



§. 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)



§. 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)

```
addr:009BAD20
009BAD1C: ffffffff
009BAD1D: ffffffff
009BAD1E: ffffffff
009BAD1F: ffffffff
009BAD20: 31
009BAD21: 32
009BAD22: 33
009BAD23: 34
009BAD24: 35
009BAD25: 36
009BAD26: 37
009BAD27: 38
009BAD28: 39
009BAD29: 00
009BAD2A: ffffffff
009BAD2B: ffffffff
009BAD2C: ffffffff
009BAD2D: ffffffff
009BAD2E: 41
009BAD2F: 42
```

VS
x86/debug

```
addr:014726D0
014726CC: 0
014726CD: 1
014726CE: 0
014726CF: ffffffff8e
014726D0: 31
014726D1: 32
014726D2: 33
014726D3: 34
014726D4: 35
014726D5: 36
014726D6: 37
014726D7: 38
014726D8: 39
014726D9: 0
014726DA: ffffffff
014726DB: 0
014726DC: 0
014726DD: 0
014726DE: 41
014726DF: 42
D:\test\Release\demo.exe
按任意键关闭此窗口...
```

x86/release

```
C:\Users\asus\Desktop\test2.exe
addr:00D31568
00D31564: ffffffffa2
00D31565: 65
00D31566: 00
00D31567: 0e
00D31568: 31
00D31569: 32
00D3156A: 33
00D3156B: 34
00D3156C: 35
00D3156D: 36
00D3156E: 37
00D3156F: 38
00D31570: 39
00D31571: 00
00D31572: 73
00D31573: 41
00D31574: 70
00D31575: 70
00D31576: 41
00D31577: 42
```

dev
32bit-debug

```
C:\Users\asus\Desktop\test2.exe
addr:00D01568
00D01564: 47
00D01565: 30
00D01566: 00
00D01567: 0e
00D01568: 31
00D01569: 32
00D0156A: 33
00D0156B: 34
00D0156C: 35
00D0156D: 36
00D0156E: 37
00D0156F: 38
00D01570: 39
00D01571: 00
00D01572: 73
00D01573: 41
00D01574: 70
00D01575: 70
00D01576: 41
00D01577: 42
```

32bit-release

```
[u2351114@oop ~]$ ./test2
addr:0x2c0002a0
0x2c00029c: 00
0x2c00029d: 00
0x2c00029e: 00
0x2c00029f: 00
0x2c0002a0: 31
0x2c0002a1: 32
0x2c0002a2: 33
0x2c0002a3: 34
0x2c0002a4: 35
0x2c0002a5: 36
0x2c0002a6: 37
0x2c0002a7: 38
0x2c0002a8: 39
0x2c0002a9: 00
0x2c0002aa: 00
0x2c0002ab: 00
0x2c0002ac: 00
0x2c0002ad: 00
0x2c0002ae: 41
0x2c0002af: 42
[u2351114@oop ~]$
```

linux

在VS2022的x86/Debug模式下运行:
1、①②③全部注释, 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?

程序正常运行输出。越界写操作成功写入, 越界读操作正常读出,
未赋值的地址中为随机值(vs)
linux中是0



§. 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)

```
addr:0081AD20
0081AD1C: ffffffff
0081AD1D: ffffffff
0081AD1E: ffffffff
0081AD1F: ffffffff
0081AD20: 31
0081AD21: 32
0081AD22: 33
0081AD23: 34
0081AD24: 35
0081AD25: 36
0081AD26: 37
0081AD27: 38
0081AD28: 39
0081AD29: 00
0081AD2A: 61
0081AD2B: ffffffff
0081AD2C: ffffffff
0081AD2D: ffffffff
0081AD2E: 41
0081AD2F: 42
```

