§.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）

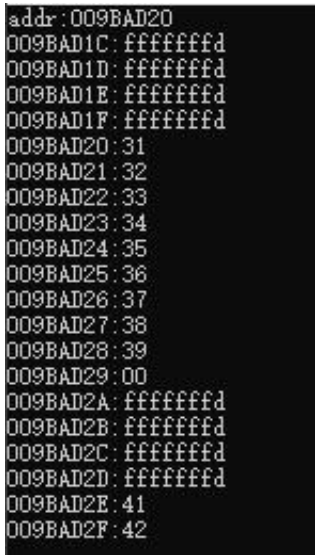
```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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？

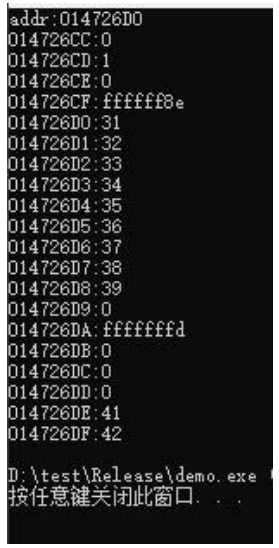
再观察下面四种环境下的运行结果：
VS2022 x86/Release
Dev 32bit-Debug
Dev 32bit-Release
Linux
每种讨论的结果可截图+文字说明，
如果几种环境的结果一致，用一个
环境的截图+文字说明即可(可加页)

§.关于动态内存申请后越界访问的深度讨论

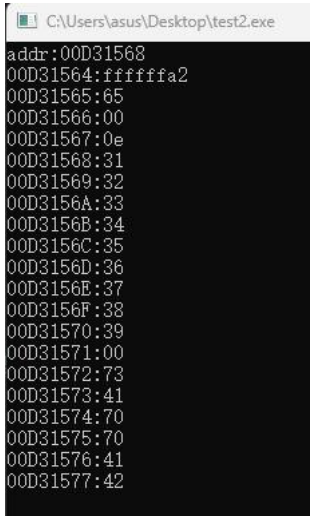
★ 如何判断动态申请越界（C方式，注意源程序后缀为.c）



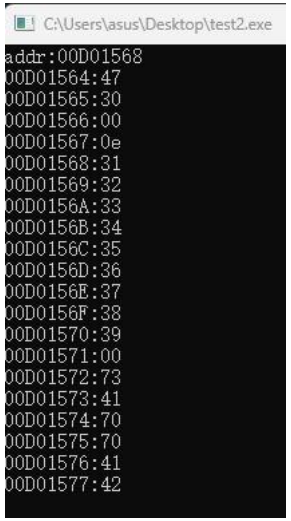
vs
x86/debug



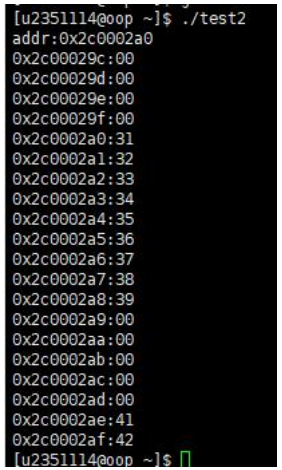
x86/release



dev
32bit-debug



32bit-release



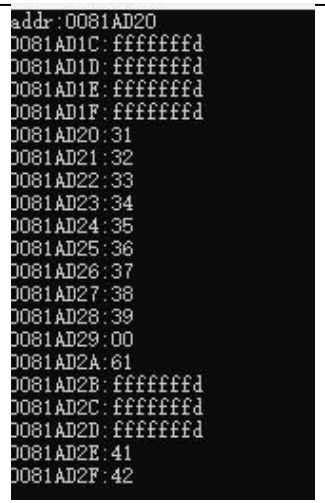
linux

