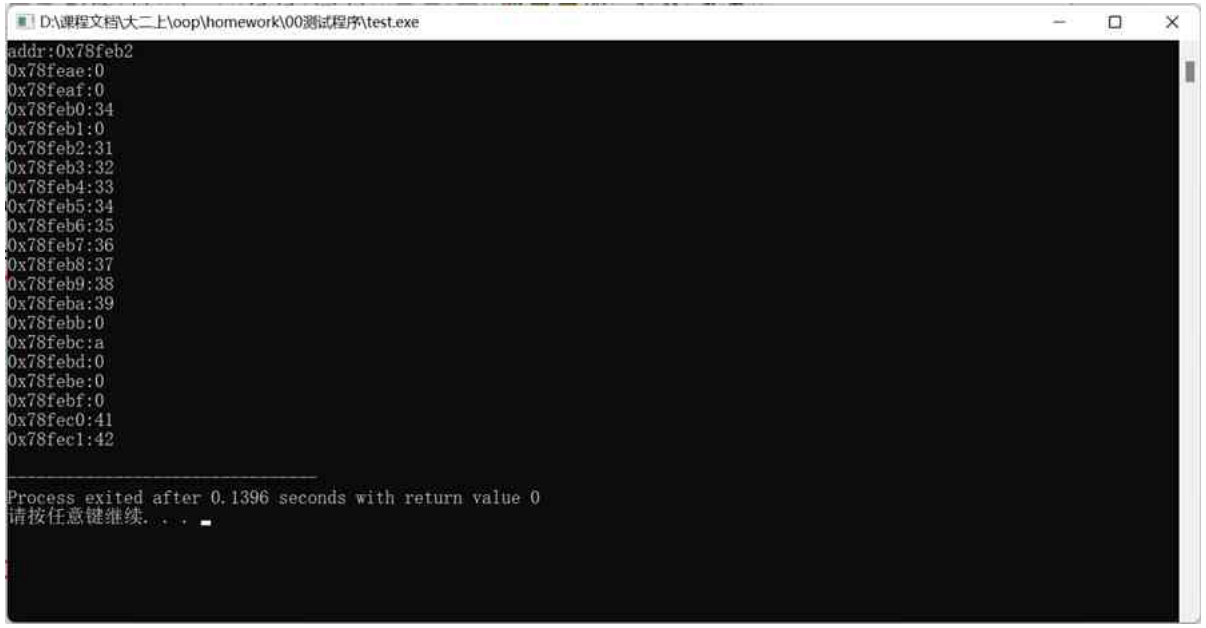

2、①放开，②注释




3、②放开，①注释

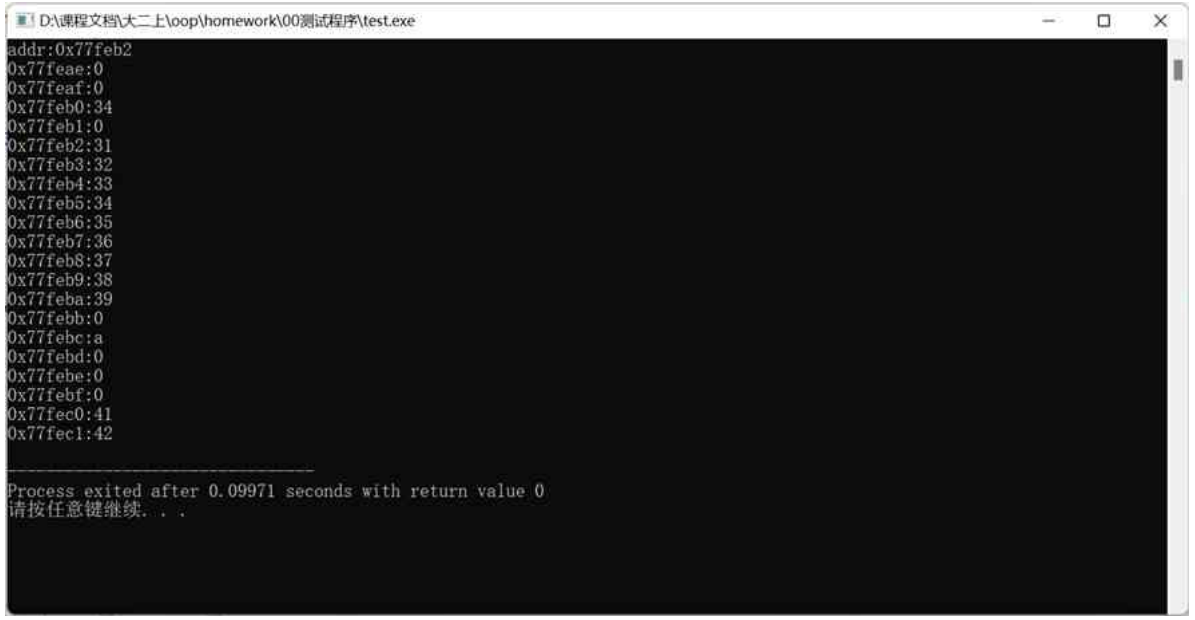



4、①②全部放开

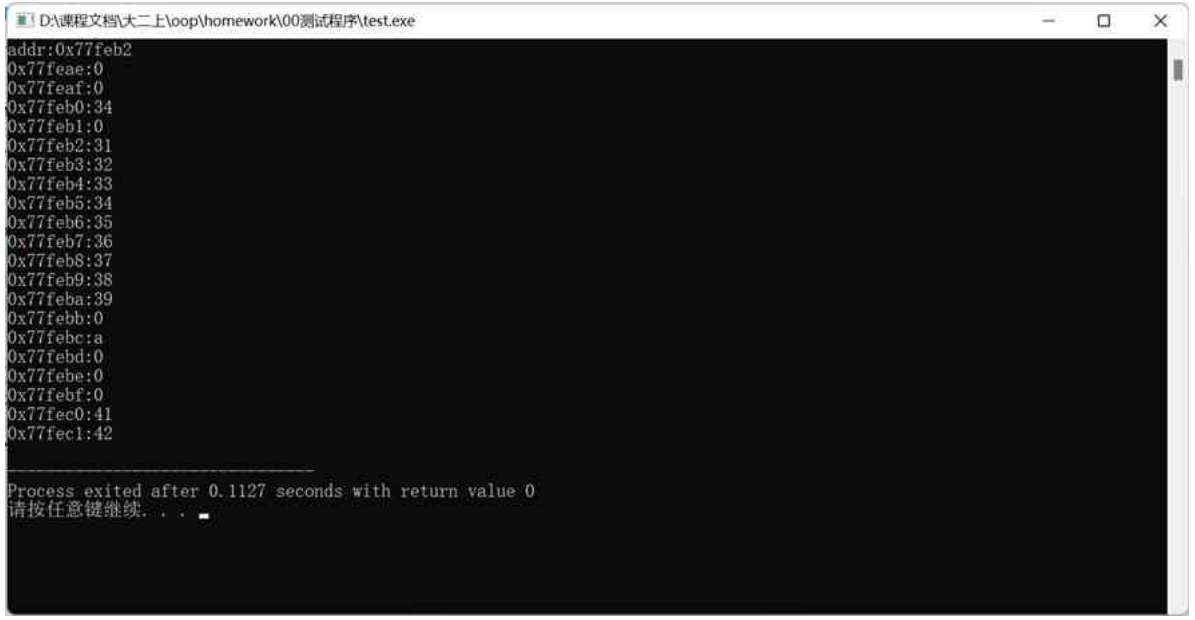



Dev 32bit-Debug

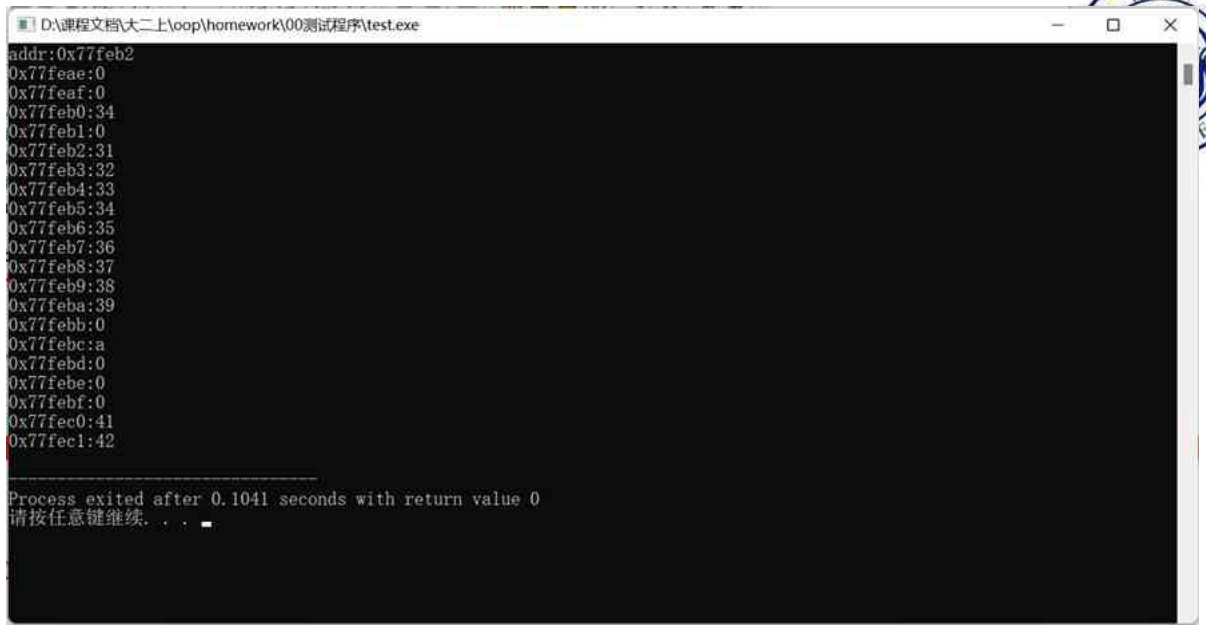
1、①②全部注释



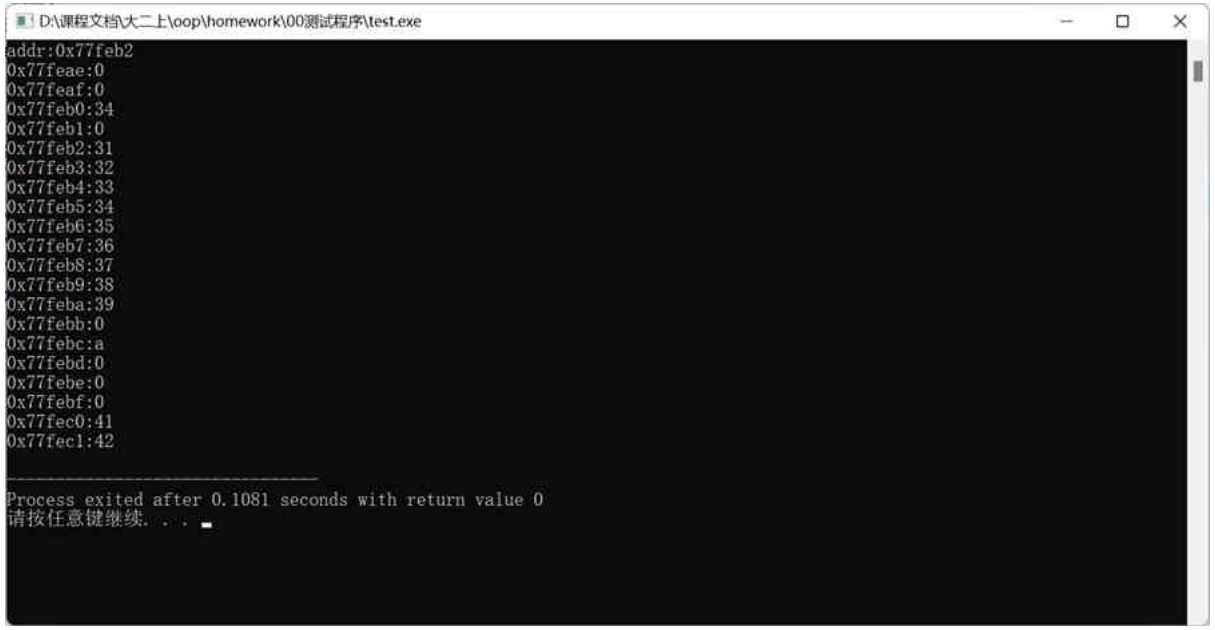

2、①放开，②注释




3、 ②放开， ①注释




4、①②全部放开




Dev 32bit-Release

1、①②全部注释

```
[u2152041@oop ~]$ ./test
addr:0x7ffc0bc38612
0x7ffc0bc3860e:0
0x7ffc0bc3860f:0
0x7ffc0bc38610:0
0x7ffc0bc38611:0
0x7ffc0bc38612:31
0x7ffc0bc38613:32
0x7ffc0bc38614:33
0x7ffc0bc38615:34
0x7ffc0bc38616:35
0x7ffc0bc38617:36
0x7ffc0bc38618:37
0x7ffc0bc38619:38
0x7ffc0bc3861a:39
0x7ffc0bc3861b:0
0x7ffc0bc3861c:a
0x7ffc0bc3861d:0
0x7ffc0bc3861e:0
0x7ffc0bc3861f:0
0x7ffc0bc38620:41
0x7ffc0bc38621:42
```

