Assignment 2

1 概述

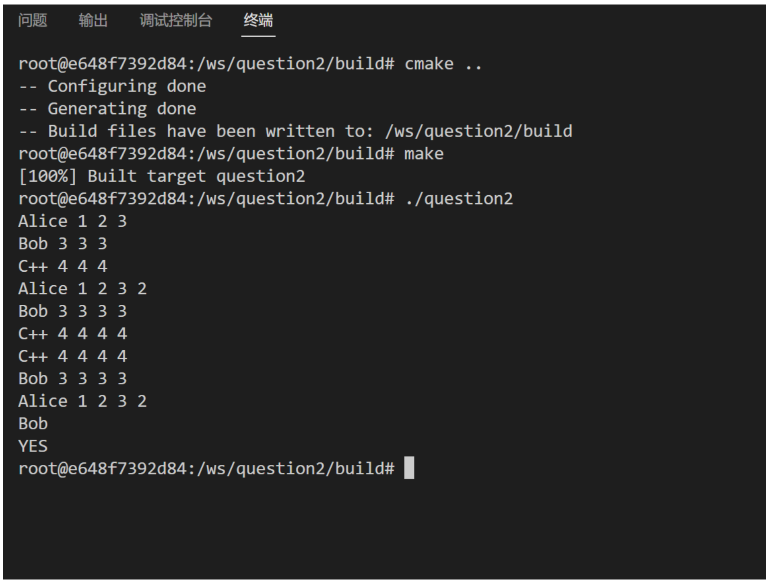
第二次作业。

这次就不是全部截图了(

建议学长下次布置作业下发代码包(复制带行号好痛苦)

最后一题较冗杂，纯折磨人。

2 Makefile & Cmake



```
root@e648f7392d84:/ws/question2/build# cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /ws/question2/build
root@e648f7392d84:/ws/question2/build# make
[100%] Built target question2
root@e648f7392d84:/ws/question2/build# ./question2
Alice 1 2 3
Bob 3 3 3
C++ 4 4 4
Alice 1 2 3 2
Bob 3 3 3 3
C++ 4 4 4 4
C++ 4 4 4 4
Bob 3 3 3 3
Alice 1 2 3 2
Bob
YES
root@e648f7392d84:/ws/question2/build# 
```

基于最简单的cmake打包方式，文件夹内容如下:

question2

----build

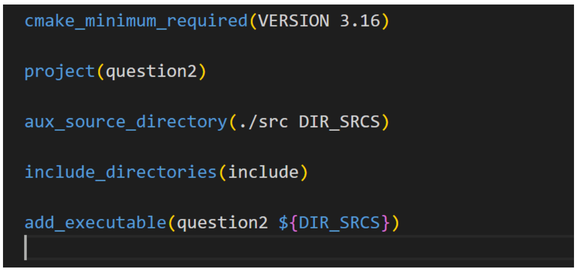
----CMakelists.txt

----main.cpp

----stuinfo.cpp

----stuinfo.h

把头文件独立出来的打包方式:



```
cmake_minimum_required(VERSION 3.16)

project(question2)

aux_source_directory(./src DIR_SRCS)

include_directories(include)

add_executable(question2 ${DIR_SRCS})
```

文件内容:

question2

---build

---src

------main.cpp

------stuinfo.cpp

---include

-------stuinfo.h

3 Types

3.1 static 用法和作用

用 static 修饰局部变量时，被修饰的变量变为静态变量，在main函数之前初始化，在程序退出时销毁。

用 static 修饰全局变量，只能被static修饰的全局变量只能被该包含该定义的文件访问

static 修饰函数使得函数只能在包含该函数定义的文件中被调用。

static 修饰成员变量时，该变量不属于哪一个具体的对象，而属于整个类，当某个对象修改该变量时，也会影响到其他对象。静态成员变量的内存分配在（类外）初始化时分配。static 成员变量不占用对象的内存，而是在所有对象之外开辟内存，即使不创建对象也可以访问。

static 修饰成员函数时，静态成员函数只能访问静态成员。静态成员函数没有 this 指针，不指向哪个对象，无法访问对象的成员变量。

3.2 隐式转换

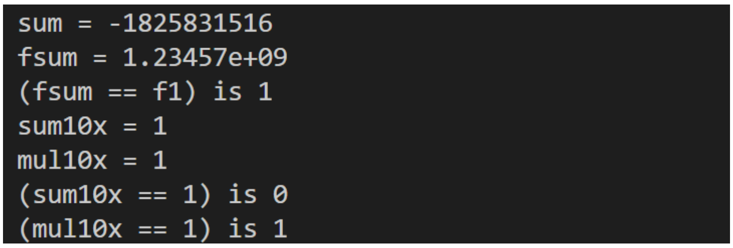
隐式转换是 C++ 内部默认的强制类型转换方式，当一个表达包含若干个不同类型的变量时， C++ 会自动对其进行类型转换。例如，在对变量赋值时，若等号两边的数据类型不同，需要把右边表达式的类型转换为左边变量的类型；在运算过程中，不同类型的数据需要转换为同一类；在函数调用中，传入函数的变量会自动转换为函数定义的类型。一般来说，隐式转换的转换规则如下：转换按数据长度增加的方向进行，以保证数据精度不降低；在条件判断中，非布尔类型自动转换为布尔类型；当有符号类型转换为无符号类型时，其值可能发生变化。

关于隐式转换，我们在写表达式时尽可能使用显示转换来规避这个问题。或者用宏 EXPLICIT 关闭函数的隐式转换。在写函数时使用模板函数替代。

3.3 程序解释

3.3.1 程序1

运行结果如下：



```
sum = -1825831516
fsum = 1.23457e+09
(fsum == f1) is 1
sum10x = 1
mul10x = 1
(sum10x == 1) is 0
(mul10x == 1) is 1
```

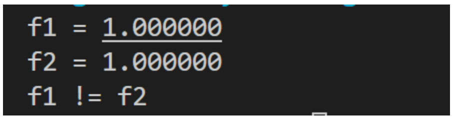
对于 sum ， 由于 num1 + num2 爆 int 了，答案自溢出了。

对于 fsum , 由于fsum,f1,f2都是 float，f1在存1234567890时丢失了精度，在进行fsum=f1+f2 时也丢失了精度，fsum 在当前精度下和 f1 是相同的。

由于计算机存储小数是二进制存储，在存0.1时是丢失了精度的，而进行加法会加大这种精度误差，从而导致最后不等于1，而乘法精度丢失更小。而在cout输出里是会自动四舍五入的。

3.3.2 程序2

运行结果如下：

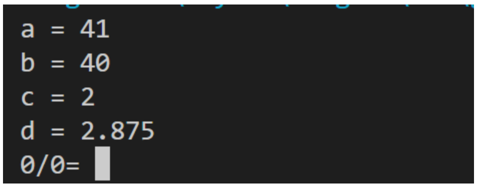


```
f1 = 1.000000
f2 = 1.000000
f1 != f2
```