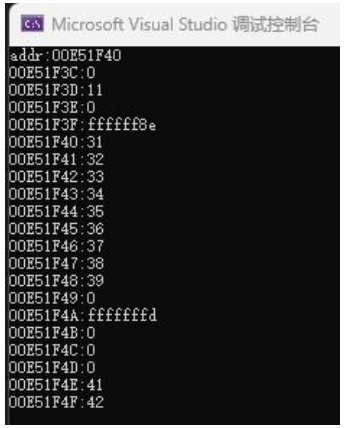
在VS2022的x86/Debug模式下运行：
1、①②③全部注释，观察运行结果
结论：VS的Debug模式是如何判断
动态申请内存访问越界的？

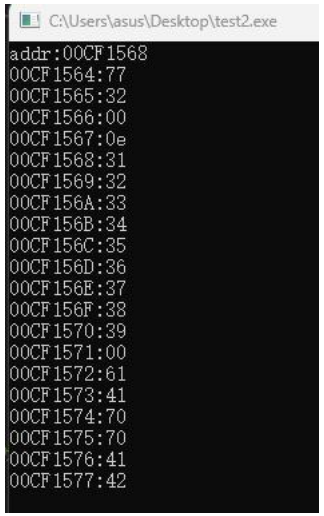
程序正常运行输出。越界写操作成功写入，越界读操作正常读出，
未赋值的地址中为随机值(vs)
linux中是0

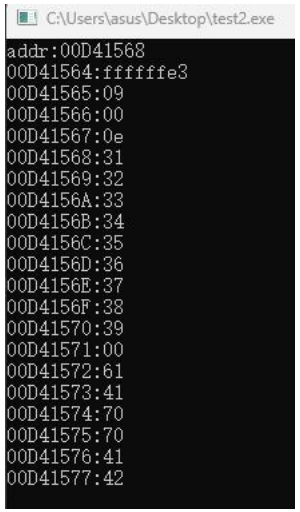
§.关于动态内存申请后越界访问的深度讨论

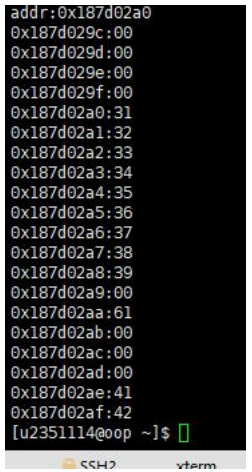
★ 如何判断动态申请越界（C方式，注意源程序后缀为.c）











vs　　dev

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

在VS2022的x86/Debug模式下运行：

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？

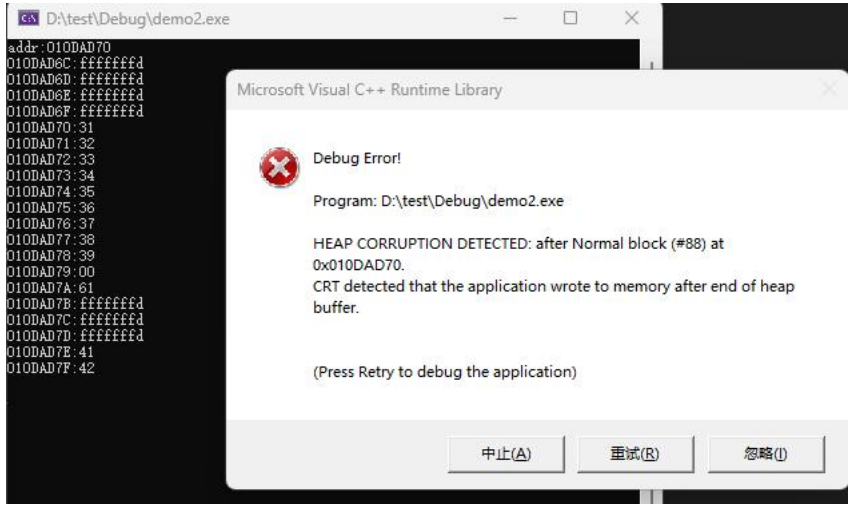
程序正常运行输出。越界写操作成功写入，越界读操作正常读出，

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

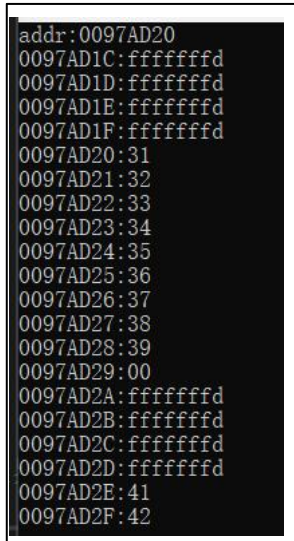