2、①放开，②注释

```
[u2152041@oop ~]$ ./test
addr:0x7ffef727dff2
0x7ffef727dfee:0
0x7ffef727dfef:0
0x7ffef727dff0:0
0x7ffef727dff1:0
0x7ffef727dff2:31
0x7ffef727dff3:32
0x7ffef727dff4:33
0x7ffef727dff5:34
0x7ffef727dff6:35
0x7ffef727dff7:36
0x7ffef727dff8:37
0x7ffef727dff9:38
0x7ffef727dffa:39
0x7ffef727dffb:0
0x7ffef727dffc:a
0x7ffef727dffd:0
0x7ffef727dffe:0
0x7ffef727dfff:0
0x7ffef727e000:41
0x7ffef727e001:42
```

3、 ② 放开， ①注释

```
[u2152041@oop ~]$ ./test
addr:0x7fffd6cf5c62
0x7fffd6cf5c5e:0
0x7fffd6cf5c5f:0
0x7fffd6cf5c60:0
0x7fffd6cf5c61:0
0x7fffd6cf5c62:31
0x7fffd6cf5c63:32
0x7fffd6cf5c64:33
0x7fffd6cf5c65:34
0x7fffd6cf5c66:35
0x7fffd6cf5c67:36
0x7fffd6cf5c68:37
0x7fffd6cf5c69:38
0x7fffd6cf5c6a:39
0x7fffd6cf5c6b:0
0x7fffd6cf5c6c:a
0x7fffd6cf5c6d:0
0x7fffd6cf5c6e:0
0x7fffd6cf5c6f:0
0x7fffd6cf5c70:41
0x7fffd6cf5c71:42
```

4、①②全部放开

```
[u2152041@oop ~]$ ./test
addr:0x7ffe0a8e1642
0x7ffe0a8e163e:0
0x7ffe0a8e163f:0
0x7ffe0a8e1640:0
0x7ffe0a8e1641:0
0x7ffe0a8e1642:31
0x7ffe0a8e1643:32
0x7ffe0a8e1644:33
0x7ffe0a8e1645:34
0x7ffe0a8e1646:35
0x7ffe0a8e1647:36
0x7ffe0a8e1648:37
0x7ffe0a8e1649:38
0x7ffe0a8e164a:39
0x7ffe0a8e164b:0
0x7ffe0a8e164c:a
0x7ffe0a8e164d:0
0x7ffe0a8e164e:0
0x7ffe0a8e164f:0
0x7ffe0a8e1650:41
0x7ffe0a8e1651:42
```

Linux

# §.关于动态内存申请后越界访问的深度讨论

★ 如何判断普通数组的越界访问（C++方式，注意源程序后缀为.cpp）

```
/* 2152041 王浩 计科 */
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    int p[10] = { 0,1,2,3,4,5,6,7,8,9 };
①  p[10] = 10;     //此句越界
    p[14] = 14;     //此句越界
    p[15] = 15;     //此句越界
②  p[10] = (int)'\xfd'; //此句越界
    cout << "addr:" << hex << (void *)(p) << endl;
    for (int i = -4; i < 16; i++) //注意，只有0-9是合理范围，其余都是越界读
        cout << hex << (void *)(p + i) << ":" << int(p[i]) << endl;

    return 0;
}
```

数组为int a[10]形式

在理解P.1/P.2的情况下，自行构造相似的程序，来观察数组越界后的内存表现，并验证与动态申请是否相似

要求：

1、数组用 char a[10]; 形式

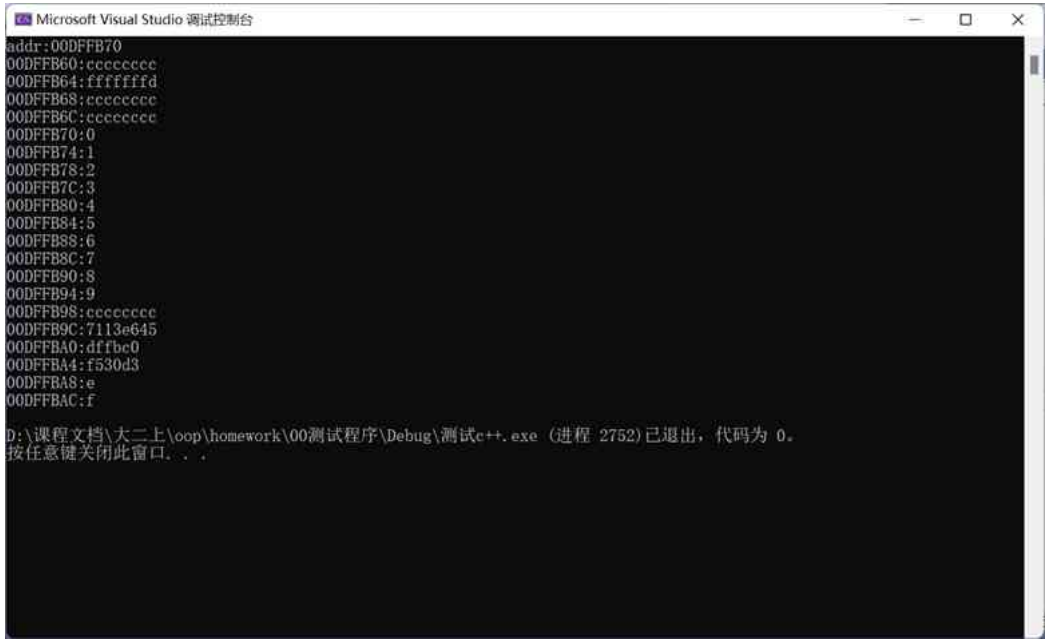
2、数组用 int a[10]; 形式

3、测试程序在下面五种环境下运行

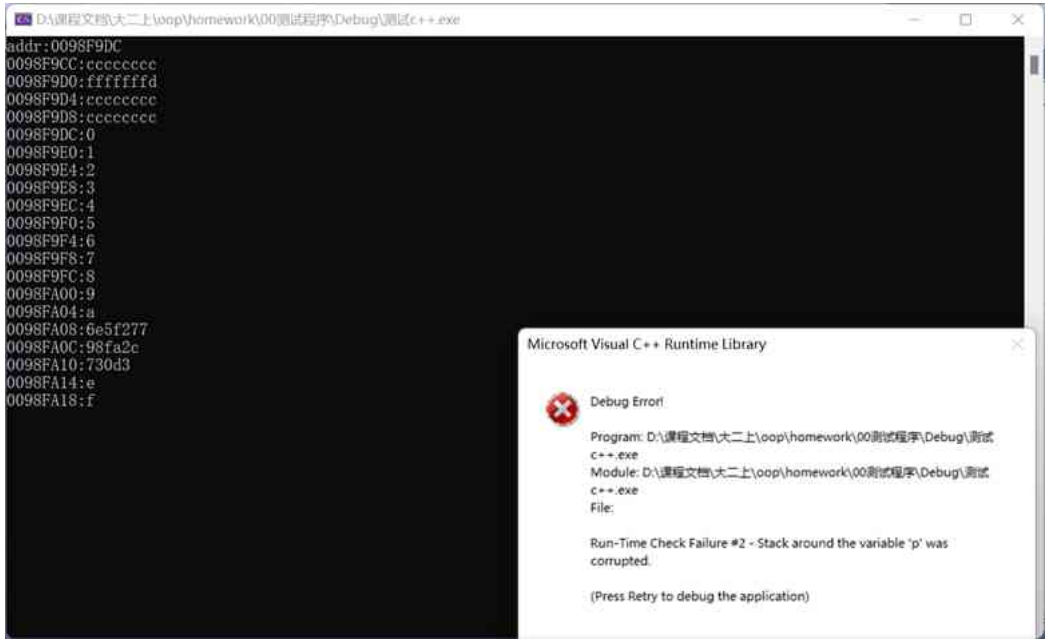
- VS2022 x86/Debug
- VS2022 x86Release
- Dev 32bit-Debug
- Dev 32bit-Release
- Linux

4、每种讨论的结果可截图+文字说明，如果几种环境的结果一致，用一个环境的截图+文字说明即可(可加页)