cout输出会自动四舍五入，所以输出f2会是1.000000，而他们在二进制存储上是不一样的。

3.3.3 程序3

运行结果如下：



```
a = 41
b = 40
c = 2
d = 2.875
0/0=
```

对于a，由于a是整数，后面是浮点运算，会先进行浮点运算然后自动丢弃小数点后的值。

对于b，由于后面有强制类型转换，所以实际上就是19+21。

对于c，整数除法是向下取整。

对于d，由于运算式子存在浮点数，运算过程的值会自动转换为浮点数。

对于 0/0，这会run time error，也有可能会输出 nan 吧。

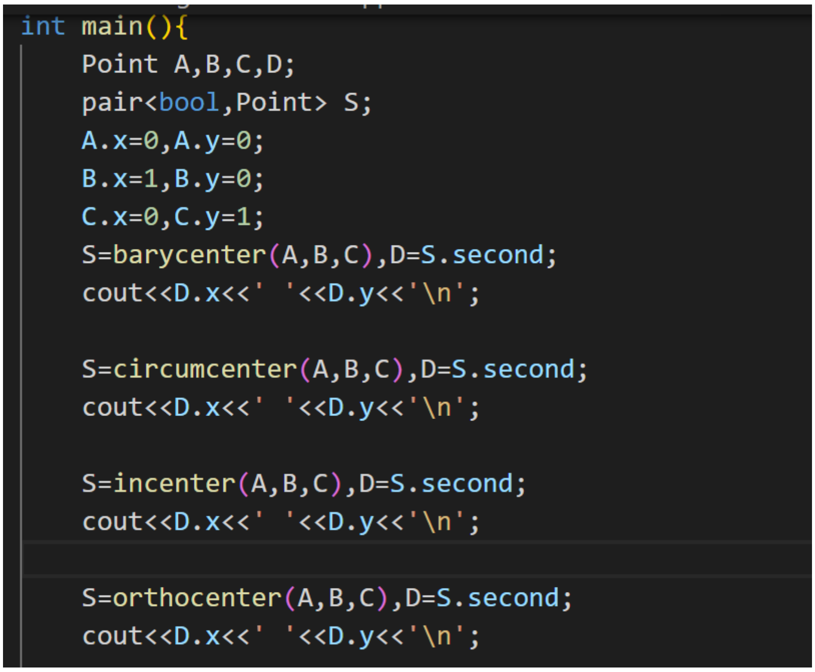
4 Structs

4.1 结构体对齐

alignas定义的对齐大小大于等于结构体内所有成员的大小时，可以生效，反之不能。

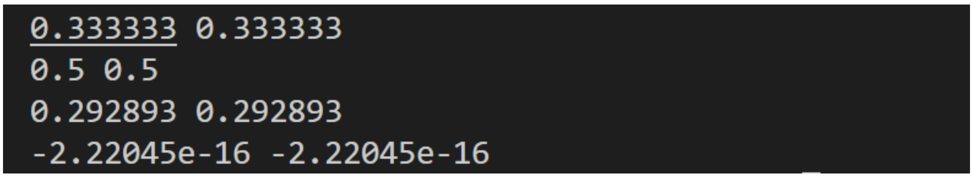
4.2 Exercise

main.cpp:



```
int main(){
    Point A,B,C,D;
    pair<bool,Point> S;
    A.x=0,A.y=0;
    B.x=1,B.y=0;
    C.x=0,C.y=1;
    S=barycenter(A,B,C),D=S.second;
    cout<<D.x<<' '<<D.y<<'\n';

    S=circumcenter(A,B,C),D=S.second;
    cout<<D.x<<' '<<D.y<<'\n';

    S=incenter(A,B,C),D=S.second;
    cout<<D.x<<' '<<D.y<<'\n';

    S=orthocenter(A,B,C),D=S.second;
    cout<<D.x<<' '<<D.y<<'\n';
```

运行结果:



```
0.333333 0.333333
0.5 0.5
0.292893 0.292893
-2.22045e-16 -2.22045e-16
```

(看起来第四个应该是(0,0))

5 C++ 动态内存申请

5.1 内存分区

(1)static定义的全局变量应该在全局/静态存储区

(2)auto变量在栈区

(3)字符数组在栈

(4)字符指针在栈，字符常量在静态存储区

(5)malloc申请的空间在堆，p1变量本身在栈

(6)new申请的空间在自由存储区，a本身在栈

5.2 问答题

1. new 和 malloc 的区别

new操作符从自由存储区上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。

new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配；而malloc内存分配成功则是返回void * ，需要通过强制类型转换将void*指针转换成我们需要的类型

new内存分配失败时，会抛出bac_alloc异常，它不会返回NULL；malloc分配内存失败时返回NULL

使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算，而malloc则需要显式地指出所需内存的尺寸

使用new[]分配的内存必须使用delete[]进行释放

operator new /operator delete的实现可以基于malloc，而malloc的实现不可以去调用new

2. delete p、delete[ ] p、allocator 都有什么作用?

delete p会调用一次析构函数，而delete[] p会调用每个成员的析构函数，对于普通数据类型而言，他们作用的效果是一样的；如果数组类型是自定义类，那么new[]只能用delete[]来对应，new和delete对应；

allocator类是C++的一个模板，它提供类型化的内存分配以及对象的分配和撤销。

3.malloc 申请的存储空间能用 delete 释放吗?

可以的。new 和delete会自动进行类型检查和大小。

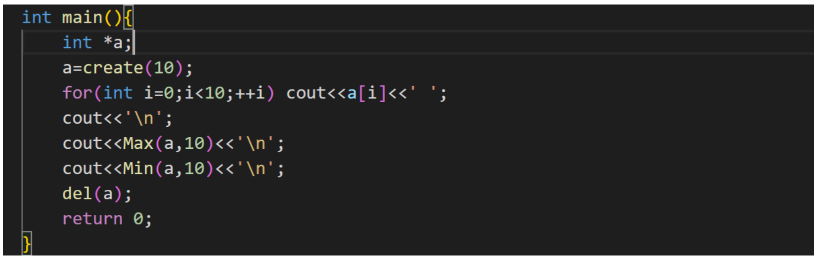
4. malloc 与 free 的实现原理?

malloc采用的是内存池的实现方式，先申请一大块内存，然后将内存分成不同大小的内存块，然后用户申请内存时，直接从内存池中选择一块相近的内存块即可。具体而言，在堆中申请空间会遍历空闲链表，然后就寻找第一个空间大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。

free函数将用户释放的内存块连接到空闲链表上。

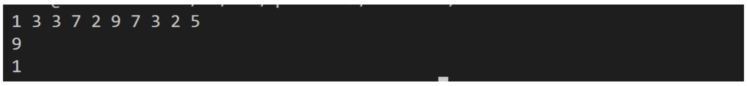
5.3 Exercise

主函数:



```
int main(){
    int *a;
    a=create(10);
    for(int i=0;i<10;++i) cout<<a[i]<<' ';
    cout<<'\n';
    cout<<Max(a,10)<<'\n';
    cout<<Min(a,10)<<'\n';
    del(a);
    return 0;
}
```

运行结果如下:



```
1 3 3 7 2 9 7 3 2 5
9
1
```

具体代码见:question5

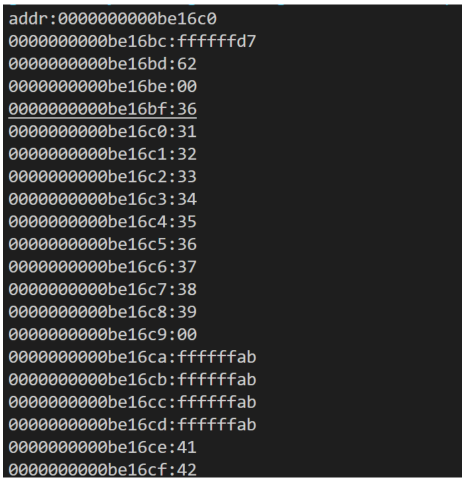
6 Debug 和 Release

6.1 如何判断动态申请越界（C 方式，注意源程序后缀为.c）

VS2022 的 x86/Debug，Release 模式:

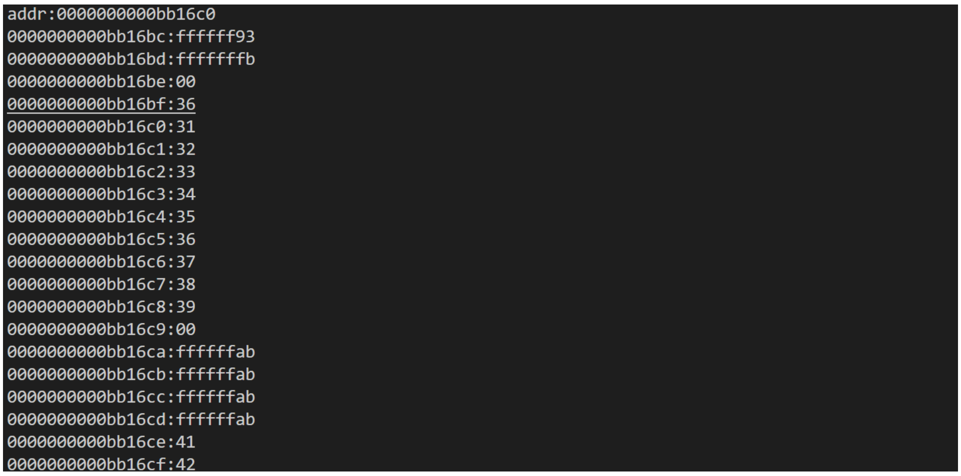
结果1

Release



```
addr:0000000000be16c0
0000000000be16bc:ffffffd7
0000000000be16bd:62
0000000000be16be:00
0000000000be16bf:36
0000000000be16c0:31
0000000000be16c1:32
0000000000be16c2:33
0000000000be16c3:34
0000000000be16c4:35
0000000000be16c5:36
0000000000be16c6:37
0000000000be16c7:38
0000000000be16c8:39
0000000000be16c9:00
0000000000be16ca:ffffffab
0000000000be16cb:ffffffab
0000000000be16cc:ffffffab
0000000000be16cd:ffffffab
0000000000be16ce:41
0000000000be16cf:42
```

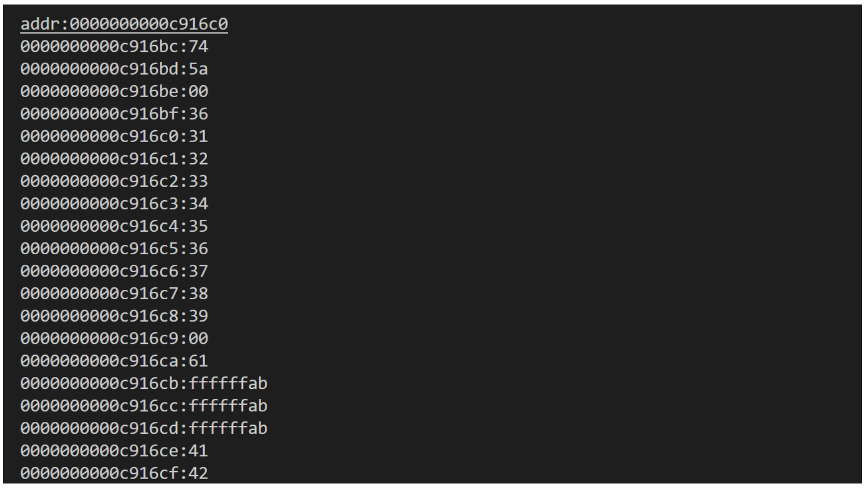
Debug



```
addr:0000000000bb16c0
0000000000bb16bc:ffffff93
0000000000bb16bd:fffffffb
0000000000bb16be:00
0000000000bb16bf:36
0000000000bb16c0:31
0000000000bb16c1:32
0000000000bb16c2:33
0000000000bb16c3:34
0000000000bb16c4:35
0000000000bb16c5:36
0000000000bb16c6:37
0000000000bb16c7:38
0000000000bb16c8:39
0000000000bb16c9:00
0000000000bb16ca:ffffffab
0000000000bb16cb:ffffffab
0000000000bb16cc:ffffffab
0000000000bb16cd:ffffffab
0000000000bb16ce:41
0000000000bb16cf:42
```