在VS2022的x86/Debug模式下运行：

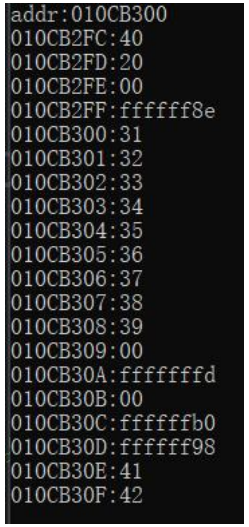
3、①③放开，②注释，观察运行结果

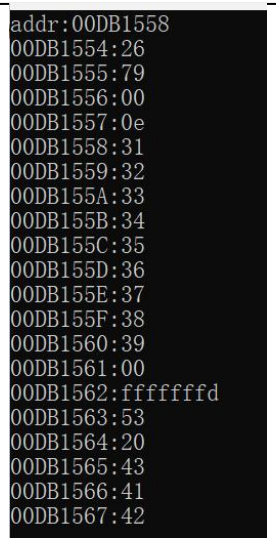
程序运行出现错误，释放了一个没有安排空间的内容

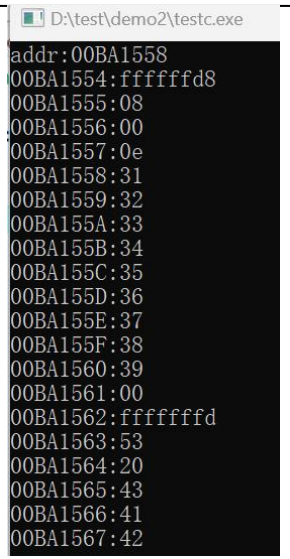
§.关于动态内存申请后越界访问的深度讨论

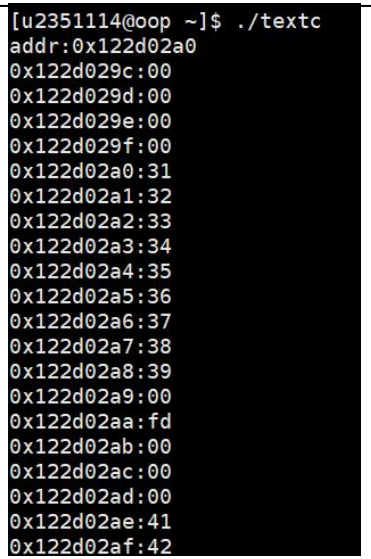
★ 如何判断动态申请越界（C方式，注意源程序后缀为.c）











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）

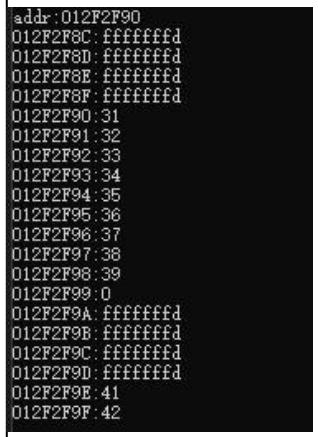
```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

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;
}
```

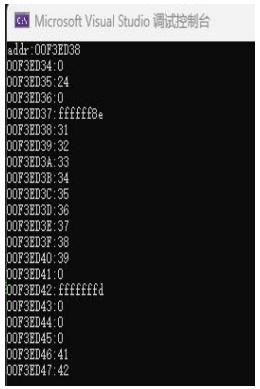
在VS2022的x86/Debug模式下运行：
1、①②③全部注释，观察运行结果
2、①放开，②③注释，观察运行结果
3、①③放开，②注释，观察运行结果
4、①②③全部放开，观察运行结果
结论：VS的Debug模式是如何判断
动态申请内存访问越界的？