Microsoft Visual Studio 调试控制台

```
addr:00E51F40
00E51F3C: 0
00E51F3D: 11
00E51F3E: 0
00E51F3F: ffffffff8e
00E51F40: 31
00E51F41: 32
00E51F42: 33
00E51F43: 34
00E51F44: 35
00E51F45: 36
00E51F46: 37
00E51F47: 38
00E51F48: 39
00E51F49: 0
00E51F4A: ffffffff
00E51F4B: 0
00E51F4C: 0
00E51F4D: 0
00E51F4E: 41
00E51F4F: 42
```

C:\Users\asus\Desktop\test2.exe

```
addr:00CF1568
00CF1564: 77
00CF1565: 32
00CF1566: 00
00CF1567: 0e
00CF1568: 31
00CF1569: 32
00CF156A: 33
00CF156B: 34
00CF156C: 35
00CF156D: 36
00CF156E: 37
00CF156F: 38
00CF1570: 39
00CF1571: 00
00CF1572: 61
00CF1573: 41
00CF1574: 70
00CF1575: 70
00CF1576: 41
00CF1577: 42
```

C:\Users\asus\Desktop\test2.exe

```
addr:00D41568
00D41564: ffffffff3
00D41565: 09
00D41566: 00
00D41567: 0e
00D41568: 31
00D41569: 32
00D4156A: 33
00D4156B: 34
00D4156C: 35
00D4156D: 36
00D4156E: 37
00D4156F: 38
00D41570: 39
00D41571: 00
00D41572: 61
00D41573: 41
00D41574: 70
00D41575: 70
00D41576: 41
00D41577: 42
```

```
addr:0x187d02a0
0x187d029c: 00
0x187d029d: 00
0x187d029e: 00
0x187d029f: 00
0x187d02a0: 31
0x187d02a1: 32
0x187d02a2: 33
0x187d02a3: 34
0x187d02a4: 35
0x187d02a5: 36
0x187d02a6: 37
0x187d02a7: 38
0x187d02a8: 39
0x187d02a9: 00
0x187d02aa: 61
0x187d02ab: 00
0x187d02ac: 00
0x187d02ad: 00
0x187d02ae: 41
0x187d02af: 42
[u2351114@oop ~]$
```

SSH2 vterm

VS

x86/debug

x86/release

dev

32bit-debug

32bit-release

linux

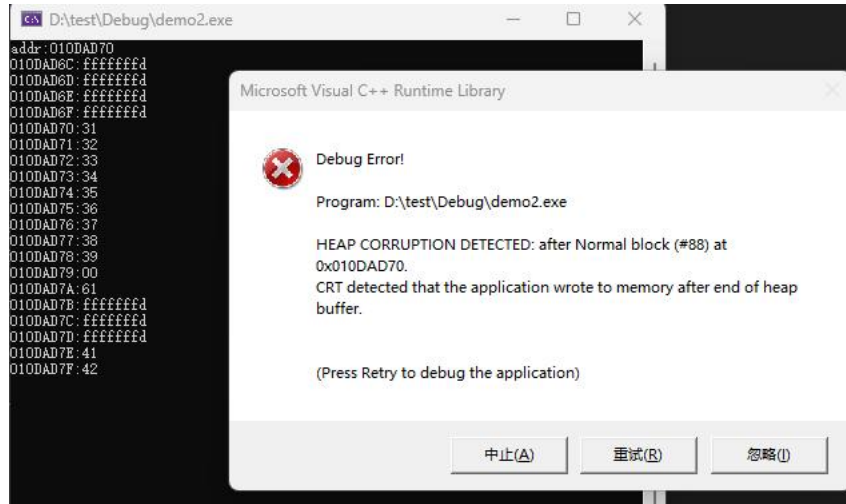
在VS2022的x86/Debug模式下运行:
2、①放开, ②③注释, 观察运行结果
结论: VS的Debug模式是如何判断
动态申请内存访问越界的?