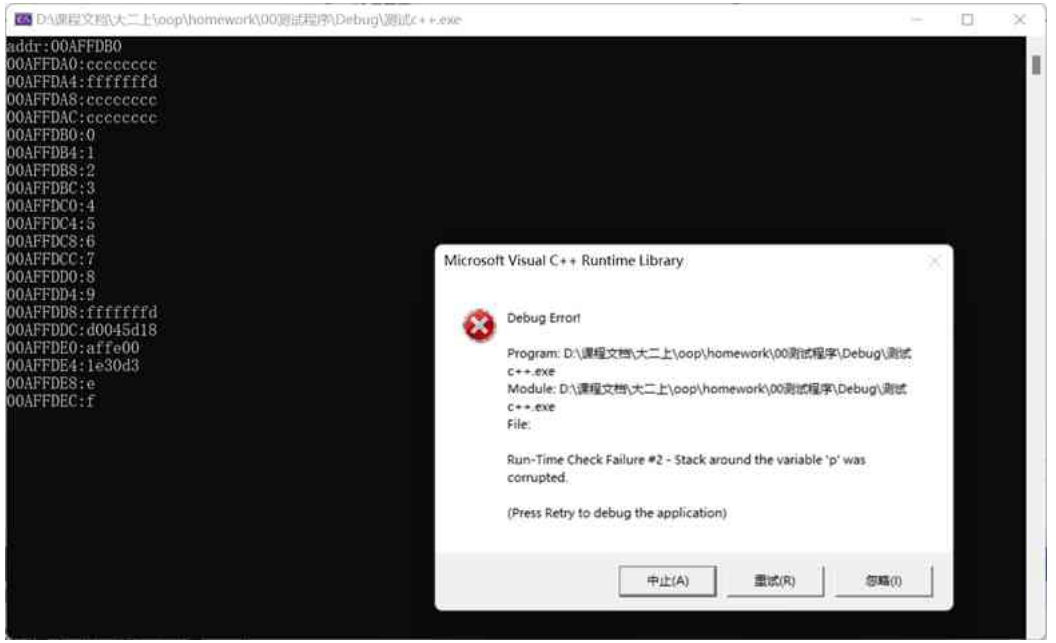
1、①②全部注释



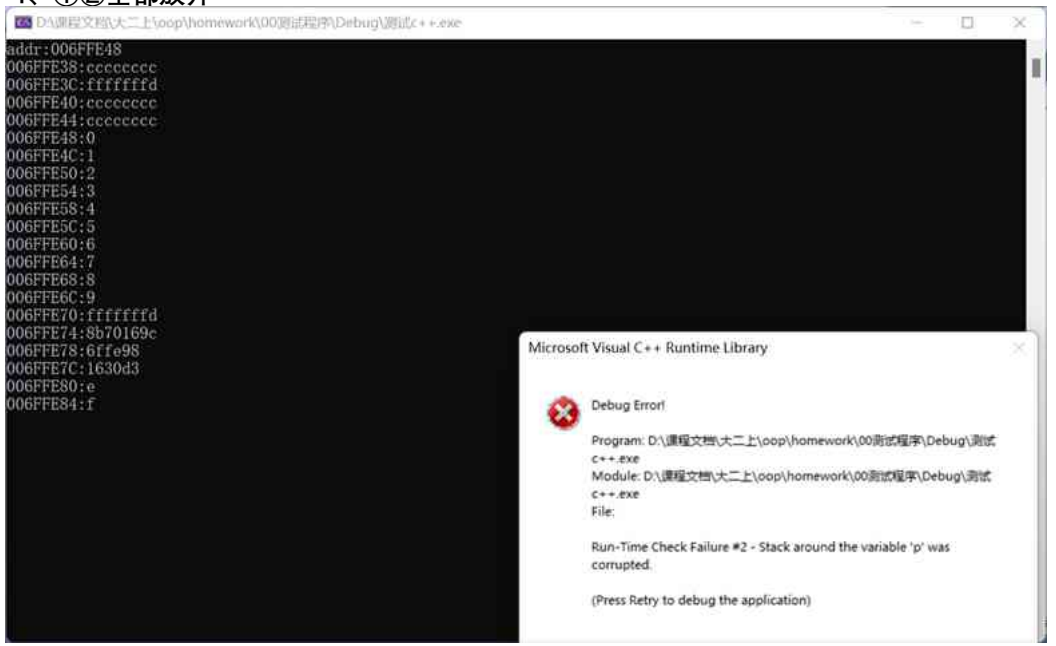

2、①放开，②注释




3、 ②放开， ①注释

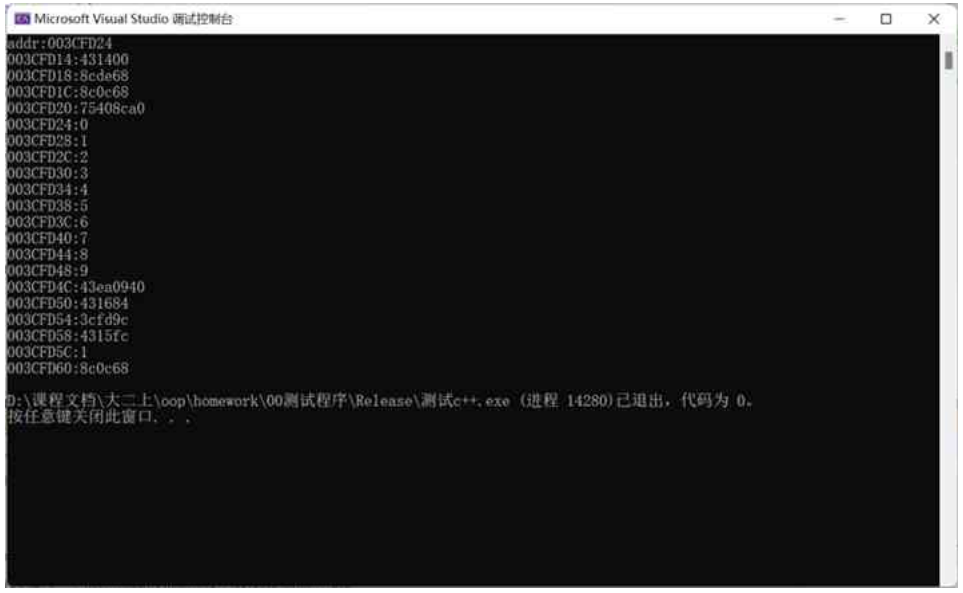



4、①②全部放开

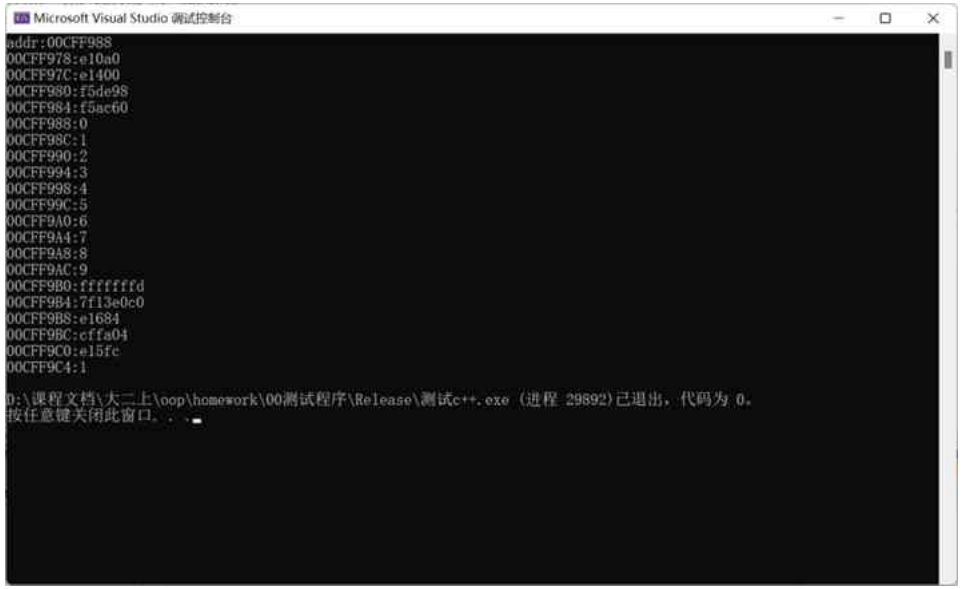



VS2022 x86/Debug

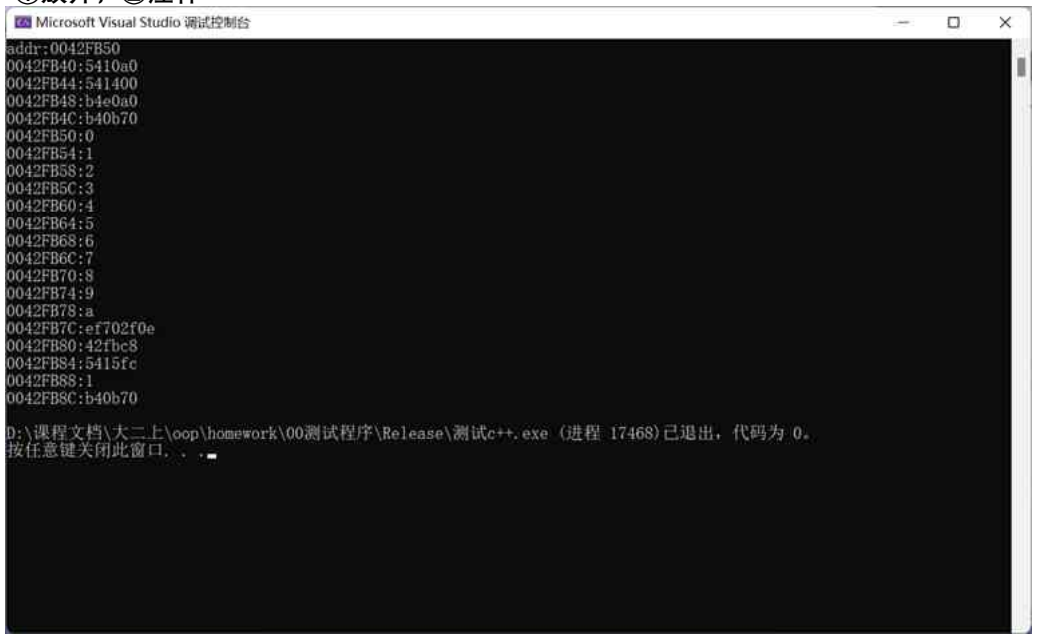
1、①②全部注释



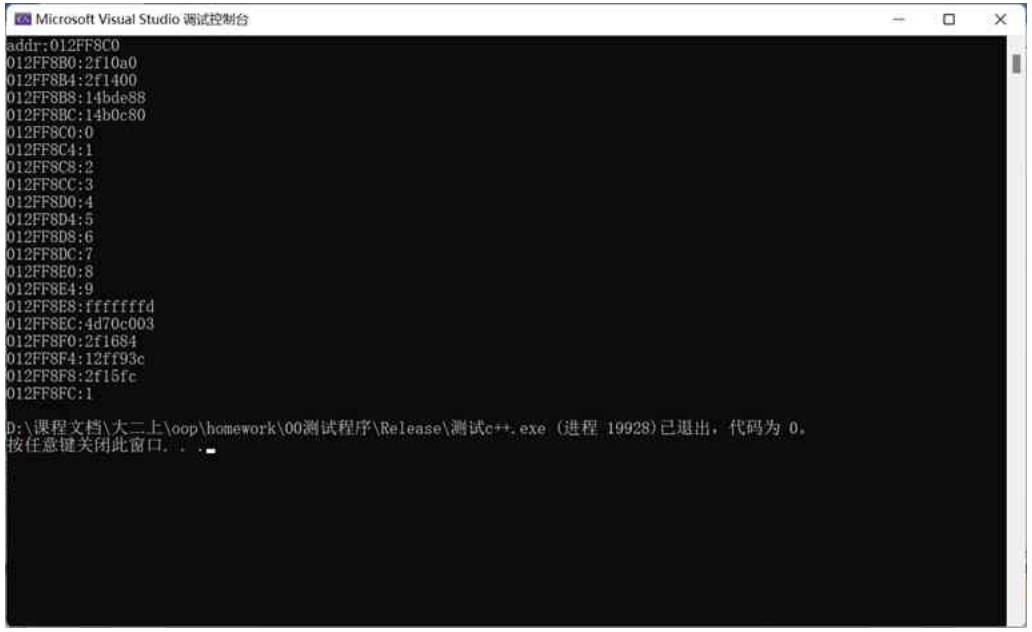

3、 ②放开， ①注释




2、①放开，②注释

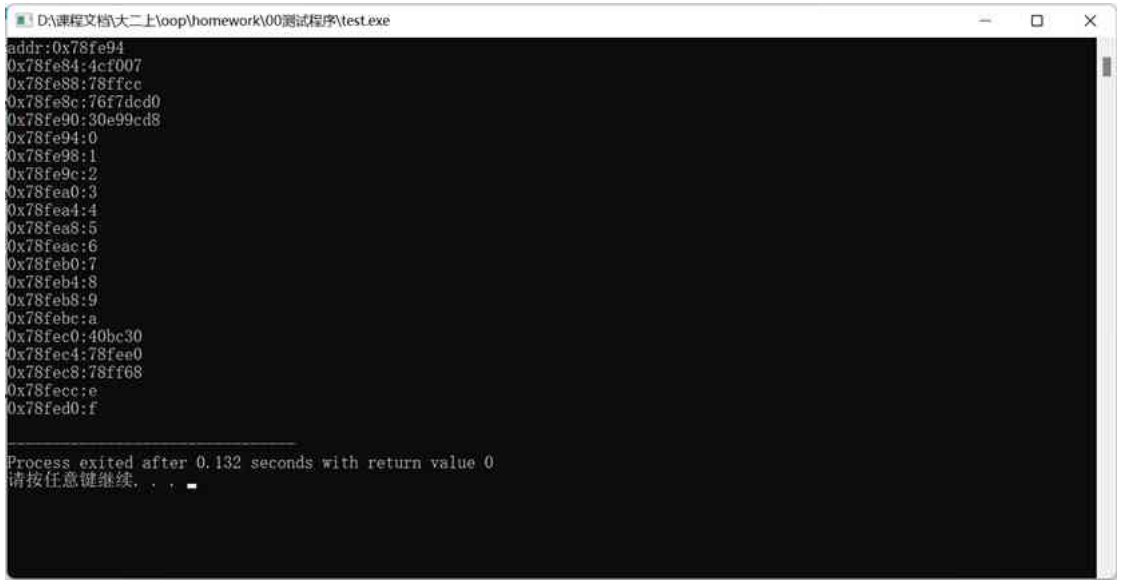



4、①②全部放开

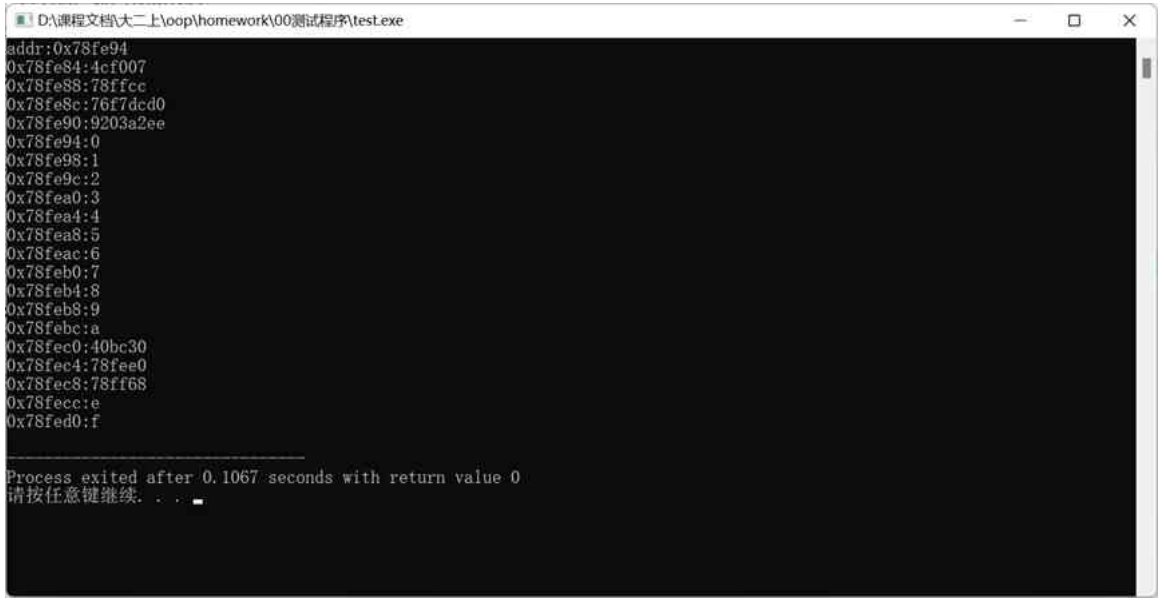



VS2022 x86/Release

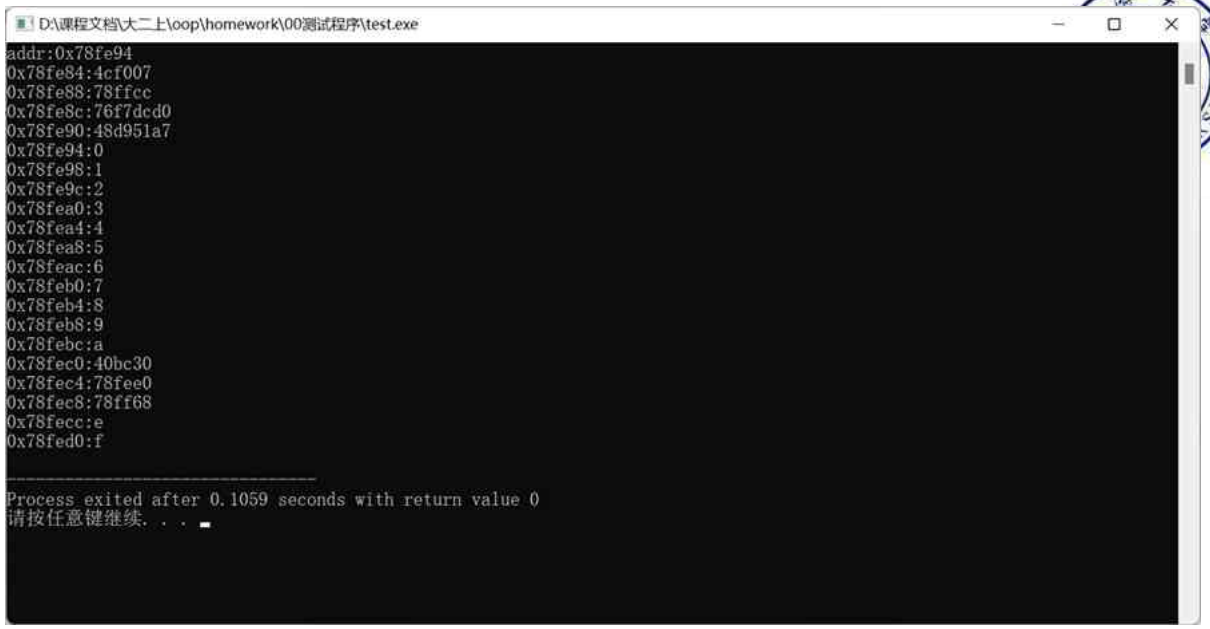
1、①②全部注释



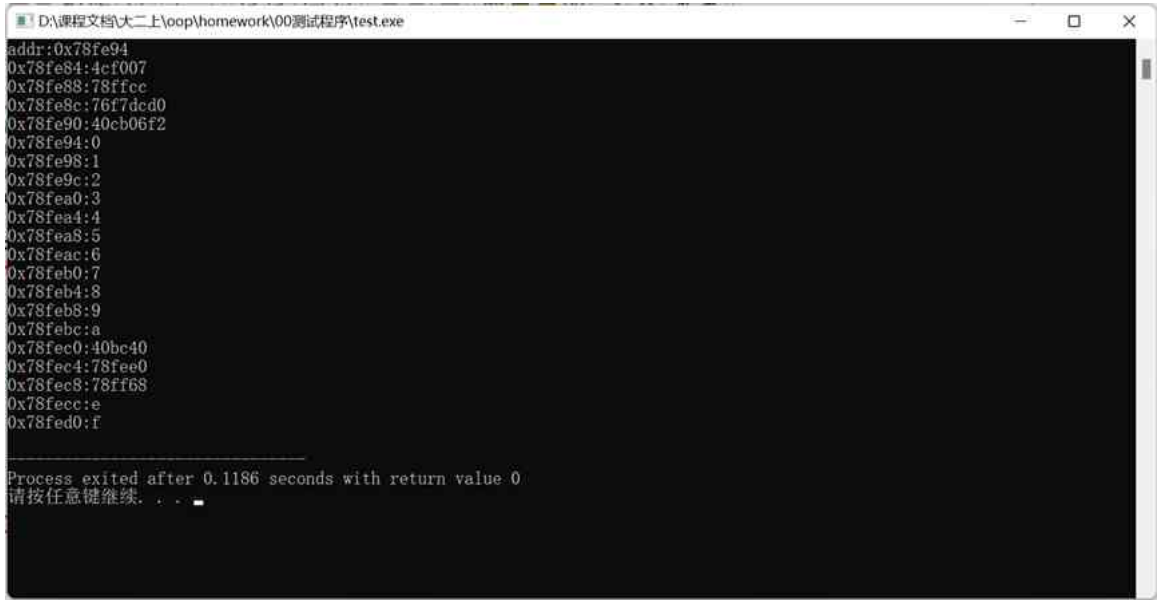

2、①放开，②注释




3、 ②放开， ①注释

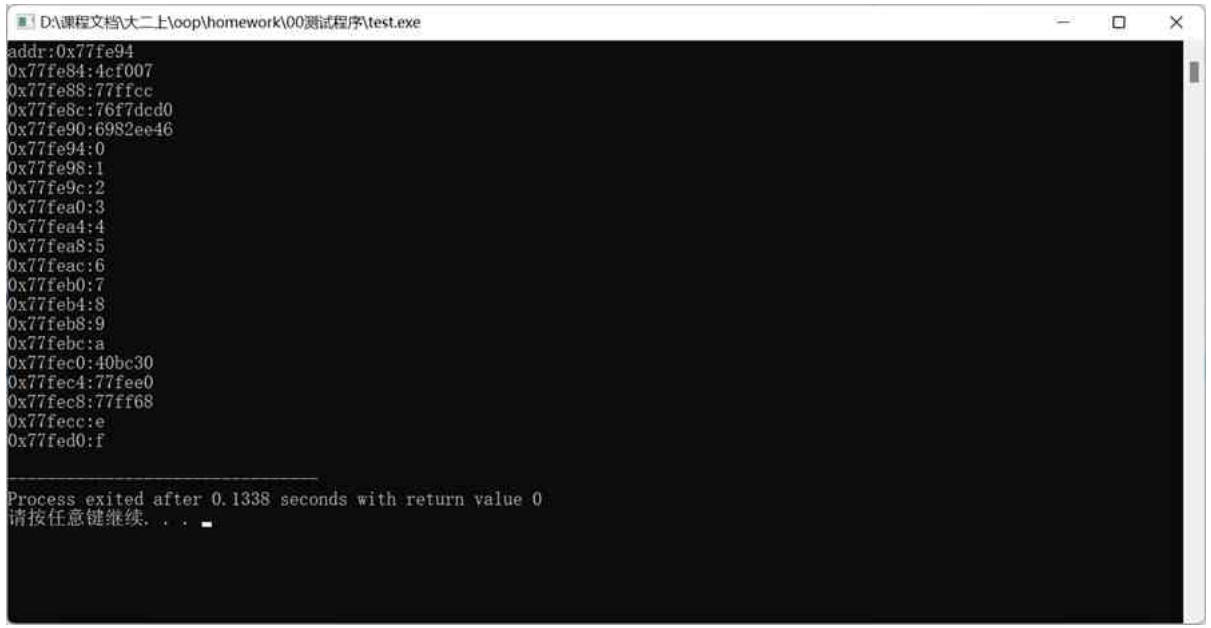



4、①②全部放开

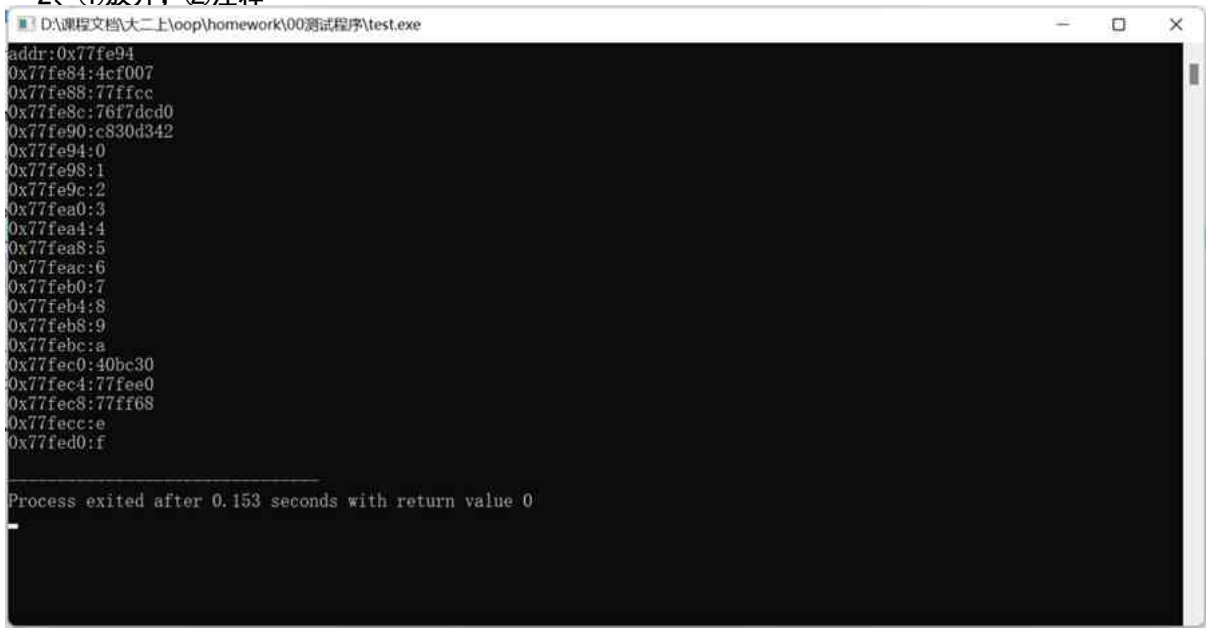



Dev 32bit-Debug

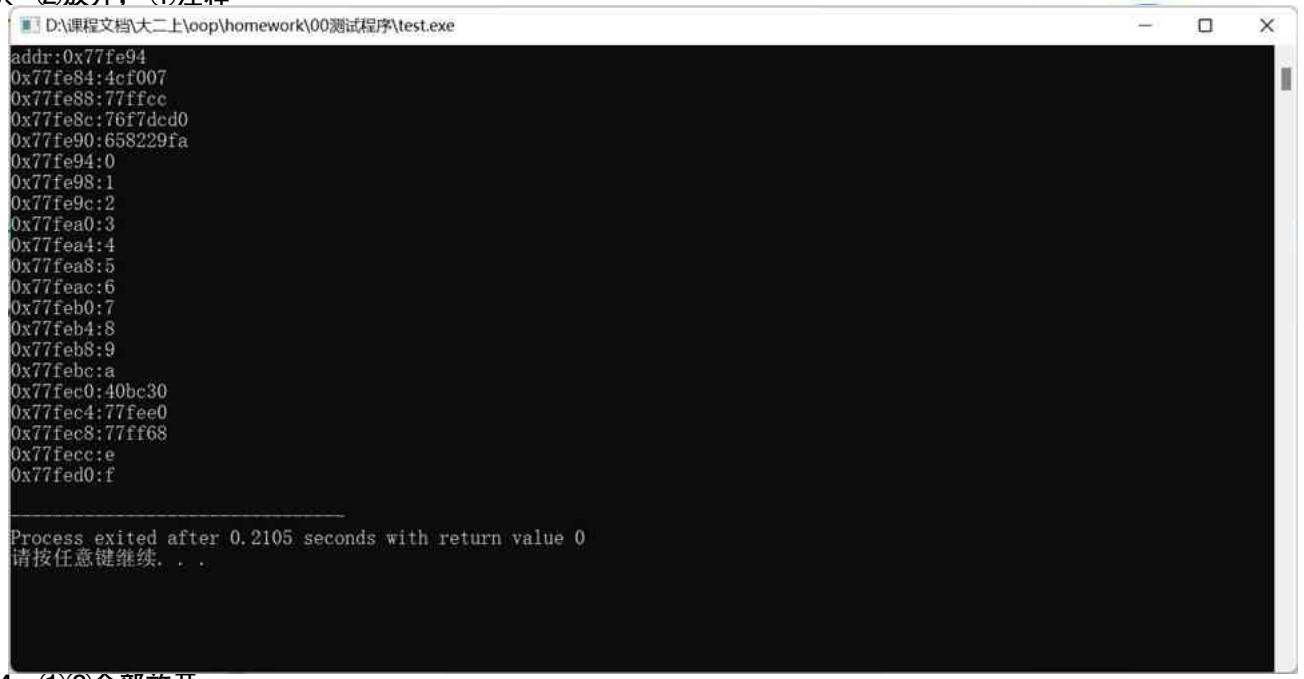
1、①②全部注释



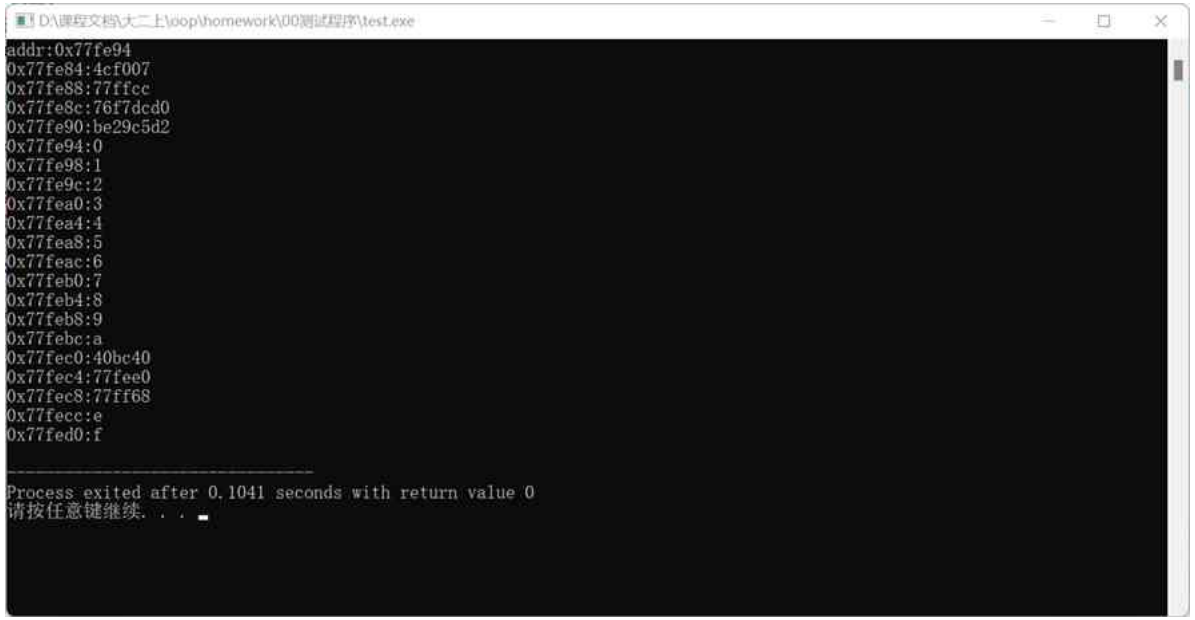

2、①放开，②注释




3、②放开，①注释




4、①②全部放开




Dev 32bit-Release

1、①②全部注释

```
[u2152041@oop ~]$ ./test
addr:0x7ffcfa9c9500
0x7ffcfa9c94f0:fa9c9668
0x7ffcfa9c94f4:7ffc
0x7ffcfa9c94f8:401284
0x7ffcfa9c94fc:0
0x7ffcfa9c9500:0
0x7ffcfa9c9504:1
0x7ffcfa9c9508:2
0x7ffcfa9c950c:3
0x7ffcfa9c9510:4
0x7ffcfa9c9514:5
0x7ffcfa9c9518:6