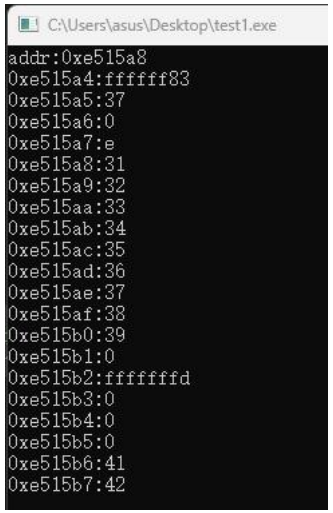
再观察下面四种环境下的运行结果：
VS2022 x86/Release
Dev 32bit-Debug
Dev 32bit-Release
Linux
每种讨论的结果可截图+文字说明，
如果几种环境的结果一致，用一个
环境的截图+文字说明即可(可加页)

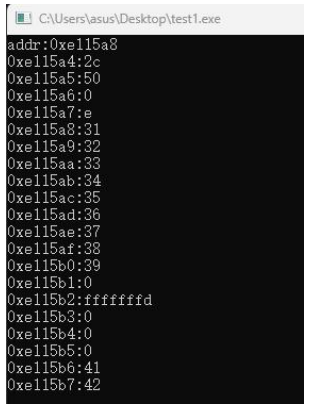
§.关于动态内存申请后越界访问的深度讨论

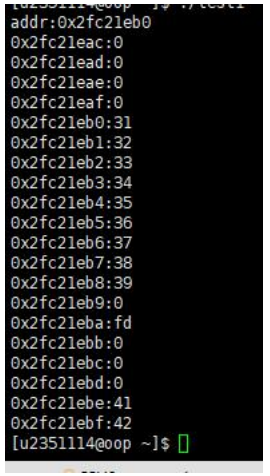
★ 如何判断动态申请越界（C++方式，注意源程序后缀为.cpp）











vs dev

x86/debug x86/release 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

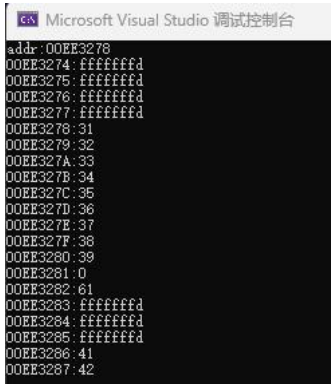
结论：VS的Debug模式是如何判断
动态申请内存访问越界的？

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）



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？

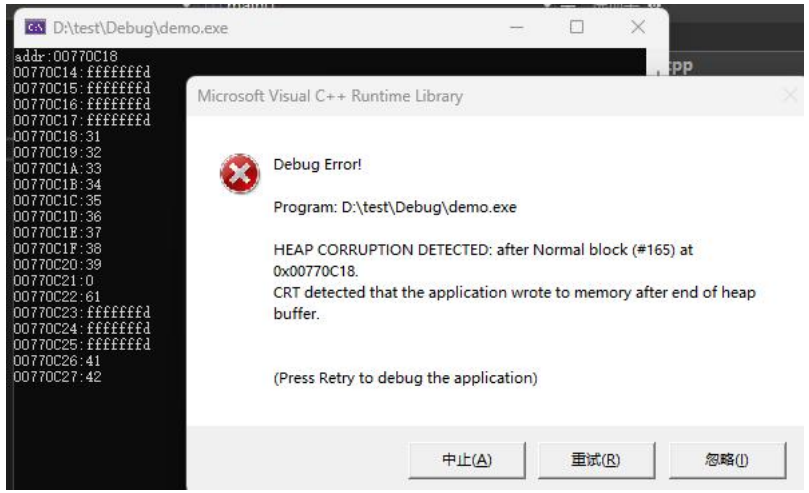
说明：程序正常运行输出。越界写操作未被写入，越界读操作正常读出，未赋值的地址中为随机值(vs)。