程序正常运行输出。越界写操作成功写入, 越界读操作正常读出,
未赋值的地址中为随机值(vs)
linux中是0



§. 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**）



VS		dev		
x86/debug	x86/release	32bit-debug	32bit-release	linux

在VS2022的x86/Debug模式下运行：
3、①③放开，②注释，观察运行结果

程序运行出现错误，释放了一个没有安排空间的内容



§. 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)

<pre>addr:0097AD20 0097AD1C:fffffffd 0097AD1D:fffffffd 0097AD1E:fffffffd 0097AD1F:fffffffd 0097AD20:31 0097AD21:32 0097AD22:33 0097AD23:34 0097AD24:35 0097AD25:36 0097AD26:37 0097AD27:38 0097AD28:39 0097AD29:00 0097AD2A:fffffffd 0097AD2B:fffffffd 0097AD2C:fffffffd 0097AD2D:fffffffd 0097AD2E:41 0097AD2F:42</pre>	<pre>addr:010CB300 010CB2FC:40 010CB2FD:20 010CB2FE:00 010CB2FF:ffffff8e 010CB300:31 010CB301:32 010CB302:33 010CB303:34 010CB304:35 010CB305:36 010CB306:37 010CB307:38 010CB308:39 010CB309:00 010CB30A:fffffffd 010CB30B:00 010CB30C:ffffffb0 010CB30D:ffffff98 010CB30E:41 010CB30F:42</pre>	<pre>addr:00DB1558 00DB1554:26 00DB1555:79 00DB1556:00 00DB1557:0e 00DB1558:31 00DB1559:32 00DB155A:33 00DB155B:34 00DB155C:35 00DB155D:36 00DB155E:37 00DB155F:38 00DB1560:39 00DB1561:00 00DB1562:fffffffd 00DB1563:53 00DB1564:20 00DB1565:43 00DB1566:41 00DB1567:42</pre>	<pre>D:\test\demo2\testc.exe addr:00BA1558 00BA1554:ffffffd8 00BA1555:08 00BA1556:00 00BA1557:0e 00BA1558:31 00BA1559:32 00BA155A:33 00BA155B:34 00BA155C:35 00BA155D:36 00BA155E:37 00BA155F:38 00BA1560:39 00BA1561:00 00BA1562:fffffffd 00BA1563:53 00BA1564:20 00BA1565:43 00BA1566:41 00BA1567:42</pre>	<pre>[u2351114@oop ~]\$./textc addr:0x122d02a0 0x122d029c:00 0x122d029d:00 0x122d029e:00 0x122d029f:00 0x122d02a0:31 0x122d02a1:32 0x122d02a2:33 0x122d02a3:34 0x122d02a4:35 0x122d02a5:36 0x122d02a6:37 0x122d02a7:38 0x122d02a8:39 0x122d02a9:00 0x122d02aa:fd 0x122d02ab:00 0x122d02ac:00 0x122d02ad:00 0x122d02ae:41 0x122d02af:42</pre>
--	--	--	--	---

VS		dev		
x86/debug	x86/release	32bit-debug	32bit-release	linux

在VS2022的x86/Debug模式下运行:
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?

说明: 程序正常运行输出。越界写操作未被写入, 越界读操作正常读出,
未赋值的地址中为随机值(vs)。
linux中是0



§. 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)