0x7ffcfa9c951c:7
0x7ffcfa9c9520:8
0x7ffcfa9c9524:9
0x7ffcfa9c9528:0
0x7ffcfa9c952c:b
0x7ffcfa9c9530:1
0x7ffcfa9c9534:0
0x7ffcfa9c9538:e
0x7ffcfa9c953c:f
段错误（核心已转储）
```

2、①放开，②注释

```
[u2152041@oop ~]$ ./test
addr:0x7ffc67d75a90
0x7ffc67d75a80:67d75bf8
0x7ffc67d75a84:7ffc
0x7ffc67d75a88:40128b
0x7ffc67d75a8c:0
0x7ffc67d75a90:0
0x7ffc67d75a94:1
0x7ffc67d75a98:2
0x7ffc67d75a9c:3
0x7ffc67d75aa0:4
0x7ffc67d75aa4:5
0x7ffc67d75aa8:6
0x7ffc67d75aac:7
0x7ffc67d75ab0:8
0x7ffc67d75ab4:9
0x7ffc67d75ab8:a
0x7ffc67d75abc:b
0x7ffc67d75ac0:1
0x7ffc67d75ac4:0
0x7ffc67d75ac8:e
0x7ffc67d75acc:f
段错误（核心已转储）
```

3、 ② 放开， ①注释

```
[u2152041@oop ~]$ ./test
addr:0x7ffd4fb1ee80
0x7ffd4fb1ee70:4fb1efe8
0x7ffd4fb1ee74:7ffd
0x7ffd4fb1ee78:40128b
0x7ffd4fb1ee7c:0
0x7ffd4fb1ee80:0
0x7ffd4fb1ee84:1
0x7ffd4fb1ee88:2
0x7ffd4fb1ee8c:3
0x7ffd4fb1ee90:4
0x7ffd4fb1ee94:5
0x7ffd4fb1ee98:6
0x7ffd4fb1ee9c:7
0x7ffd4fb1eea0:8
0x7ffd4fb1eea4:9
0x7ffd4fb1eea8:ffffffd
0x7ffd4fb1eeac:b
0x7ffd4fb1eeb0:1
0x7ffd4fb1eeb4:0
0x7ffd4fb1eeb8:e
0x7ffd4fb1eebc:f
段错误（核心已转储）
```

4、①②全部放开

```
[u2152041@oop ~]$ ./test
addr:0x7ffd32b3a690
0x7ffd32b3a680:32b3a7f8
0x7ffd32b3a684:7ffd