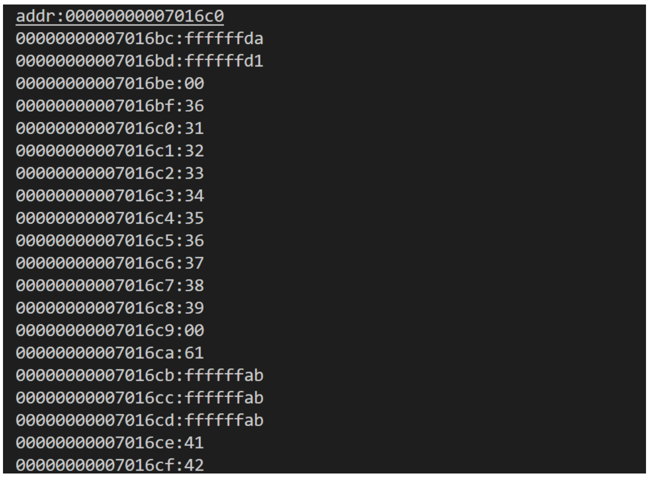
结果2

Release



```
addr:0000000000c916c0
0000000000c916bc:74
0000000000c916bd:5a
0000000000c916be:00
0000000000c916bf:36
0000000000c916c0:31
0000000000c916c1:32
0000000000c916c2:33
0000000000c916c3:34
0000000000c916c4:35
0000000000c916c5:36
0000000000c916c6:37
0000000000c916c7:38
0000000000c916c8:39
0000000000c916c9:00
0000000000c916ca:61
0000000000c916cb:ffffffab
0000000000c916cc:ffffffab
0000000000c916cd:ffffffab
0000000000c916ce:41
0000000000c916cf:42
```

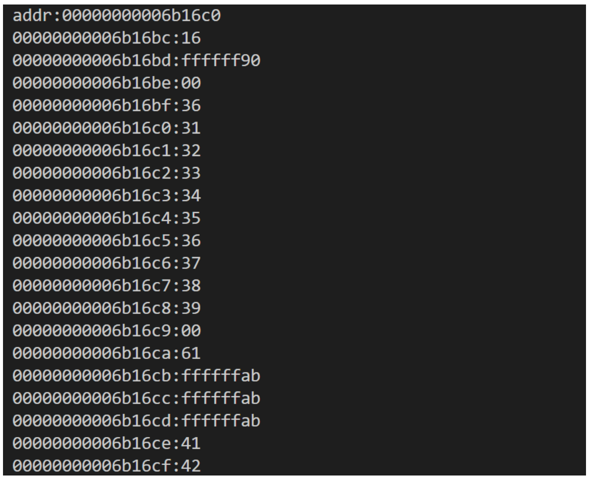
Debug



```
addr:00000000007016c0
00000000007016bc:ffffffda
00000000007016bd:ffffffd1
00000000007016be:00
00000000007016bf:36
00000000007016c0:31
00000000007016c1:32
00000000007016c2:33
00000000007016c3:34
00000000007016c4:35
00000000007016c5:36
00000000007016c6:37
00000000007016c7:38
00000000007016c8:39
00000000007016c9:00
00000000007016ca:61
00000000007016cb:ffffffab
00000000007016cc:ffffffab
00000000007016cd:ffffffab
00000000007016ce:41
00000000007016cf:42
```

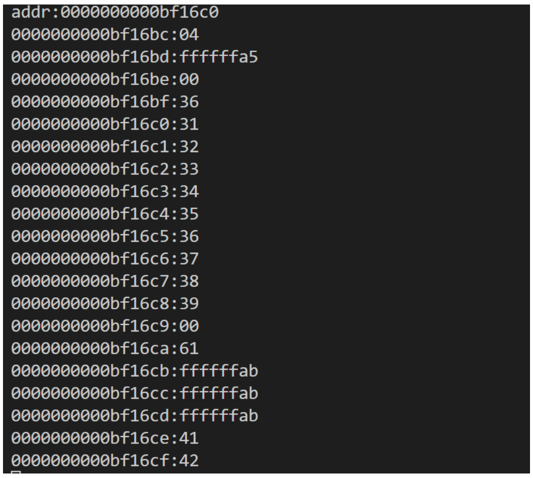
结果3

Release



```
addr:00000000006b16c0
00000000006b16bc:16
00000000006b16bd:ffffff90
00000000006b16be:00
00000000006b16bf:36
00000000006b16c0:31
00000000006b16c1:32
00000000006b16c2:33
00000000006b16c3:34
00000000006b16c4:35
00000000006b16c5:36
00000000006b16c6:37
00000000006b16c7:38
00000000006b16c8:39
00000000006b16c9:00
00000000006b16ca:61
00000000006b16cb:ffffffab
00000000006b16cc:ffffffab
00000000006b16cd:ffffffab
00000000006b16ce:41
00000000006b16cf:42
```

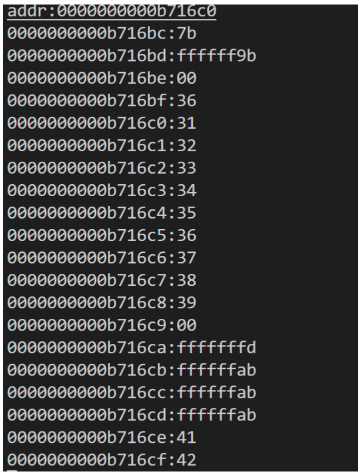
Debug



```
addr:0000000000bf16c0
0000000000bf16bc:04
0000000000bf16bd:ffffffa5
0000000000bf16be:00
0000000000bf16bf:36
0000000000bf16c0:31
0000000000bf16c1:32
0000000000bf16c2:33
0000000000bf16c3:34
0000000000bf16c4:35
0000000000bf16c5:36
0000000000bf16c6:37
0000000000bf16c7:38
0000000000bf16c8:39
0000000000bf16c9:00
0000000000bf16ca:61
0000000000bf16cb:ffffffab
0000000000bf16cc:ffffffab
0000000000bf16cd:ffffffab
0000000000bf16ce:41
0000000000bf16cf:42
```

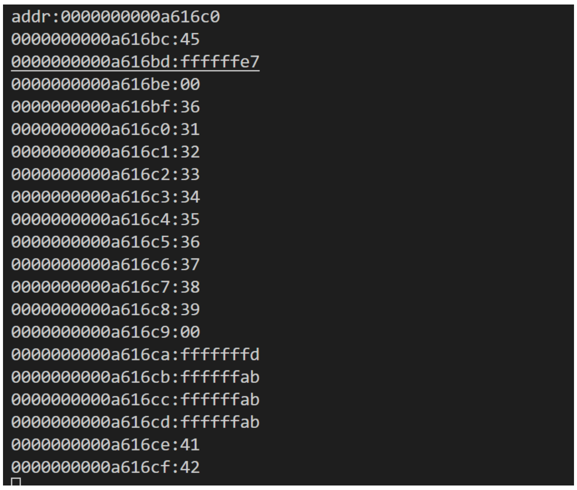
结果4

Release



```
addr:0000000000b716c0
0000000000b716bc:7b
0000000000b716bd:ffffff9b
0000000000b716be:00
0000000000b716bf:36
0000000000b716c0:31
0000000000b716c1:32
0000000000b716c2:33
0000000000b716c3:34
0000000000b716c4:35
0000000000b716c5:36
0000000000b716c6:37
0000000000b716c7:38
0000000000b716c8:39
0000000000b716c9:00
0000000000b716ca:fffffffd
0000000000b716cb:ffffffab
0000000000b716cc:ffffffab
0000000000b716cd:ffffffab
0000000000b716ce:41
0000000000b716cf:42
```