§. 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)

```
addr:012F2F90
012F2F8C: ffffffff
012F2F8D: ffffffff
012F2F8E: ffffffff
012F2F8F: ffffffff
012F2F90: 31
012F2F91: 32
012F2F92: 33
012F2F93: 34
012F2F94: 35
012F2F95: 36
012F2F96: 37
012F2F97: 38
012F2F98: 39
012F2F99: 0
012F2F9A: ffffffff
012F2F9B: ffffffff
012F2F9C: ffffffff
012F2F9D: ffffffff
012F2F9E: 41
012F2F9F: 42
```

```
Microsoft Visual Studio 调试控制台
addr:00F3ED38
00F3ED3A: 0
00F3ED3B: 24
00F3ED3C: 0
00F3ED3D: ffffffff
00F3ED3E: 31
00F3ED3F: 32
00F3ED40: 33
00F3ED41: 34
00F3ED42: 35
00F3ED43: 36
00F3ED44: 37
00F3ED45: 38
00F3ED46: 39
00F3ED47: 42
```

```
C:\Users\asus\Desktop\test1.exe
addr:0xe515a8
0xe515a4: ffffffff83
0xe515a5: 37
0xe515a6: 0
0xe515a7: e
0xe515a8: 31
0xe515a9: 32
0xe515aa: 33
0xe515ab: 34
0xe515ac: 35
0xe515ad: 36
0xe515ae: 37
0xe515af: 38
0xe515b0: 39
0xe515b1: 0
0xe515b2: ffffffff
0xe515b3: 0
0xe515b4: 0
0xe515b5: 0
0xe515b6: 41
0xe515b7: 42
```

```
C:\Users\asus\Desktop\test1.exe
addr:0xe115a8
0xe115a4: 2c
0xe115a5: 50
0xe115a6: 0
0xe115a7: e
0xe115a8: 31
0xe115a9: 32
0xe115aa: 33
0xe115ab: 34
0xe115ac: 35
0xe115ad: 36
0xe115ae: 37
0xe115af: 38
0xe115b0: 39
0xe115b1: 0
0xe115b2: ffffffff
0xe115b3: 0
0xe115b4: 0
0xe115b5: 0
0xe115b6: 41
0xe115b7: 42
```

```
addr:0x2fc21eb0
0x2fc21eac: 0
0x2fc21ead: 0
0x2fc21eae: 0
0x2fc21eaf: 0
0x2fc21eb0: 31
0x2fc21eb1: 32
0x2fc21eb2: 33
0x2fc21eb3: 34
0x2fc21eb4: 35
0x2fc21eb5: 36
0x2fc21eb6: 37
0x2fc21eb7: 38
0x2fc21eb8: 39
0x2fc21eb9: 0
0x2fc21eba: fd
0x2fc21ebb: 0
0x2fc21ebc: 0
0x2fc21ebd: 0
0x2fc21ebe: 41
0x2fc21ebf: 42
[u2351114@oop ~]$
```

VS

x86/debug

x86/release

dev

32bit-debug

32bit-release

linux

在VS2022的x86/Debug模式下运行:

1、①②③全部注释, 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?

说明: 程序正常运行输出。越界写操作未被写入, 越界读操作正常读出,
未赋值的地址中为随机值(vs)。

linux中是0



§. 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)

```
Microsoft Visual Studio 调试控制台
addr: 00EE3278
00EE3274: ffffffff
00EE3275: ffffffff
00EE3276: ffffffff
00EE3277: ffffffff
00EE3278: 31
00EE3279: 32
00EE327A: 33
00EE327B: 34
00EE327C: 35
00EE327D: 36
00EE327E: 37
00EE327F: 38
00EE3280: 39
00EE3281: 0
00EE3282: 61
00EE3283: ffffffff
00EE3284: ffffffff
00EE3285: ffffffff
00EE3286: 41
00EE3287: 42
```

VS		dev		
x86/debug	x86/release	32bit-debug	32bit-release	linux

在VS2022的x86/Debug模式下运行:
2、①放开, ②③注释, 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?

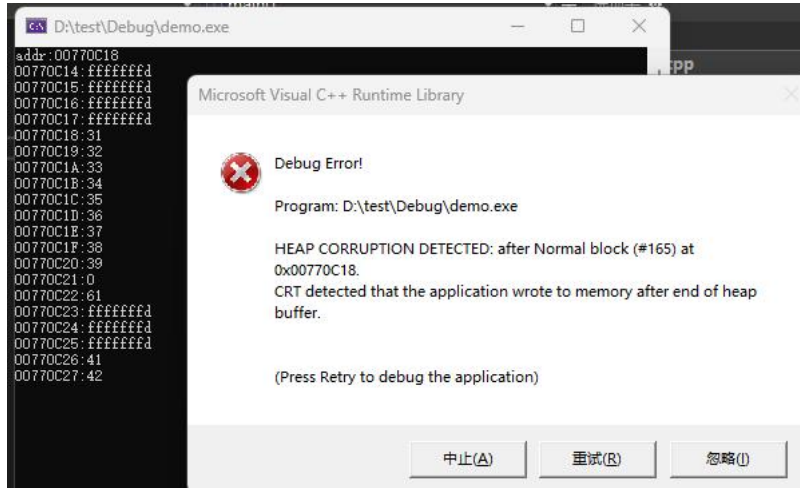
说明: 程序正常运行输出。越界写操作未被写入, 越界读操作正常读出,
未赋值的地址中为随机值(vs)。

linux中是0
和c相似



§. 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)