0x7ffd32b3a688:401292
0x7ffd32b3a68c:0
0x7ffd32b3a690:0
0x7ffd32b3a694:1
0x7ffd32b3a698:2
0x7ffd32b3a69c:3
0x7ffd32b3a6a0:4
0x7ffd32b3a6a4:5
0x7ffd32b3a6a8:6
0x7ffd32b3a6ac:7
0x7ffd32b3a6b0:8
0x7ffd32b3a6b4:9
0x7ffd32b3a6b8:ffffffd
0x7ffd32b3a6bc:b
0x7ffd32b3a6c0:1
0x7ffd32b3a6c4:0
0x7ffd32b3a6c8:e
0x7ffd32b3a6cc:f
段错误（核心已转储）
```

Linux

# §.关于动态内存申请后越界访问的深度讨论

★ 如何判断普通数组的越界访问（C方式，注意源程序后缀为.c）

```c
/* 2152041 王浩 计科 */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char p[10];

    strcpy(p, "123456789");
     // p[10] = 'a';    //此句越界
    p[14] = 'A';    //此句越界
    p[15] = 'B';    //此句越界
//    p[10] = '\xfd'; //此句越界
    printf("addr:%p\n", p);
    for (int i = -4; i < 16; i++) //注意，只有0-9是合理范围，其余都是越界读
        printf("%p:%02x\n", (p + i), p[i]);

    return 0;
}
```

数组为char a[10]形式

在理解P.1/P.2的情况下，自行构造相似的程序，来观察数组越界后的内存表现，并验证与动态申请是否相似

要求：

1、数组用 char a[10]; 形式

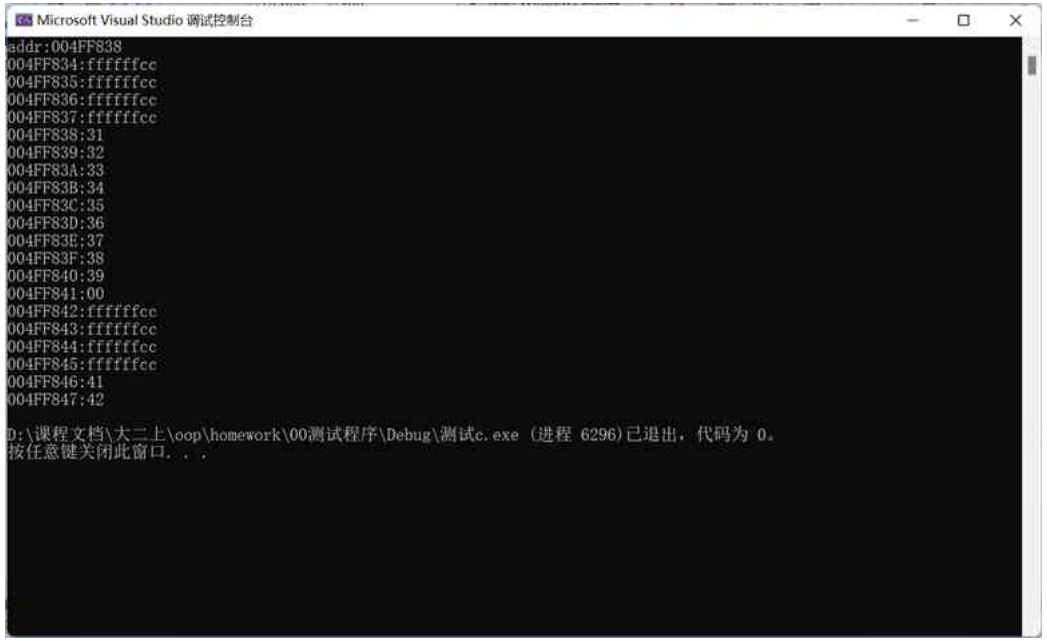
2、数组用 int a[10]; 形式

3、测试程序在下面五种环境下运行

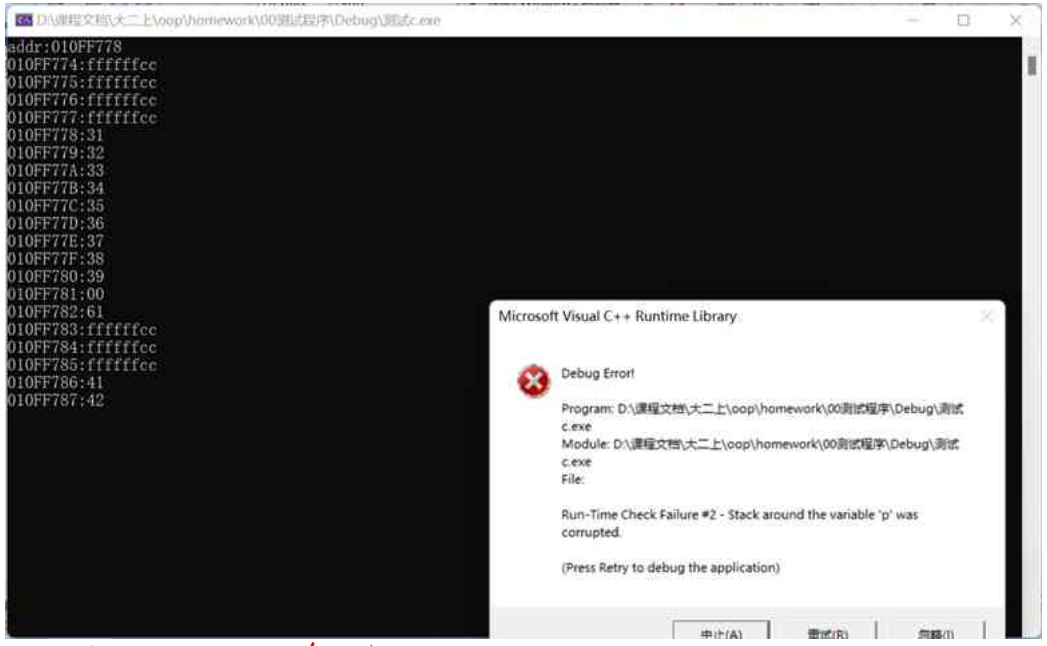
- VS2022 x86/Debug
- VS2022 x86Release
- Dev 32bit-Debug
- Dev 32bit-Release
- Linux

4、每种讨论的结果可截图+文字说明，如果几种环境的结果一致，用一个环境的截图+文字说明即可(可加页)