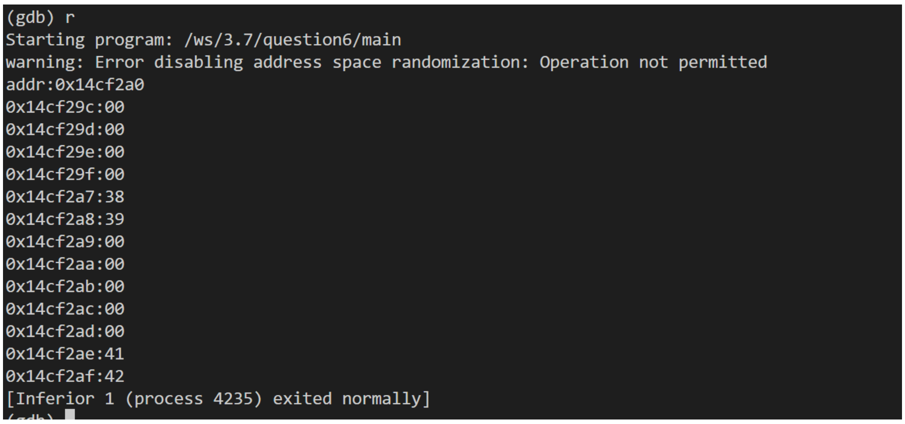
Debug



```
addr:0000000000a616c0
0000000000a616bc:45
0000000000a616bd:ffffffe7
0000000000a616be:00
0000000000a616bf:36
0000000000a616c0:31
0000000000a616c1:32
0000000000a616c2:33
0000000000a616c3:34
0000000000a616c4:35
0000000000a616c5:36
0000000000a616c6:37
0000000000a616c7:38
0000000000a616c8:39
0000000000a616c9:00
0000000000a616ca:fffffffd
0000000000a616cb:ffffffab
0000000000a616cc:ffffffab
0000000000a616cd:ffffffab
0000000000a616ce:41
0000000000a616cf:42
```

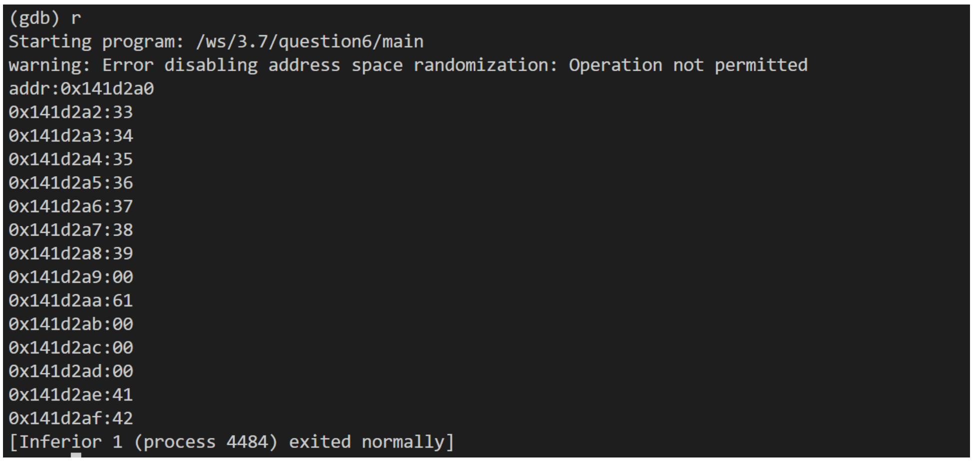
linux下GDB：

结果1



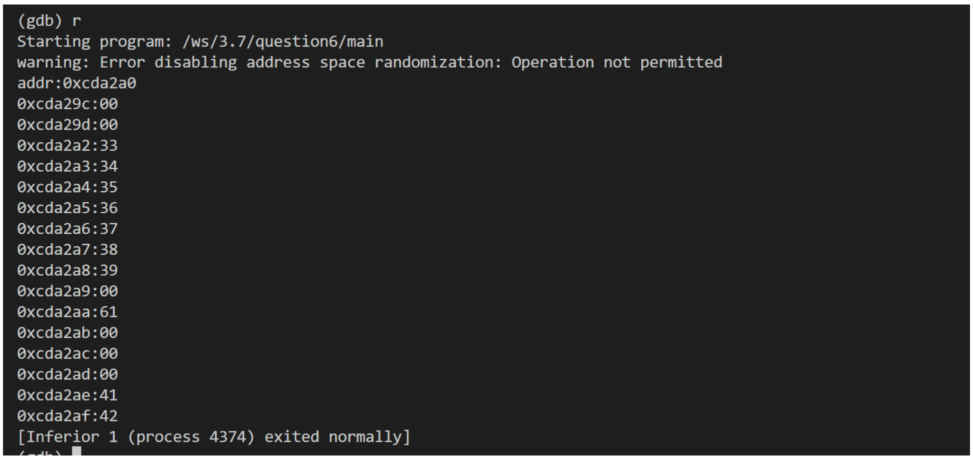
```
(gdb) r
Starting program: /ws/3.7/question6/main
warning: Error disabling address space randomization: Operation not permitted
addr:0x14cf2a0
0x14cf29c:00
0x14cf29d:00
0x14cf29e:00
0x14cf29f:00
0x14cf2a7:38
0x14cf2a8:39
0x14cf2a9:00
0x14cf2aa:00
0x14cf2ab:00
0x14cf2ac:00
0x14cf2ad:00
0x14cf2ae:41
0x14cf2af:42
[Inferior 1 (process 4235) exited normally]
(gdb)
```

结果2



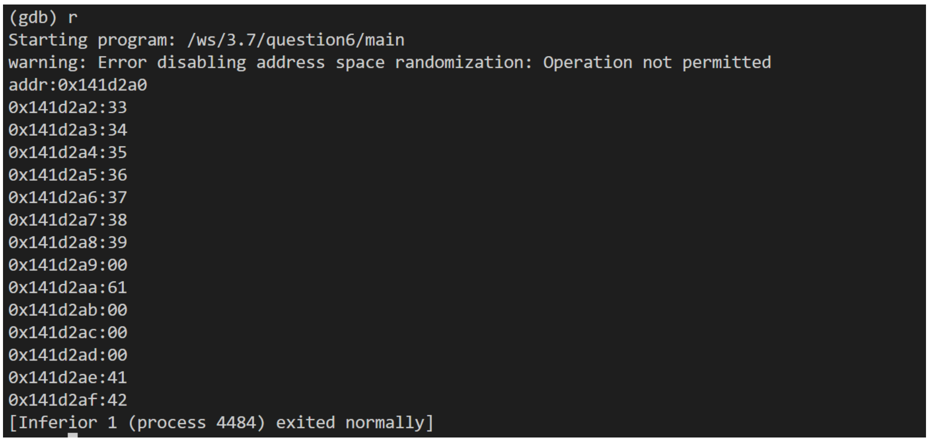
```
(gdb) r
Starting program: /ws/3.7/question6/main
warning: Error disabling address space randomization: Operation not permitted
addr:0x141d2a0
0x141d2a2:33
0x141d2a3:34
0x141d2a4:35
0x141d2a5:36
0x141d2a6:37
0x141d2a7:38
0x141d2a8:39
0x141d2a9:00
0x141d2aa:61
0x141d2ab:00
0x141d2ac:00
0x141d2ad:00
0x141d2ae:41
0x141d2af:42
[Inferior 1 (process 4484) exited normally]
```

结果3



```
(gdb) r
Starting program: /ws/3.7/question6/main
warning: Error disabling address space randomization: Operation not permitted
addr:0xcda2a0
0xcda29c:00
0xcda29d:00
0xcda2a2:33
0xcda2a3:34
0xcda2a4:35
0xcda2a5:36
0xcda2a6:37
0xcda2a7:38
0xcda2a8:39
0xcda2a9:00
0xcda2aa:61
0xcda2ab:00
0xcda2ac:00
0xcda2ad:00
0xcda2ae:41
0xcda2af:42
[Inferior 1 (process 4374) exited normally]
```

结果4



```
(gdb) r
Starting program: /ws/3.7/question6/main
warning: Error disabling address space randomization: Operation not permitted
addr:0x141d2a0
0x141d2a2:33
0x141d2a3:34
0x141d2a4:35
0x141d2a5:36
0x141d2a6:37
0x141d2a7:38
0x141d2a8:39
0x141d2a9:00
0x141d2aa:61
0x141d2ab:00
0x141d2ac:00
0x141d2ad:00
0x141d2ae:41
0x141d2af:42
[Inferior 1 (process 4484) exited normally]
```

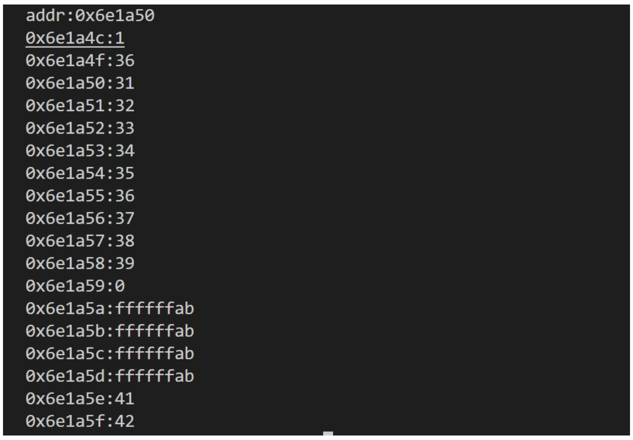
总结：在windows下，数组越界的位置值为ffffffxx来判断越界，负下标为随机的值(规律很诡异，-1为字符‘6’，-2为空)，但能给越界的位置赋值，Linux GDB会报warning，越界为0，负下标为0，但能赋值。