VS		dev		
x86/debug	x86/release	32bit-debug	32bit-release	linux

在VS2022的x86/Debug模式下运行:
3、①③放开, ②注释, 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?

报错



§. 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)

```
Microsoft Visual Studio 调试控制台
addr: 010F2370
010F236C: ffffffff
010F236D: ffffffff
010F236E: ffffffff
010F236F: ffffffff
010F2370: 31
010F2371: 32
010F2372: 33
010F2373: 34
010F2374: 35
010F2375: 36
010F2376: 37
010F2377: 38
010F2378: 39
010F2379: 0
010F237A: ffffffff
010F237B: ffffffff
010F237C: ffffffff
010F237D: ffffffff
010F237E: 41
010F237F: 42
```

VS		dev		linux
x86/debug	x86/release	32bit-debug	32bit-release	

在VS2022的x86/Debug模式下运行:
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?

说明: 程序正常运行输出。越界写操作未被写入, 越界读操作正常读出,
未赋值的地址中为随机值(vs)。

linux中是0
和c相似



§. 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**）

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char a[10];
    char* p = a;
    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;
}
```

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

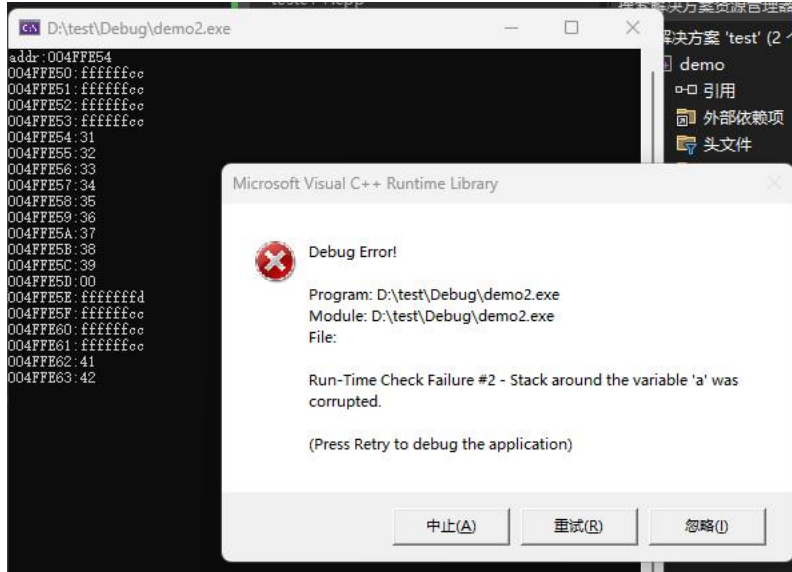
要求：

- 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 x86/Release
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)



§. 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**）



VS

x86/debug

x86/release

dev

32bit-debug

32bit-release

linux

程序报错，并且很多越界的输入没有实现
而不是vs x86 debug的都不会报错



§. 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**）

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char a[10];
    char* p = a;
    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;
}
```

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 x86/Release
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)


```
addr: 012FFB1C
012FFB18: ffffffff0
012FFB19: ffffffff0
012FFB1A: 55
012FFB1B: 1
012FFB1C: 31
012FFB1D: 32
012FFB1E: 33
012FFB1F: 34
012FFB20: 35
012FFB21: 36
012FFB22: 37
012FFB23: 38
012FFB24: 39
012FFB25: 0
012FFB26: 61
012FFB27: 75
012FFB28: 64
012FFB29: 16
012FFB2A: ffffffff0
012FFB2B: 0
```

linux

而不是vs x86 debug的都不会报错



§. 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)

```
#define _CRT_SECURE_NO_WARNINGS
#include
<iostream>
#include <cstring>
using namespace std;
int main()
{
    char a[10];
    char* p = a;
    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;
}
```