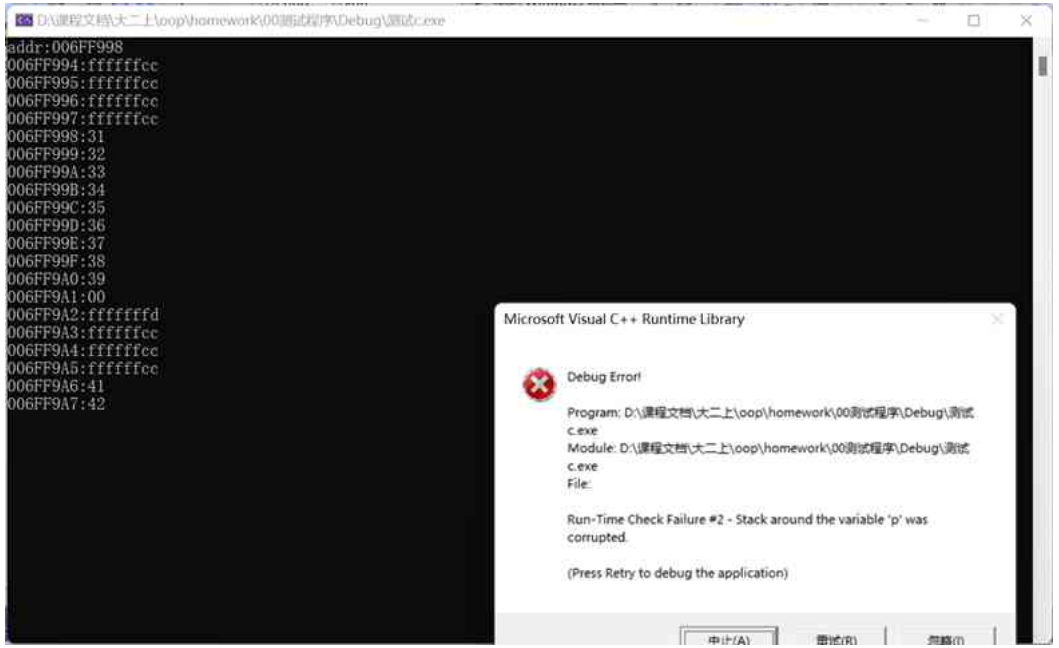
1、①②全部注释



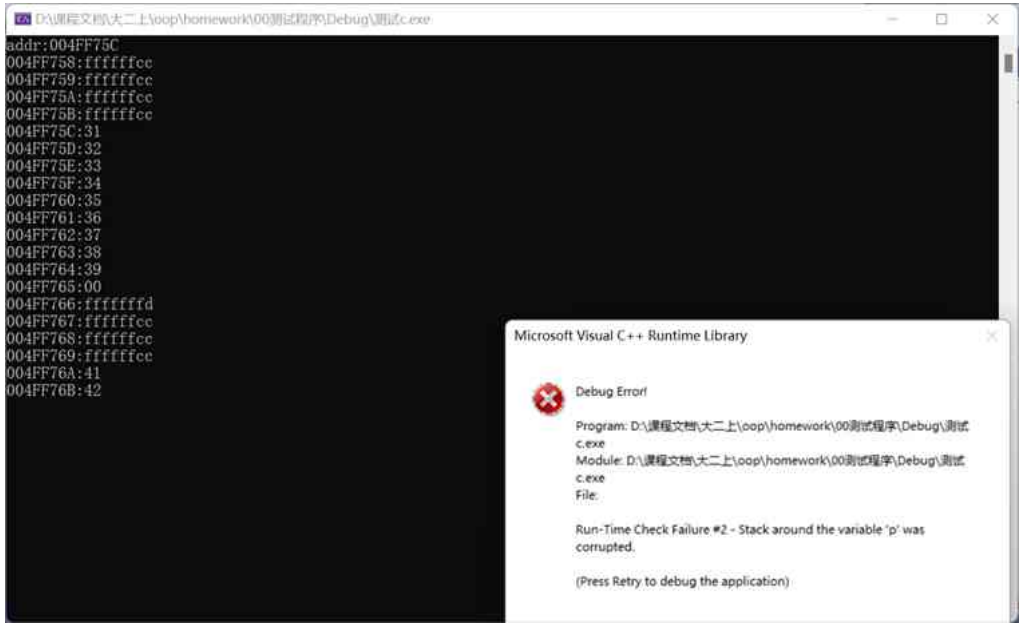

2、①放开，②注释




3、 ②放开， ①注释

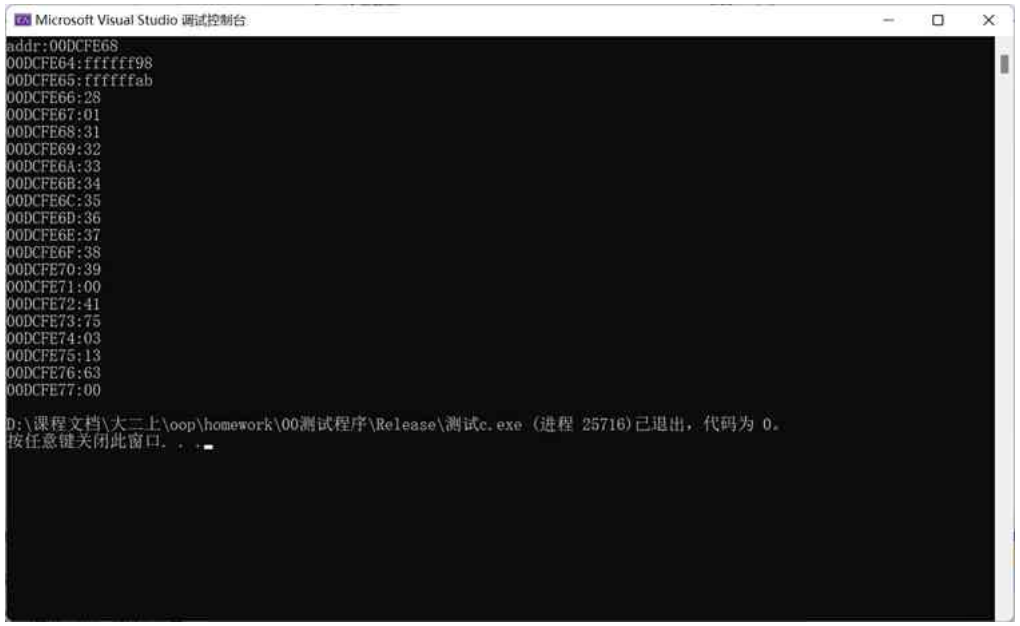



4、①②全部放开

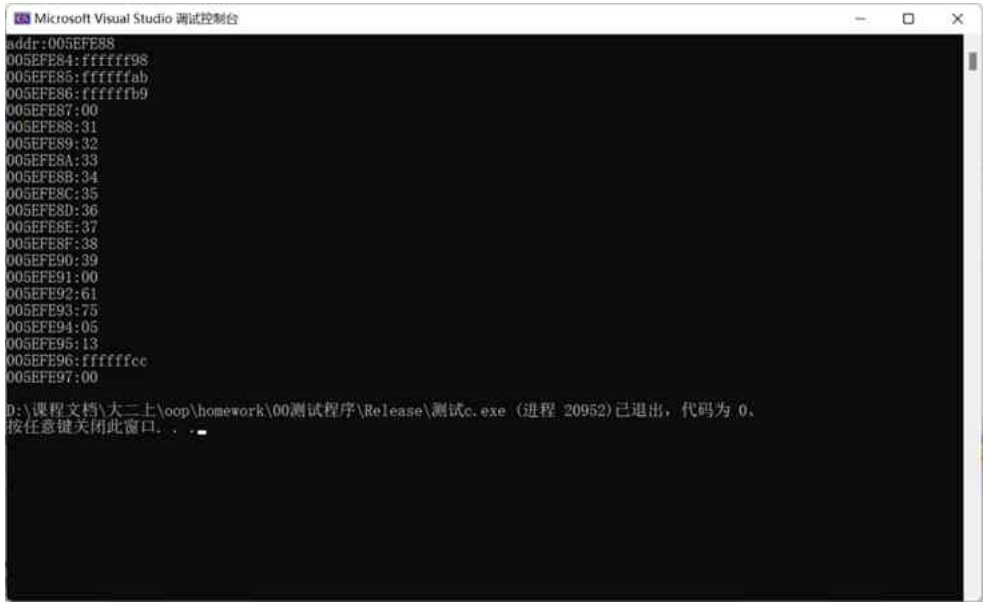



VS2022 x86/Debug

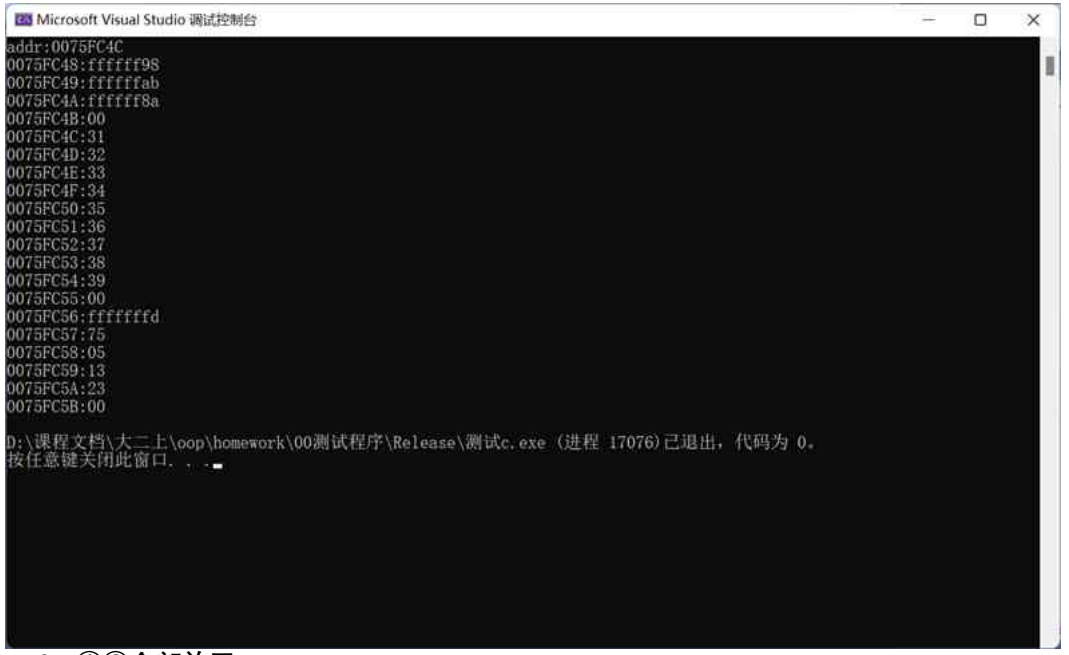
1、①②全部注释



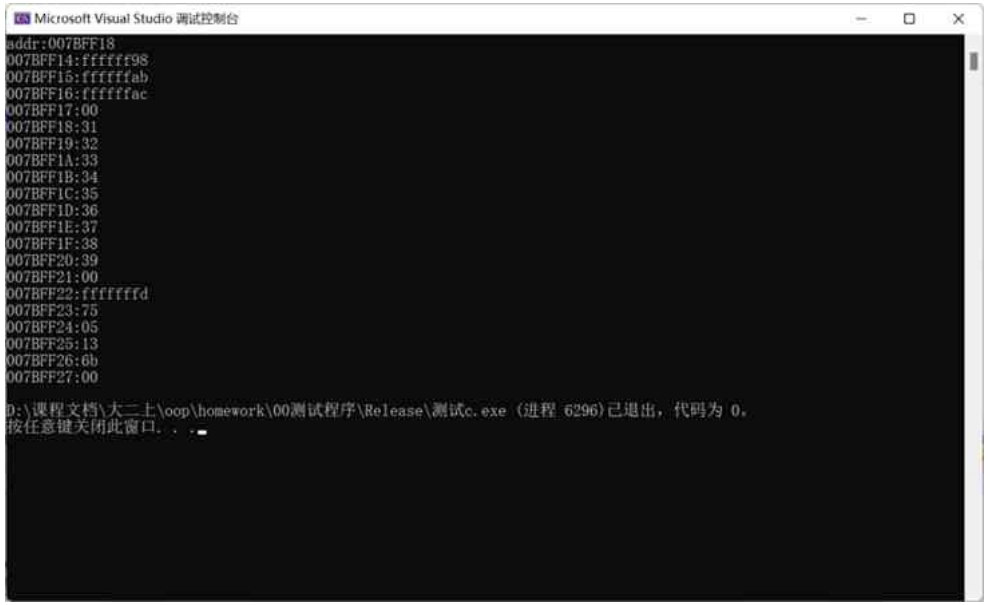

2、①放开，②注释




3、 ②放开， ①注释




4、①②全部放开




VS2022 x86/Release　　Dev 32bit-Debug　　Dev 32bit-Release

1、①②全部注释

```
addr:0x7fffd5f57c42
0x7fffd5f57c3e:00
0x7fffd5f57c3f:00
0x7fffd5f57c40:0d
0x7fffd5f57c41:00
0x7fffd5f57c42:31
0x7fffd5f57c43:32
0x7fffd5f57c44:33
0x7fffd5f57c45:34
0x7fffd5f57c46:35
0x7fffd5f57c47:36
0x7fffd5f57c48:37
0x7fffd5f57c49:38
0x7fffd5f57c4a:39
0x7fffd5f57c4b:00
0x7fffd5f57c4c:0a
0x7fffd5f57c4d:00
0x7fffd5f57c4e:00
0x7fffd5f57c4f:00
0x7fffd5f57c50:41
0x7fffd5f57c51:42
```