6.2 如何判断动态申请越界（C++ 方式，注意源程序后缀为.cpp）

VS2022 的 x86/Debug，Release 模式

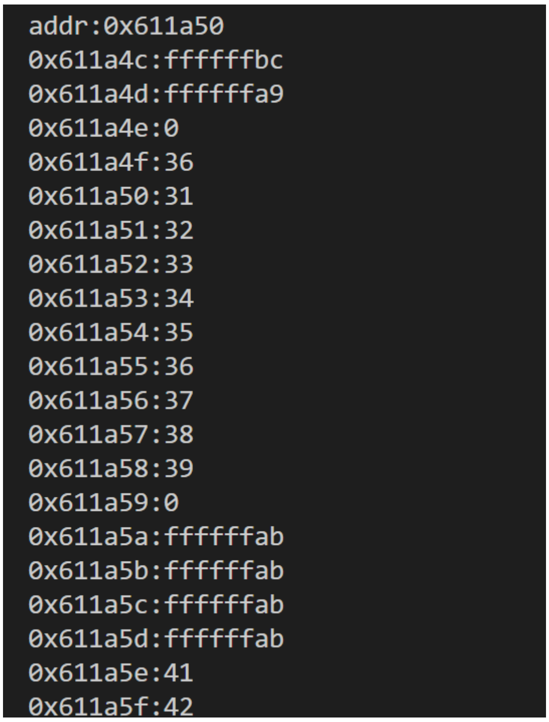
结果1：

Release



```
addr:0x6e1a50
0x6e1a4c:1
0x6e1a4f:36
0x6e1a50:31
0x6e1a51:32
0x6e1a52:33
0x6e1a53:34
0x6e1a54:35
0x6e1a55:36
0x6e1a56:37
0x6e1a57:38
0x6e1a58:39
0x6e1a59:0
0x6e1a5a:ffffffab
0x6e1a5b:ffffffab
0x6e1a5c:ffffffab
0x6e1a5d:ffffffab
0x6e1a5e:41
0x6e1a5f:42
```

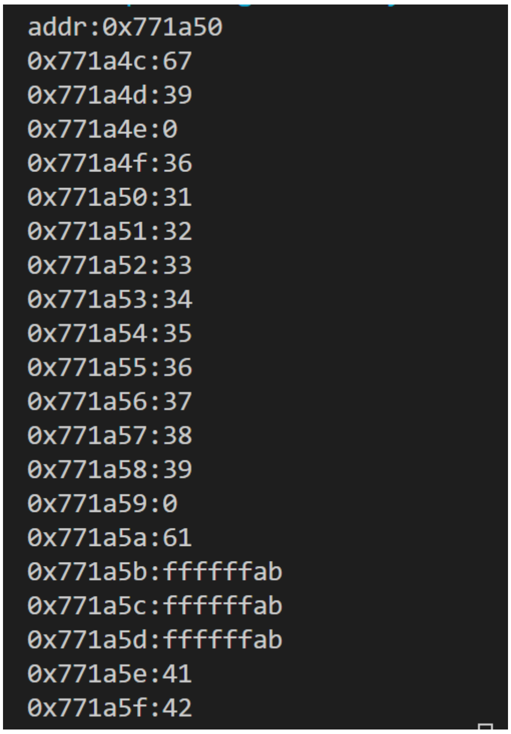
Debug



```
addr:0x611a50
0x611a4c:ffffffbc
0x611a4d:ffffffa9
0x611a4e:0
0x611a4f:36
0x611a50:31
0x611a51:32
0x611a52:33
0x611a53:34
0x611a54:35
0x611a55:36
0x611a56:37
0x611a57:38
0x611a58:39
0x611a59:0
0x611a5a:ffffffab
0x611a5b:ffffffab
0x611a5c:ffffffab
0x611a5d:ffffffab
0x611a5e:41
0x611a5f:42
```

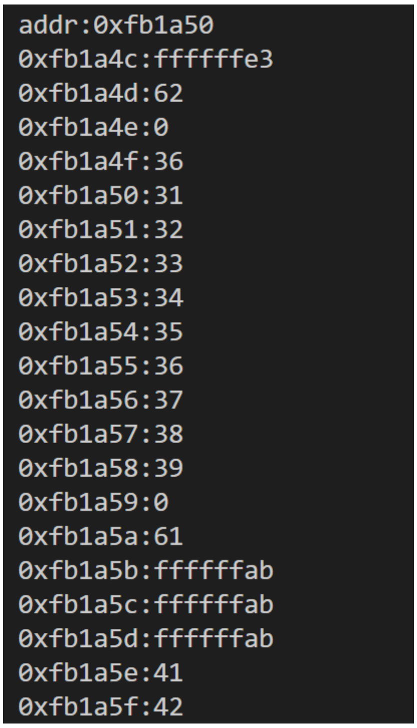
结果2

Release



```
addr:0x771a50
0x771a4c:67
0x771a4d:39
0x771a4e:0
0x771a4f:36
0x771a50:31
0x771a51:32
0x771a52:33
0x771a53:34
0x771a54:35
0x771a55:36
0x771a56:37
0x771a57:38
0x771a58:39
0x771a59:0
0x771a5a:61
0x771a5b:ffffffab
0x771a5c:ffffffab
0x771a5d:ffffffab
0x771a5e:41
0x771a5f:42
```

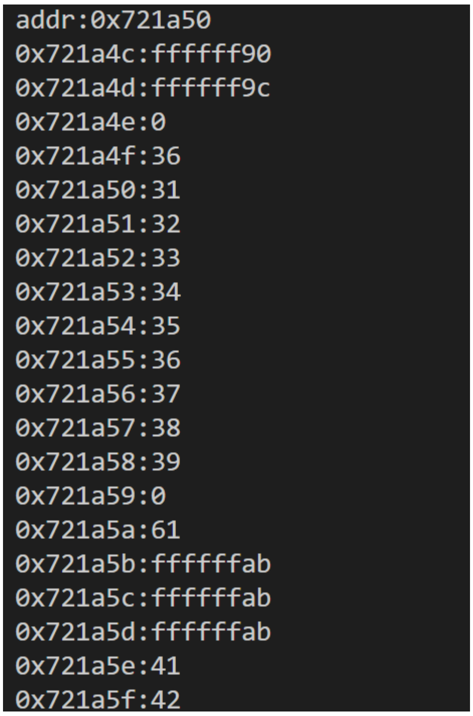
Debug



```
addr:0xfb1a50
0xfb1a4c:ffffffe3
0xfb1a4d:62
0xfb1a4e:0
0xfb1a4f:36
0xfb1a50:31
0xfb1a51:32
0xfb1a52:33
0xfb1a53:34
0xfb1a54:35
0xfb1a55:36
0xfb1a56:37
0xfb1a57:38
0xfb1a58:39
0xfb1a59:0
0xfb1a5a:61
0xfb1a5b:ffffffab
0xfb1a5c:ffffffab
0xfb1a5d:ffffffab
0xfb1a5e:41
0xfb1a5f:42
```

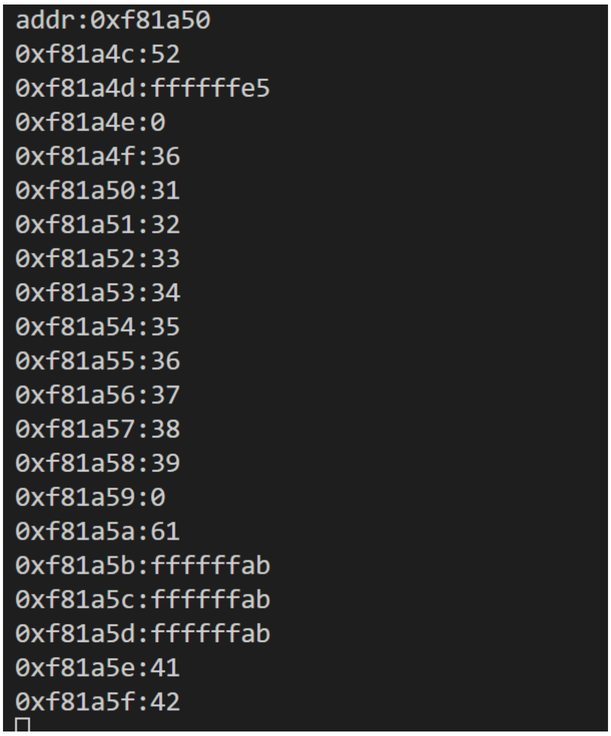
结果3

Release



```
addr:0x721a50
0x721a4c:ffffff90
0x721a4d:ffffff9c
0x721a4e:0
0x721a4f:36
0x721a50:31
0x721a51:32
0x721a52:33
0x721a53:34
0x721a54:35
0x721a55:36
0x721a56:37
0x721a57:38
0x721a58:39
0x721a59:0
0x721a5a:61
0x721a5b:ffffffab
0x721a5c:ffffffab
0x721a5d:ffffffab
0x721a5e:41
0x721a5f:42
```