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

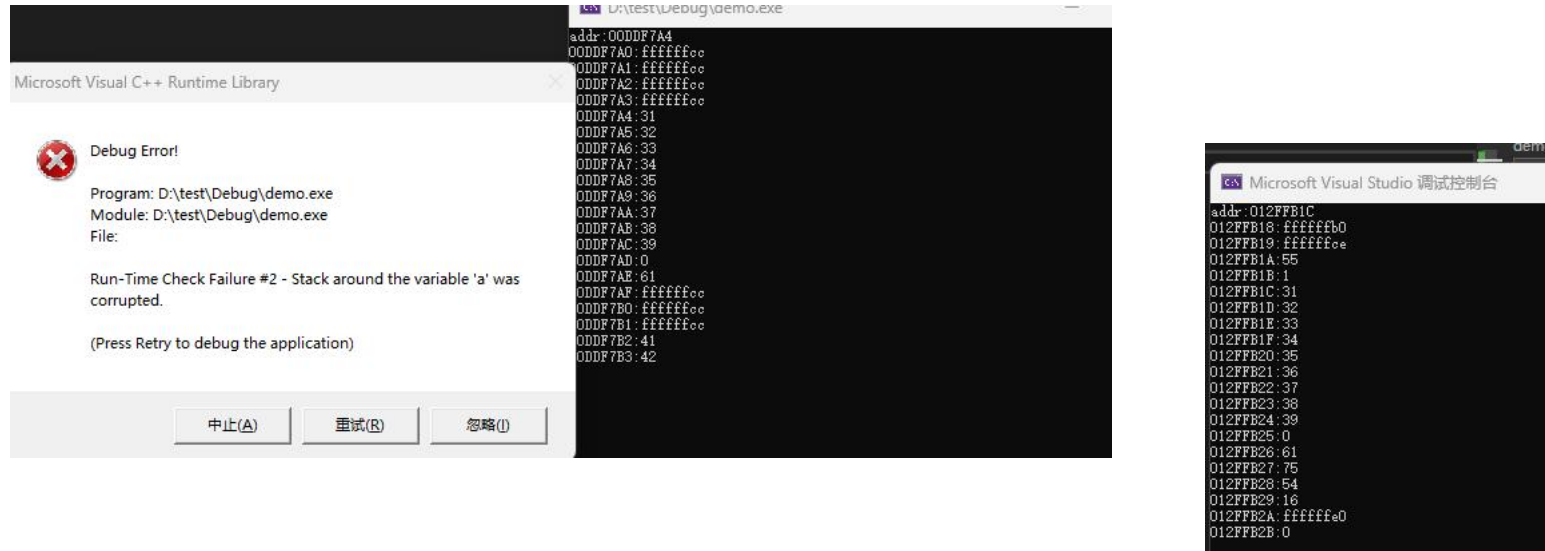
要求:

- 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 x86/Release
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)



§. 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（Cpp方式，**注意源程序后缀为.cpp**）



VS		dev		
x86/debug	x86/release	32bit-debug	32bit-release	linux

vs的debug会报错
而不是vs x86 debug的都不会报错



§. 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)