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

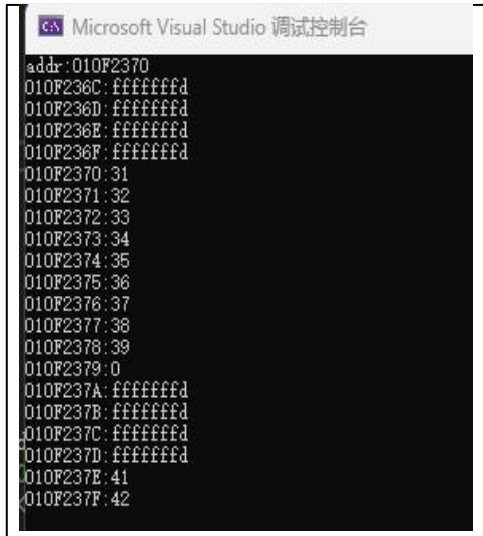
结论：VS的Debug模式是如何判断

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）



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

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

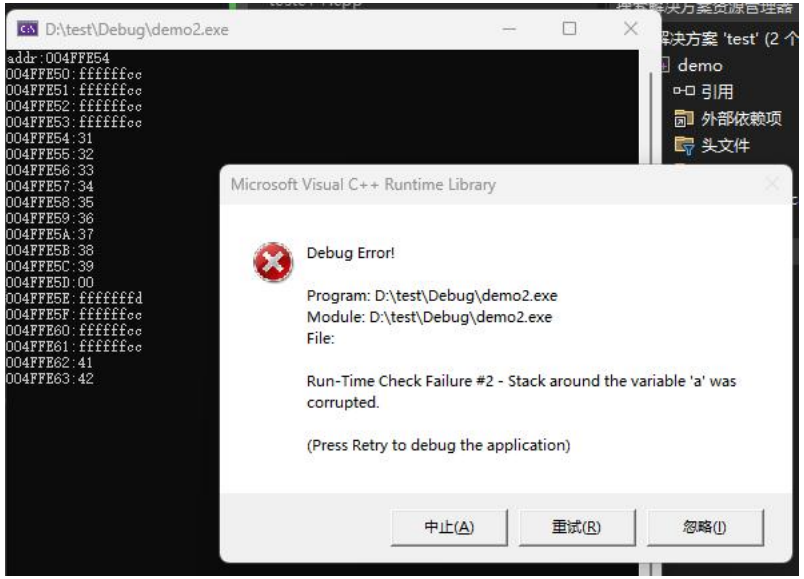
3、测试程序在下面五种环境下运行

- 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 dev

x86/debug x86/release 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

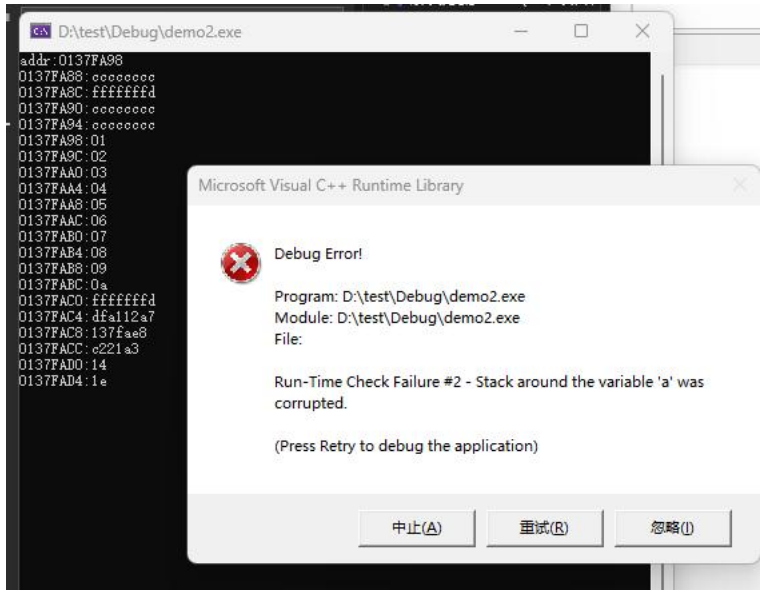
3、测试程序在下面五种环境下运行

- 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)

§.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 dev