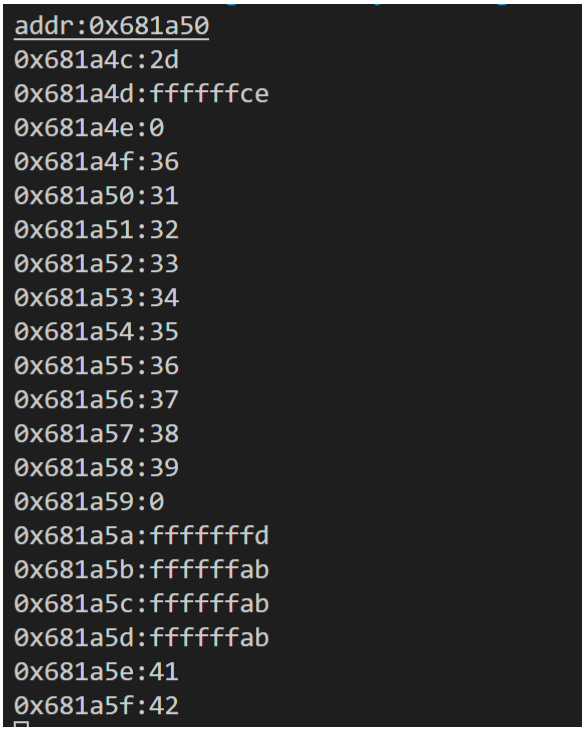
Debug



```
addr:0xf81a50
0xf81a4c:52
0xf81a4d:ffffffe5
0xf81a4e:0
0xf81a4f:36
0xf81a50:31
0xf81a51:32
0xf81a52:33
0xf81a53:34
0xf81a54:35
0xf81a55:36
0xf81a56:37
0xf81a57:38
0xf81a58:39
0xf81a59:0
0xf81a5a:61
0xf81a5b:ffffffab
0xf81a5c:ffffffab
0xf81a5d:ffffffab
0xf81a5e:41
0xf81a5f:42
```

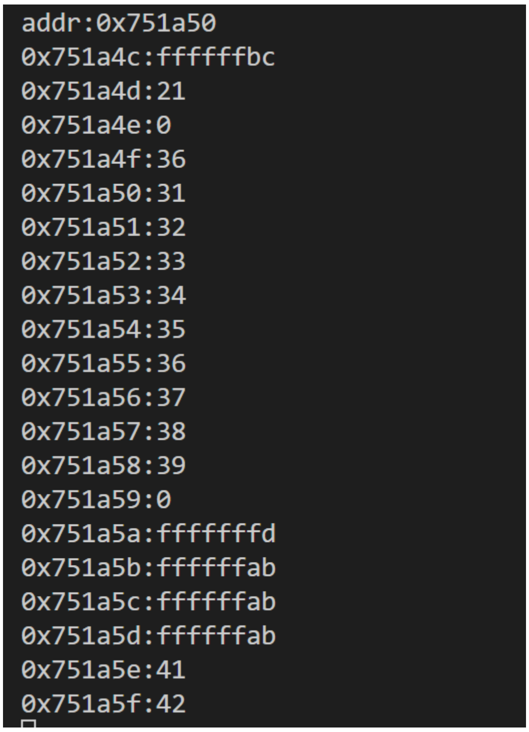
结果4

Release



```
addr:0x681a50
0x681a4c:2d
0x681a4d:ffffffce
0x681a4e:0
0x681a4f:36
0x681a50:31
0x681a51:32
0x681a52:33
0x681a53:34
0x681a54:35
0x681a55:36
0x681a56:37
0x681a57:38
0x681a58:39
0x681a59:0
0x681a5a:fffffffd
0x681a5b:ffffffab
0x681a5c:ffffffab
0x681a5d:ffffffab
0x681a5e:41
0x681a5f:42
```

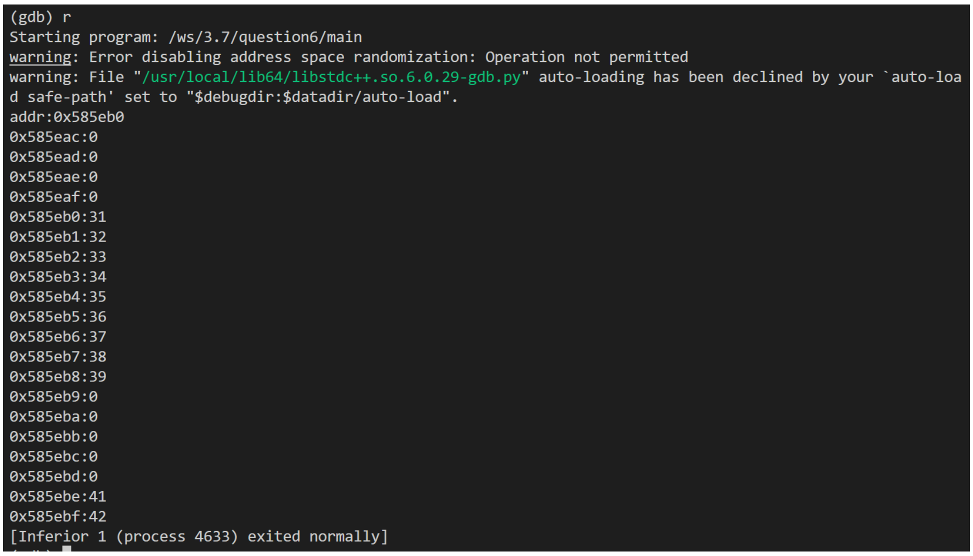
Debug



```
addr:0x751a50
0x751a4c:ffffffbc
0x751a4d:21
0x751a4e:0
0x751a4f:36
0x751a50:31
0x751a51:32
0x751a52:33
0x751a53:34
0x751a54:35
0x751a55:36
0x751a56:37
0x751a57:38
0x751a58:39
0x751a59:0
0x751a5a:fffffffd
0x751a5b:ffffffab
0x751a5c:ffffffab
0x751a5d:ffffffab
0x751a5e:41
0x751a5f:42
```

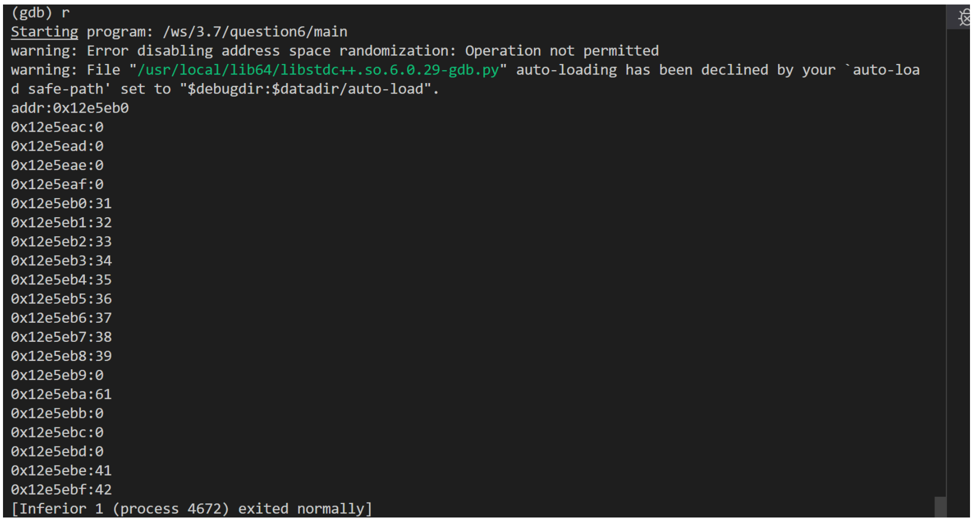
linux下GDB：

结果1



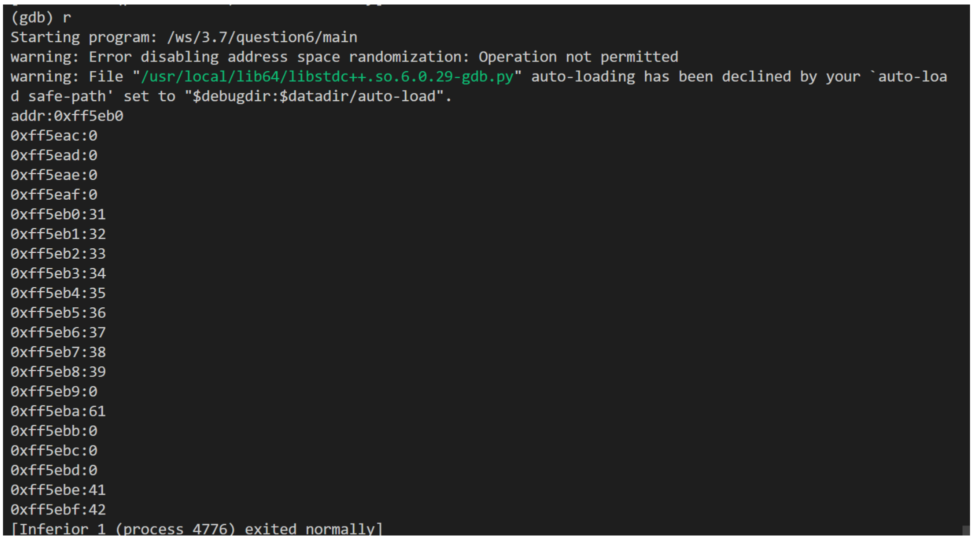
```
(gdb) r
Starting program: /ws/3.7/question6/main
warning: Error disabling address space randomization: Operation not permitted
warning: File "/usr/local/lib64/libstdc++.so.6.0.29-gdb.py" auto-loading has been declined by your `auto-loa
d safe-path' set to "$debugdir:$datadir/auto-load".
addr:0x585eb0
0x585eac:0
0x585ead:0
0x585eae:0
0x585eaf:0
0x585eb0:31
0x585eb1:32
0x585eb2:33
0x585eb3:34
0x585eb4:35
0x585eb5:36
0x585eb6:37
0x585eb7:38
0x585eb8:39
0x585eb9:0
0x585eba:0
0x585ebb:0
0x585ebc:0
0x585ebd:0
0x585ebe:41
0x585ebf:42
[Inferior 1 (process 4633) exited normally]
```