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    int a[10];
    int i;
    for (i = 0; i < 10; i++) {
        a[i] = i + 1;
    }
    ① a[10] = 10; //此句越界
    a[14] = 20; //此句越界
    a[15] = 30; //此句越界
    ② a[10] = '\xfd'; //此句越界
    cout << "addr:" << hex << (void*)(a) << endl;
    for (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*)(a + i) << ":" << a[i] << endl;
    return 0;
}
```

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

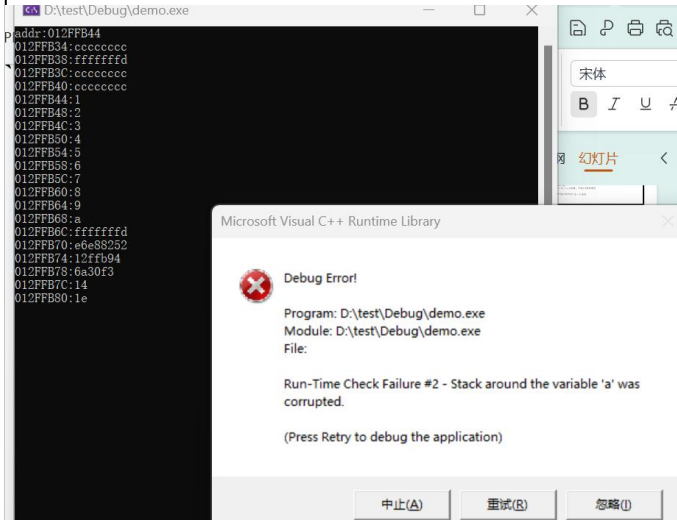
要求:

- 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 x86/Release
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)



§. 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（Cpp方式，**注意源程序后缀为.cpp**）



```
addr:0053F700
0053F6F0: 310a0
0053F6F4: 313f0
0053F6F8: 79c008
0053F6FC: 79cfa8
0053F700: 1
0053F704: 2
0053F708: 3
0053F70C: 4
0053F710: 5
0053F714: 6
0053F718: 7
0053F71C: 8
0053F720: 9
0053F724: a
0053F728: ffffffff
0053F72C: e1149e59
0053F730: 31674
0053F734: 53f77c
0053F738: 315ec
0053F73C: 1
```

```
C:\Users\asus\Desktop\test1.exe
addr:0x78fe94
0x78fe84: 4cf007
0x78fe88: 78ffcc
0x78fe8c: 7714e170
0x78fe90: 6d6de461
0x78fe94: 1
0x78fe98: 2
0x78fe9c: 3
0x78fea0: 4
0x78fea4: 5
0x78fea8: 6
0x78feac: 7
0x78feb0: 8
0x78feb4: 9
0x78feb8: a
0x78febc: a
0x78fec0: 40bc10
0x78fec4: 78fee0
0x78fec8: 78ff68
0x78fecc: 14
0x78fed0: 1e
```

vs

x86/debug

x86/release

dev

32bit-debug

32bit-release

linux

vs的debug会报错

而不是vs x86 debug的都不会报错



§. 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

内存越界是指程序访问了未分配给它的内存区域，可能发生在静态数组或者动态分配的内存块中。由于这些内存区域的分配由操作系统管理，一旦超出界限，后果不可预测，可能覆盖其他重要的数据或导致程序崩溃。

不同环境下的表现：在 Debug 模式下，很多越界问题会通过内存填充、保护机制等被检测到。然而，在Release模式下，这些保护通常会被移除，因此越界行为可能不被察觉，甚至不会引发崩溃，但仍然可能导致数据的不可预测更改或崩溃。

编译器与调试器如何防范越界：

1. Debug 模式：调试模式下，编译器通常会加入额外的内存检查和保护。确保如果写入越界，这些保护字节会被修改，触发调试器警报。
2. 内存填充：在动态分配和释放内存时，Debug 模式往往会用特殊字节（如0xCC或0xFD）填充未使用或释放的内存区域，以检测对未初始化或释放后内存的访问。

为了尽量防范内存越界问题，应该养成良好的习惯：

1. 仔细管理数组与内存大小：避免操作超过分配的内存区域。
2. 使用安全的函数：尽量使用安全版本的函数，如 ``strncpy`` 代替 ``strcpy``，以避免字符串拷贝时发生越界。
3. 边界检查：在操作数组时，检查访问的索引是否在合法范围内。
4. 养成良好的释放内存习惯：在使用malloc或其他内存分配函数后，确保及时free掉分配的内存, 而且注意不要free掉非法部分