2、①放开，②注释

```
addr:0x7ffc85d8a0e2
0x7ffc85d8a0de:00
0x7ffc85d8a0df:00
0x7ffc85d8a0e0:0d
0x7ffc85d8a0e1:00
0x7ffc85d8a0e2:31
0x7ffc85d8a0e3:32
0x7ffc85d8a0e4:33
0x7ffc85d8a0e5:34
0x7ffc85d8a0e6:35
0x7ffc85d8a0e7:36
0x7ffc85d8a0e8:37
0x7ffc85d8a0e9:38
0x7ffc85d8a0ea:39
0x7ffc85d8a0eb:00
0x7ffc85d8a0ec:0a
0x7ffc85d8a0ed:00
0x7ffc85d8a0ee:00
0x7ffc85d8a0ef:00
0x7ffc85d8a0f0:41
0x7ffc85d8a0f1:42
```

3、 ② 放开， ①注释

```
addr:0x7ffffc049662
0x7ffffc04965e:00
0x7ffffc04965f:00
0x7ffffc049660:0d
0x7ffffc049661:00
0x7ffffc049662:31
0x7ffffc049663:32
0x7ffffc049664:33
0x7ffffc049665:34
0x7ffffc049666:35
0x7ffffc049667:36
0x7ffffc049668:37
0x7ffffc049669:38
0x7ffffc04966a:39
0x7ffffc04966b:00
0x7ffffc04966c:0a
0x7ffffc04966d:00
0x7ffffc04966e:00
0x7ffffc04966f:00
0x7ffffc049670:41
0x7ffffc049671:42
```

4、①②全部放开

```
addr:0x7ffd7c77ad22
0x7ffd7c77ad1e:00
0x7ffd7c77ad1f:00
0x7ffd7c77ad20:0d
0x7ffd7c77ad21:00
0x7ffd7c77ad22:31
0x7ffd7c77ad23:32
0x7ffd7c77ad24:33
0x7ffd7c77ad25:34
0x7ffd7c77ad26:35
0x7ffd7c77ad27:36
0x7ffd7c77ad28:37
0x7ffd7c77ad29:38
0x7ffd7c77ad2a:39
0x7ffd7c77ad2b:00
0x7ffd7c77ad2c:0a
0x7ffd7c77ad2d:00
0x7ffd7c77ad2e:00
0x7ffd7c77ad2f:00
0x7ffd7c77ad30:41
0x7ffd7c77ad31:42
```

# §.关于动态内存申请后越界访问的深度讨论

★ 如何判断普通数组的越界访问（C方式，注意源程序后缀为.c）

```c
/* 2152041 王浩 计科 */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    char p[10];

    strcpy(p, "123456789");
     // p[10] = 'a';    //此句越界
    p[14] = 'A';    //此句越界
    p[15] = 'B';    //此句越界
//    p[10] = '\xfd'; //此句越界
    printf("addr:%p\n", p);
    for (int i = -4; i < 16; i++) //注意，只有0-9是合理范围，其余都是越界读
        printf("%p:%02x\n", (p + i), p[i]);

    return 0;
}
```

数组为int a[10]形式

在理解P.1/P.2的情况下，自行构造相似的程序，来观察数组越界后的内存表现，并验证与动态申请是否相似

要求：

1、数组用 char a[10]; 形式

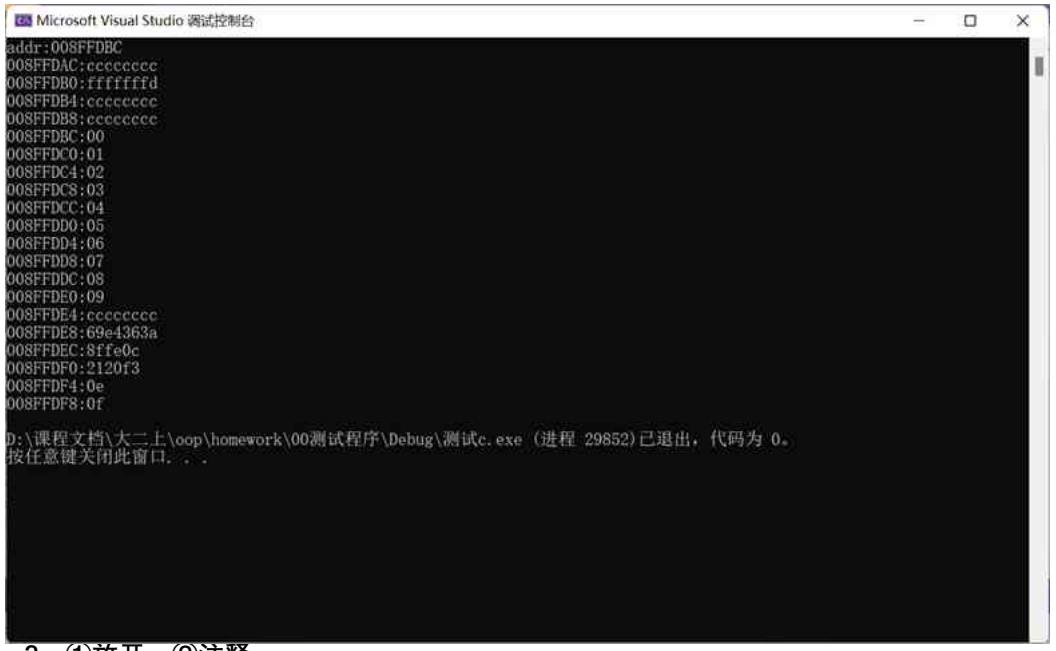
2、数组用 int a[10]; 形式

3、测试程序在下面五种环境下运行

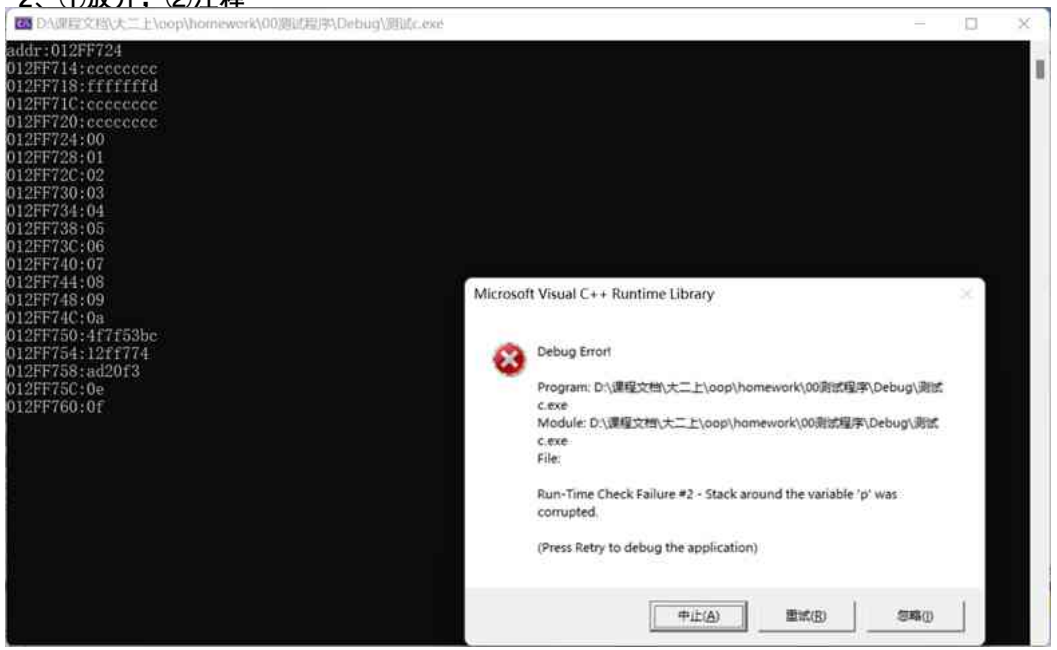
- VS2022 x86/Debug
- VS2022 x86Release
- Dev 32bit-Debug
- Dev 32bit-Release
- Linux

4、每种讨论的结果可截图+文字说明，如果几种环境的结果一致，用一个环境的截图+文字说明即可(可加页)