结果2



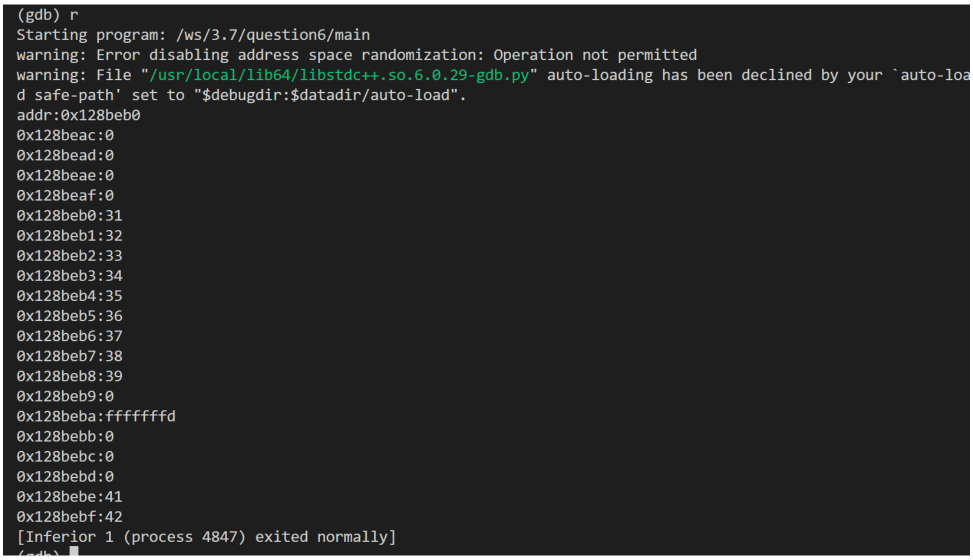
```
(gdb) r
Starting program: /ws/3.7/question6/main
warning: Error disabling address space randomization: Operation not permitted
warning: File "/usr/local/lib64/libstdc++.so.6.0.29-gdb.py" auto-loading has been declined by your `auto-loa
d safe-path' set to "$debugdir:$datadir/auto-load".
addr:0x12e5eb0
0x12e5eac:0
0x12e5ead:0
0x12e5eae:0
0x12e5eaf:0
0x12e5eb0:31
0x12e5eb1:32
0x12e5eb2:33
0x12e5eb3:34
0x12e5eb4:35
0x12e5eb5:36
0x12e5eb6:37
0x12e5eb7:38
0x12e5eb8:39
0x12e5eb9:0
0x12e5eba:61
0x12e5ebb:0
0x12e5ebc:0
0x12e5ebd:0
0x12e5ebe:41
0x12e5ebf:42
[Inferior 1 (process 4672) exited normally]
```

结果3



```
(gdb) r
Starting program: /ws/3.7/question6/main
warning: Error disabling address space randomization: Operation not permitted
warning: File "/usr/local/lib64/libstdc++.so.6.0.29-gdb.py" auto-loading has been declined by your `auto-loa
d safe-path' set to "$debugdir:$datadir/auto-load".
addr:0xff5eb0
0xff5eac:0
0xff5ead:0
0xff5eae:0
0xff5eaf:0
0xff5eb0:31
0xff5eb1:32
0xff5eb2:33
0xff5eb3:34
0xff5eb4:35
0xff5eb5:36
0xff5eb6:37
0xff5eb7:38
0xff5eb8:39
0xff5eb9:0
0xff5eba:61
0xff5ebb:0
0xff5ebc:0
0xff5ebd:0
0xff5ebe:41
0xff5ebf:42
[Inferior 1 (process 4776) exited normally]
```

结果4



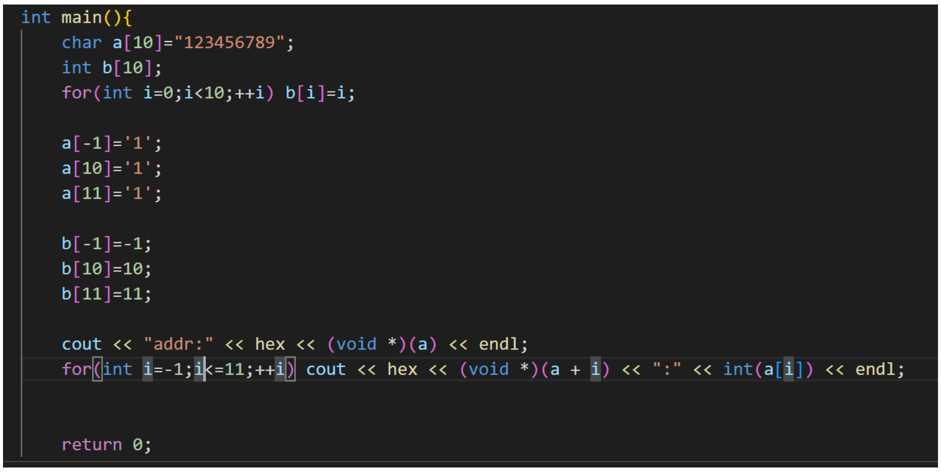
```
(gdb) r
Starting program: /ws/3.7/question6/main
warning: Error disabling address space randomization: Operation not permitted
warning: File "/usr/local/lib64/libstdc++.so.6.0.29-gdb.py" auto-loading has been declined by your `auto-loa
d safe-path' set to "$debugdir:$datadir/auto-load".
addr:0x128beb0
0x128beac:0
0x128bead:0
0x128beae:0
0x128beaf:0
0x128beb0:31
0x128beb1:32
0x128beb2:33
0x128beb3:34
0x128beb4:35
0x128beb5:36
0x128beb6:37
0x128beb7:38
0x128beb8:39
0x128beb9:0
0x128beba:fffffffd
0x128bebb:0
0x128bebc:0
0x128bebd:0
0x128bebe:41
0x128bebf:42
[Inferior 1 (process 4847) exited normally]
(gdb)
```

总结：结果似乎和c差不多，在windows下，数组越界的位置值为ffffffxx来判断越界，但能给越界的位置赋值，负下标的值随机(规律很诡异，-1为字符‘6’，-2为空)，Linux GDB会报warning，越界为0，但能赋值，负下标为0。

6.3　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）

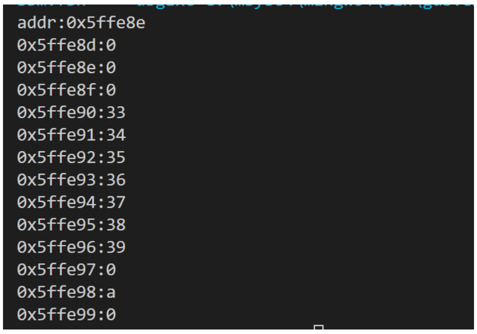
对于字符数组：

main函数



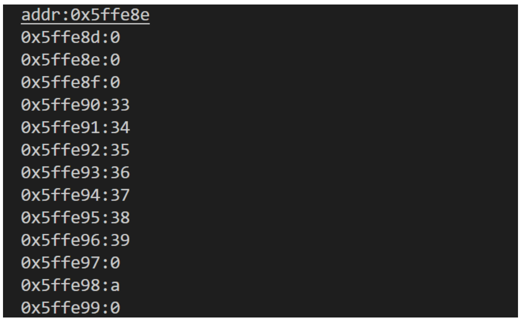
```
int main(){
    char a[10]="123456789";
    int b[10];
    for(int i=0;i<10;++i) b[i]=i;

    a[-1]='1';
    a[10]='1';
    a[11]='1';

    b[-1]=-1;
    b[10]=10;
    b[11]=11;

    cout << "addr:" << hex << (void *)(a) << endl;
    for(int i=-1;i<=11;++i) cout << hex << (void *)(a + i) << ":" << int(a[i]) << endl;


    return 0;
```

Release:



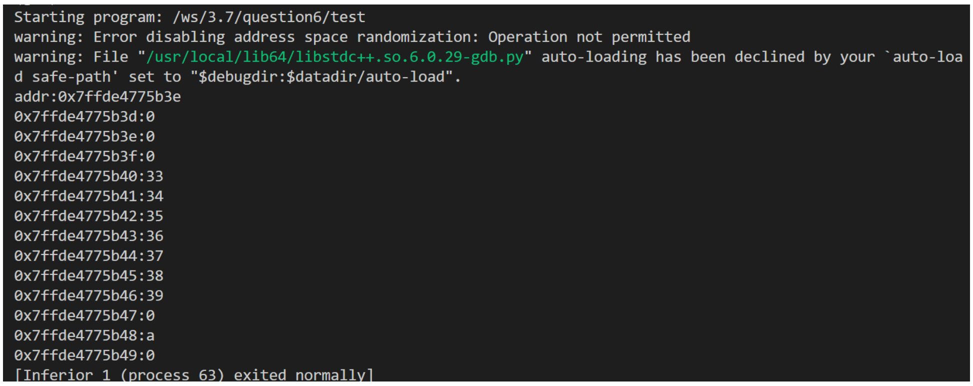
```
addr:0x5ffe8e
0x5ffe8d:0
0x5ffe8e:0
0x5ffe8f:0
0x5ffe90:33
0x5ffe91:34
0x5ffe92:35
0x5ffe93:36
0x5ffe94:37
0x5ffe95:38
0x5ffe96:39
0x5ffe97:0
0x5ffe98:a
0x5ffe99:0
```

Debug:



```
addr:0x5ffe8e
0x5ffe8d:0
0x5ffe8e:0
0x5ffe8f:0
0x5ffe90:33
0x5ffe91:34
0x5ffe92:35
0x5ffe93:36
0x5ffe94:37
0x5ffe95:38
0x5ffe96:39
0x5ffe97:0
0x5ffe98:a
0x5ffe99:0
```

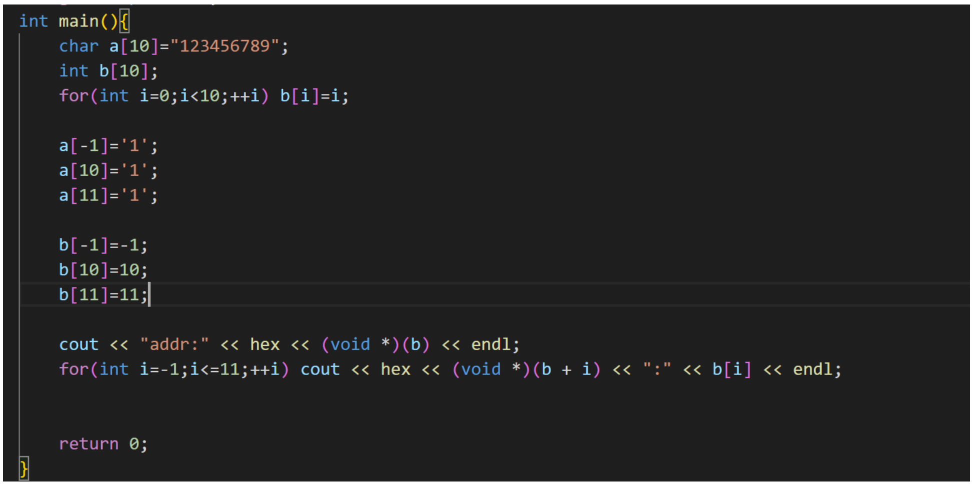
GDB:



```
Starting program: /ws/3.7/question6/test
warning: Error disabling address space randomization: Operation not permitted
warning: File "/usr/local/lib64/libstdc++.so.6.0.29-gdb.py" auto-loading has been declined by your `auto-loa
d safe-path' set to "$debugdir:$datadir/auto-load".
addr:0x7ffde4775b3e
0x7ffde4775b3d:0
0x7ffde4775b3e:0
0x7ffde4775b3f:0
0x7ffde4775b40:33
0x7ffde4775b41:34
0x7ffde4775b42:35
0x7ffde4775b43:36
0x7ffde4775b44:37
0x7ffde4775b45:38
0x7ffde4775b46:39
0x7ffde4775b47:0
0x7ffde4775b48:a
0x7ffde4775b49:0
[Inferior 1 (process 63) exited normally]
```

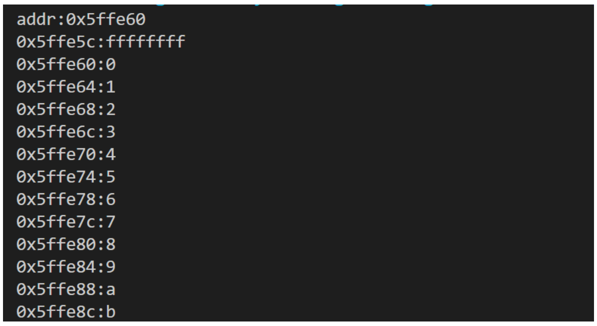
对于整形数组：

main函数



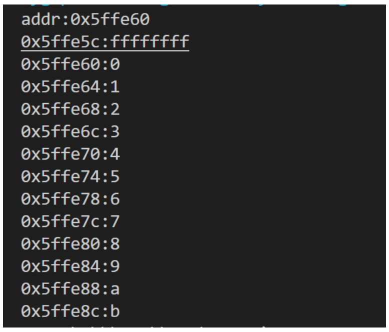
```cpp
int main(){
    char a[10]="123456789";
    int b[10];
    for(int i=0;i<10;++i) b[i]=i;

    a[-1]='1';
    a[10]='1';
    a[11]='1';

    b[-1]=-1;
    b[10]=10;
    b[11]=11;

    cout << "addr:" << hex << (void *)(b) << endl;
    for(int i=-1;i<=11;++i) cout << hex << (void *)(b + i) << ":" << b[i] << endl;


    return 0;
}
```

Release：



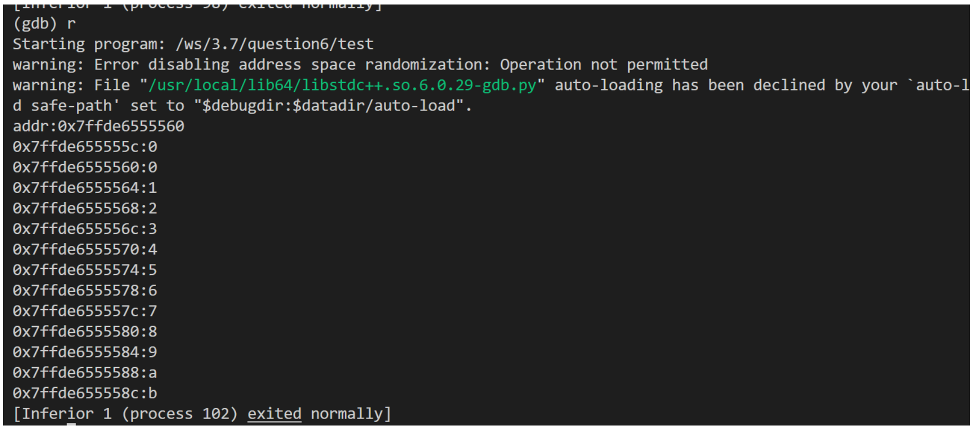
```
addr:0x5ffe60
0x5ffe5c:ffffffff
0x5ffe60:0
0x5ffe64:1
0x5ffe68:2
0x5ffe6c:3
0x5ffe70:4
0x5ffe74:5
0x5ffe78:6
0x5ffe7c:7
0x5ffe80:8
0x5ffe84:9
0x5ffe88:a
0x5ffe8c:b
```

Debug：



```
addr:0x5ffe60
0x5ffe5c:ffffffff
0x5ffe60:0
0x5ffe64:1
0x5ffe68:2
0x5ffe6c:3
0x5ffe70:4
0x5ffe74:5
0x5ffe78:6
0x5ffe7c:7
0x5ffe80:8
0x5ffe84:9
0x5ffe88:a
0x5ffe8c:b
```

GDB：



```
[Inferior 1 (process 98) exited normally]
(gdb) r
Starting program: /ws/3.7/question6/test
warning: Error disabling address space randomization: Operation not permitted
warning: File "/usr/local/lib64/libstdc++.so.6.0.29-gdb.py" auto-loading has been declined by your `auto-l
d safe-path' set to "$debugdir:$datadir/auto-load".
addr:0x7ffde6555560
0x7ffde655555c:0
0x7ffde6555560:0
0x7ffde6555564:1
0x7ffde6555568:2
0x7ffde655556c:3
0x7ffde6555570:4
0x7ffde6555574:5
0x7ffde6555578:6
0x7ffde655557c:7
0x7ffde6555580:8
0x7ffde6555584:9
0x7ffde6555588:a
0x7ffde655558c:b
[Inferior 1 (process 102) exited normally]
```

总结：字符串数组在三种模式下结果相同，负下标为0，超出范围有一个a作为越界判断。数组在windows底下相同，且能给负下标以及超出范围的位置赋值，在linux底下无法给负下标赋值，也能给超出范围赋值。

具体代码见：question6

6.4 总结

在不同编译环境下，数组越界位置的值不太相同，且在动态申请空间和静态申请空间的时候也不太相同。Debug和Release模式大致相同，linux下的GDB与windows底下差别非常大。感觉在使用静态申请空间时，数组越界的影响会稍微小一点，在动态申请空间时影响巨大。在使用数组时得尽可能清晰地了解自己使用了多少空间，避免越界。