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

越界报错了

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

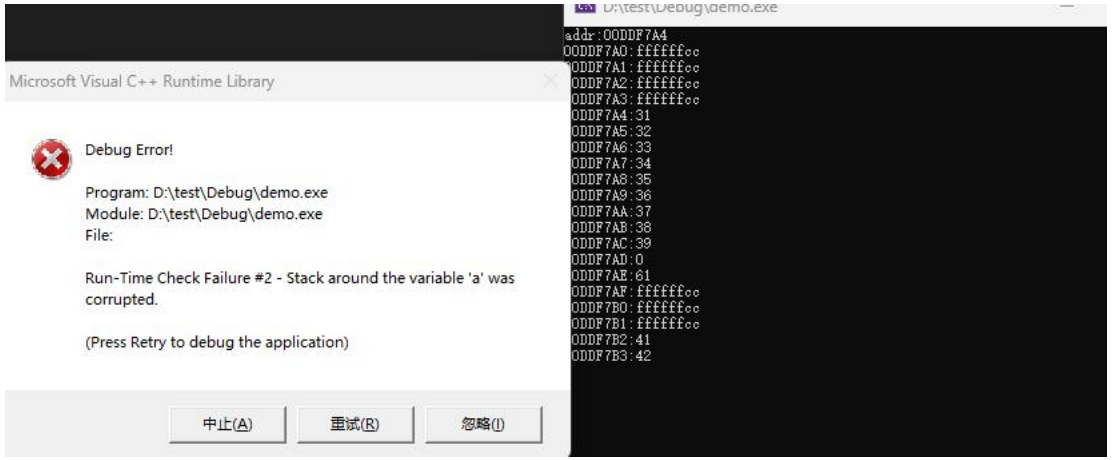
3、测试程序在下面五种环境下运行

- 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

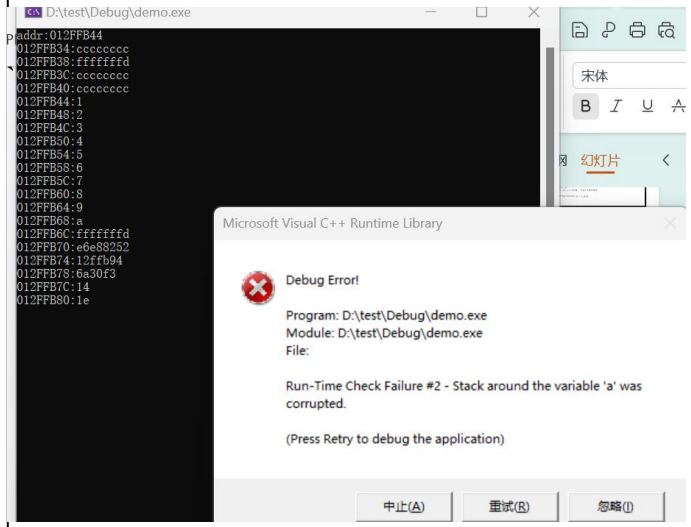
3、测试程序在下面五种环境下运行

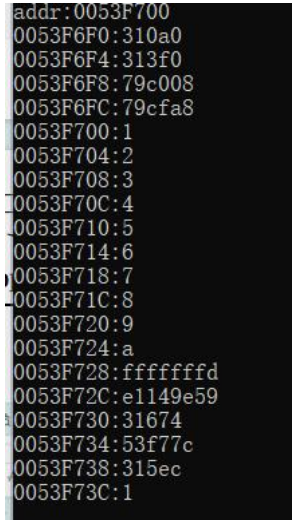
- VS2022 x86/Debug
- VS2022 x86/Release
- Dev 32bit-Debug
- Dev 32bit-Release
- Linux

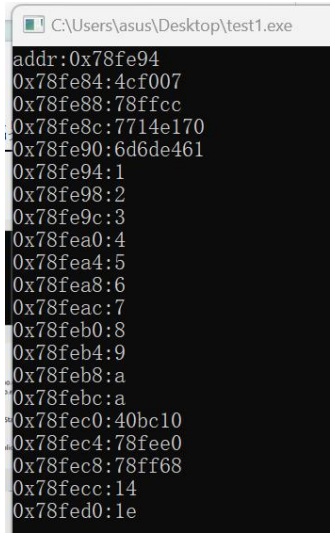
4、每种讨论的结果可截图+文字说明，如果几种环境的结果一致，用一个环境的截图+文字说明即可(可加页)

§.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）







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

★ **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