1、①②全部注释

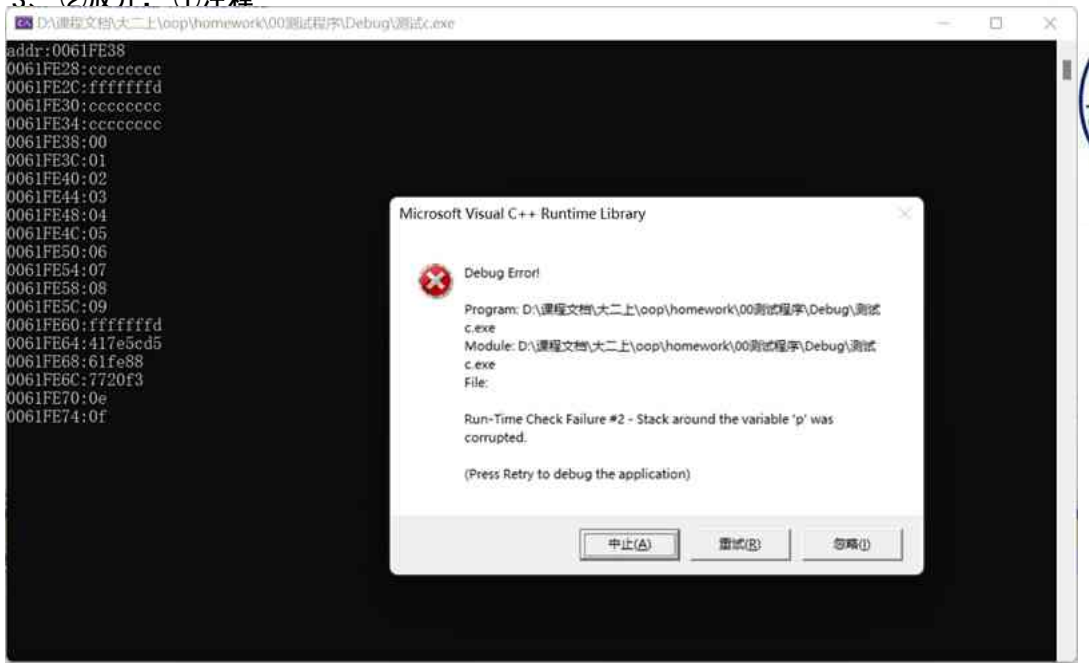



2、①放开，②注释

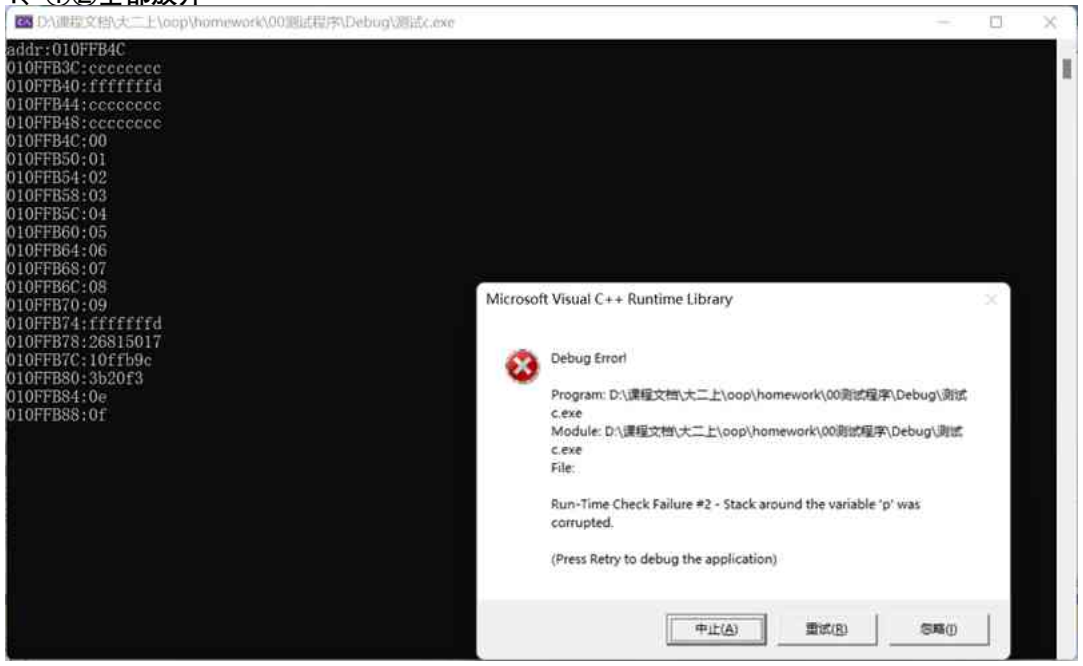



VS2022 x86/Debug

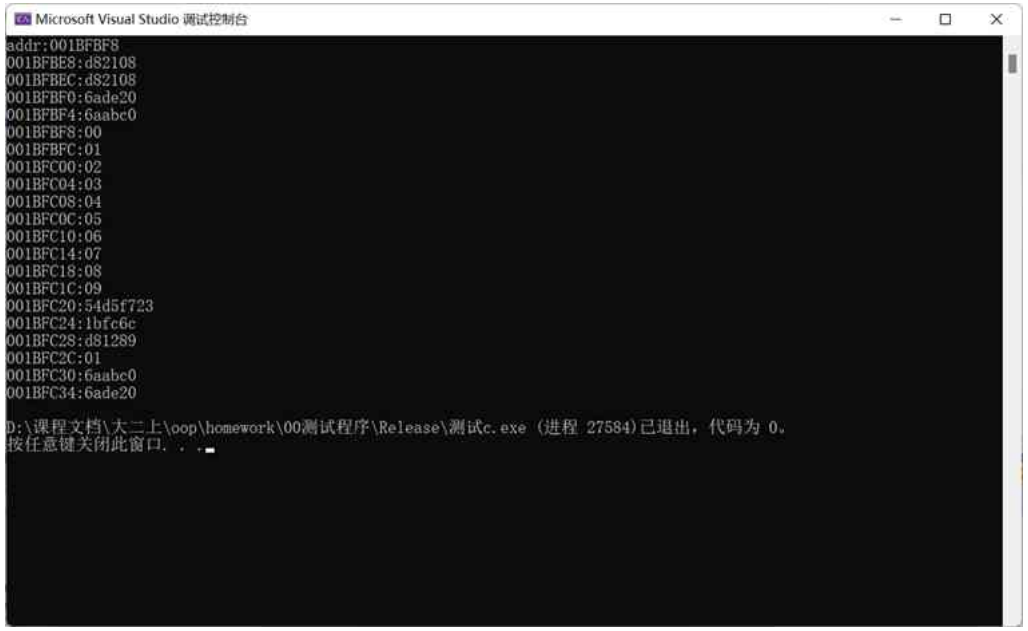
3、②放开，①注释



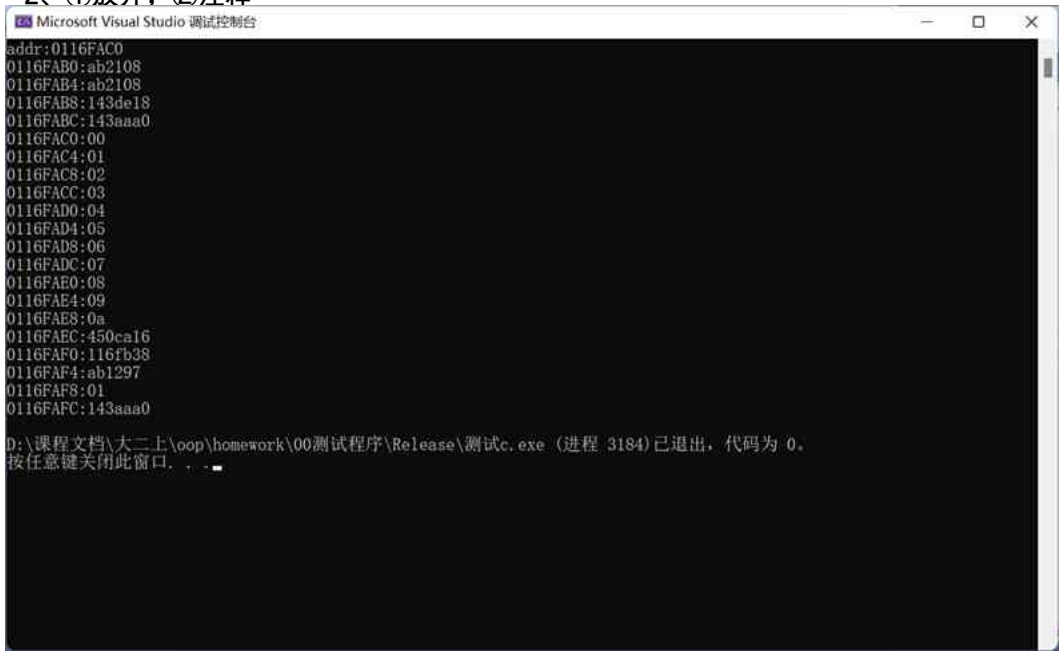

4、①②全部放开



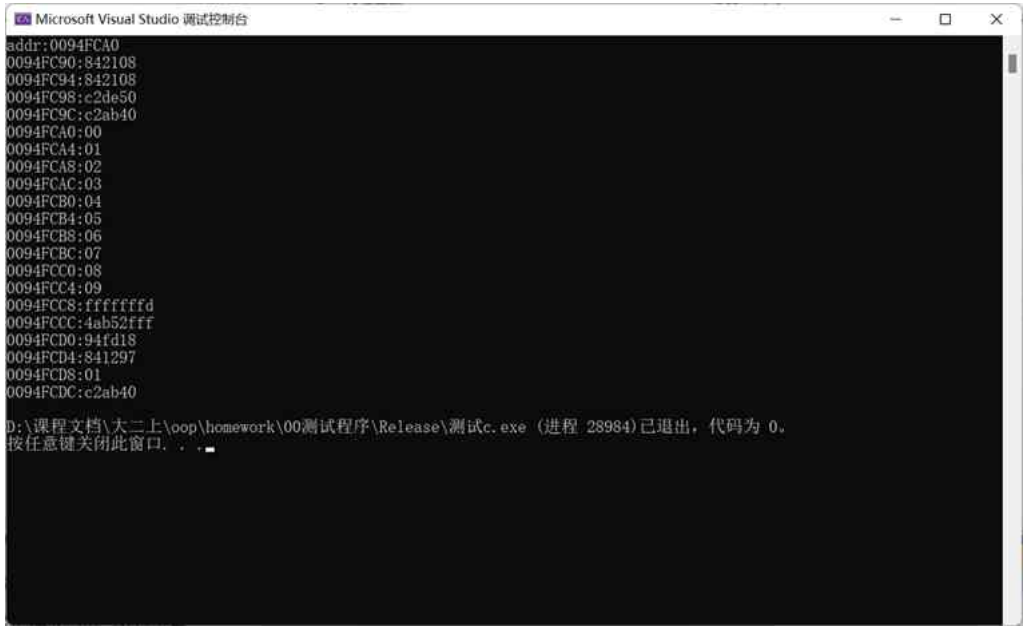
1、①②全部注释



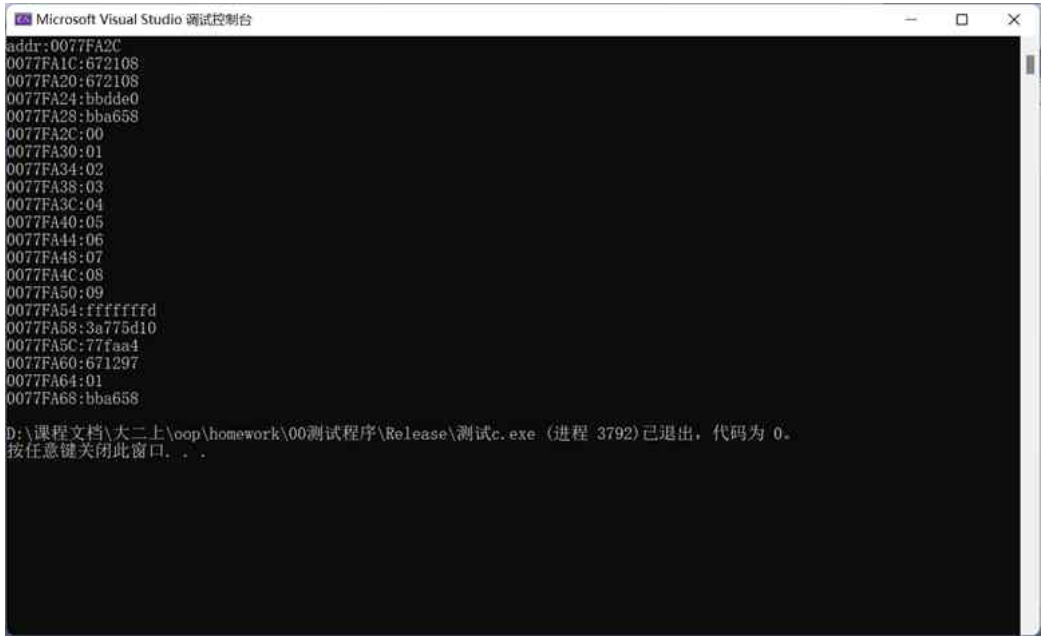

2、①放开，②注释




3、 ②放开， ①注释




4、①②全部放开




VS2022 x86/Release　　Dev 32bit-Debug　　Dev 32bit-Release

1、①②全部注释

```
addr:0x7ffc31bb22f0
0x7ffc31bb22e0:00
0x7ffc31bb22e4:00
0x7ffc31bb22e8:4011d3
0x7ffc31bb22ec:00
0x7ffc31bb22f0:00
0x7ffc31bb22f4:01
0x7ffc31bb22f8:02
0x7ffc31bb22fc:03
0x7ffc31bb2300:04
0x7ffc31bb2304:05
0x7ffc31bb2308:06
0x7ffc31bb230c:07
0x7ffc31bb2310:08
0x7ffc31bb2314:09
0x7ffc31bb2318:01
0x7ffc31bb231c:0b
0x7ffc31bb2320:01
0x7ffc31bb2324:00
0x7ffc31bb2328:0e
0x7ffc31bb232c:0f
段错误（核心已转储）
```

2、①放开，②注释

```
addr:0x7ffcff515c00
0x7ffcff515bf0:00
0x7ffcff515bf4:00
0x7ffcff515bf8:4011da
0x7ffcff515bfc:00
0x7ffcff515c00:00
0x7ffcff515c04:01
0x7ffcff515c08:02
0x7ffcff515c0c:03
0x7ffcff515c10:04
0x7ffcff515c14:05
0x7ffcff515c18:06
0x7ffcff515c1c:07
0x7ffcff515c20:08
0x7ffcff515c24:09
0x7ffcff515c28:0a
0x7ffcff515c2c:0b
0x7ffcff515c30:01
0x7ffcff515c34:00
0x7ffcff515c38:0e
0x7ffcff515c3c:0f
段错误（核心已转储）
```

Linux

3、 ② 放开， ①注释

```
addr:0x7ffd2a7652d0
0x7ffd2a7652c0:00
0x7ffd2a7652c4:00
0x7ffd2a7652c8:4011da
0x7ffd2a7652cc:00
0x7ffd2a7652d0:00
0x7ffd2a7652d4:01
0x7ffd2a7652d8:02
0x7ffd2a7652dc:03
0x7ffd2a7652e0:04
0x7ffd2a7652e4:05
0x7ffd2a7652e8:06
0x7ffd2a7652ec:07
0x7ffd2a7652f0:08
0x7ffd2a7652f4:09
0x7ffd2a7652f8:fffffffd
0x7ffd2a7652fc:0b
0x7ffd2a765300:01
0x7ffd2a765304:00
0x7ffd2a765308:0e
0x7ffd2a76530c:0f
段错误（核心已转储）
```

4、①②全部放开

```
addr:0x7fffa620d9f0
0x7fffa620d9e0:00
0x7fffa620d9e4:00
0x7fffa620d9e8:4011e1
0x7fffa620d9ec:00
0x7fffa620d9f0:00
0x7fffa620d9f4:01
0x7fffa620d9f8:02
0x7fffa620d9fc:03
0x7fffa620da00:04
0x7fffa620da04:05
0x7fffa620da08:06
0x7fffa620da0c:07
0x7fffa620da10:08
0x7fffa620da14:09
0x7fffa620da18:fffffffd
0x7fffa620da1c:0b
0x7fffa620da20:01
0x7fffa620da24:00
0x7fffa620da28:0e
0x7fffa620da2c:0f
段错误（核心已转储）
```

# §.关于动态内存申请后越界访问的深度讨论

★ 最后一页：仔细总结本作业（多种形式的测试程序/多个编译器环境/不同结论），谈谈你对内存越界访问的整体理解
包括但不限于操作系统/编译器如何防范越界、你应该养成怎样的使用习惯来尽量防范越界

内存访问时，操作系统和编译器不可能时刻监控着每块内存的变化，所以会通过相邻内存是否改变来进行简化编译过程。操作系统会监控数组和相应的动态申请的内存变化，而编译器则会监控数组的索引是否越界来判断是否越界访问。
为了防范越界，首先应该对申请的内存大小有着良好的认知，知道边界的位置，其次要在对相应的内存进行改变时进行测试，防止未知情况下的越界访问。