

现在的逆向C++题越来越多，经常上来就是一堆容器、标准模板库，这个系列主要记录这些方面的逆向学习心得

本文主要介绍`std::vector`，因为逆向题中的C++代码可能会故意写的很绕，比如输入一个数组，直接给vector赋值即可，但是也可以用稍微费解的方法■■■`push_back`

vector

内存布局

仍然用vs调试，观察内存布局

```
1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  int main(int argc, char** argv) {
6      std::vector<int> a;
7      int num[16];
8      for (int i = 0; i < 100; i++) {
9          a.push_back(i); 已用时间 <= 2ms
10         std::cout << "size : " << i + 1 << "\t" << "capacity : " << a.capacity()
11     }
12     system("pause");
13     return 0;
14 }
```

自动窗口		
搜索(Ctrl+E) 搜索深度: 3		
名称	值	类型
a	{ size=2 }	std::vector<int,st...
[capacity]	2	__int64
[allocator]	allocator	std::_Compresse...
[0]	0	int
[1]	1	int
[原始视图]	{_Mypair=allocator }	std::vector<int,st...
i	2	int
num	0x00000084678ff6e0 {-858993460, -858993460, -8589...	int[16]

vector a的第一个字段是size ■■第二个字段是capacity ■■

和`std::string`差不多

当`size>capacity`也就是空间不够用时

首先配置一块新空间，然后将元素从旧空间——搬往新空间，再把旧空间归还给操作系统

内存增长机制

测试代码：

```
#include<iostream>
#include<vector>
using namespace std;
```


具体调试一下，当push_back(0)和(1)时：

内存 1

地址: 0x00000229D968FD10

0x00000229D968FD10	00 00 00 00 01 00 00 00	fd fd fd fd dd dd dd dd dd dd dd dd dd dd dd dd dd dd dd dd dd dd c3 b0 c8 38 dd
0x00000229D968FD2D	08 00 80 dd	
0x00000229D968FD4A	dd dd	
0x00000229D968FD67	dd dd	

main.cpp

cpp (全局范围) main(int argc, char ** argv)

```
3 using namespace std;
4
5 int main(int argc, char** argv) {
6     std::vector<int> a;
7     for (int i = 0; i < 100; i++) {
8         a.push_back(i); 已用时间 <= 2ms
9         std::cout << "size : " << i + 1 << "\t" << "capacity : " << a.capacity() << std::endl;
10    }
```

98 % 未找到相关问题

自动窗口

搜索(Ctrl+E) 搜索深度: 3

名称	值	类型
	67 'C'	char
a	{ size=2 }	std::vector<int,st...
[capacity]	2	__int64
[allocator]	allocator	std::_Compresse...
[原始视图]	{_Myval2={_Myfirst=0x00000229d968fd10 {0} _Mylast...	std::_Compresse...
[0]	0	int
[1]	1	int
[原始视图]	{_Mypair=allocator }	std::vector<int,st...

输出

显示输出来源(S): 调

"cpp.exe"(Win32):
"cpp.exe"(Win32):
"cpp.exe"(Win32):
"cpp.exe"(Win32):
"cpp.exe"(Win32):
"cpp.exe"(Win32):
线程 0x40c 已退出,

注意一开始的内存窗口，每次动态扩容时确实已经改变了存储空间地址

再F5执行到断点，内存窗口的■■■说明这块内存刚动过，已经被操作系统回收了，vector中的元素也已经改变了存放地址

```
for(vector<int>::const_iterator iter=ivec.begin();iter!=ivec.end();++iter){
    /*...*/
}
```

```
}
```

IDA视角

IDA中打开，因为是windows下vs编译的，看不出vector和accumulate和lambda的特征了

```
Memory = (_DWORD *)operator_new(0x14ui64);
v4 = (unsigned __int64)(Memory + 5);
*(_OWORD *)Memory = 0i64;
Memory[4] = 0;
memory = Memory;
v6 = 5i64;
v7 = 5i64;
do
{
    std::basic_istream<char,std::char_traits<char>>::operator>>(std::cin, memory);
    ++memory;
    --v7;
}
while ( v7 );
v8 = 0;
if ( (unsigned __int64)Memory > v4 )
    v6 = 0i64;
if ( (unsigned __int64)Memory <= v4 )
{
    v9 = 0i64;
    v10 = Memory;
    do
    {
        v8 += *v10;
        ++v10;
        ++v9;
    }
    while ( v9 != v6 );
}
std::basic_ostream<char,std::char_traits<char>>::operator<<(std::cout);
if ( Memory )
    operator_delete(Memory);
return 0;
```

分析一下，开了一块内存0x14字节，也就是对应我们的5个int

依次输入赋值，最后用一个指针++遍历这个地址

获得累加和并输出

transform

换个稍复杂的std::transform的例子，保留特征，用g++编译

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<numeric>

using namespace std;

int main(int argc, char** argv) {
    std::vector<int> a = { 1,2,3,4,5};
    std::vector<int> b(5);
    std::vector<int> result;
```

```

for (int i = 0; i < 5; i++) { std::cin >> b[i]; }
std::transform(a.begin(), a.end(), b.begin(), std::back_inserter(result),
    [](int _1, int _2) { return _1 * _2; });

for (int i = 0; i < 5; i++) {
    if (result[i] != 2 * (i + 1)) {
        std::cout << "You failed!" << std::endl;
        exit(0);
    }
}
std::cout << "You win!" << std::endl;
return 0;
}
//g++ main.cpp -o test -std=c++14

```

用std::transform同时对两个列表进行操作，输入5个数存入vector b中，然后vector result分别是a[i]*b[i]，最后判断result中的每个数是否符合要求

注意，vector b大小一定要超过vector a，从参数中也可以看出来，b只传入了begin()

如果vector b较小，后面的内存存放的是未知的数据

会造成未定义行为 UB

IDA视角

IDA打开可以看到vector相关代码，但是命名很乱，根据std::transform二元操作符的特征我们可以更改一下变量名

```

v20 = __readfsqword(0x28u);
std::allocator<int>::allocator(&result, argv, envp);
std::vector<int,std::allocator<int>>::vector(&nums, &five_nums, 5LL, &result);
std::allocator<int>::~~allocator(&result);
std::allocator<int>::allocator(&result, &five_nums, v4);// 代码中的vector{1,2,3,4,5}
std::vector<int,std::allocator<int>>::vector(&input_vector, 5LL, &result);
std::allocator<int>::~~allocator(&result);
std::vector<int,std::allocator<int>>::vector(&result);
for ( i = 0; i <= 4; ++i )
{
    input_num = std::vector<int,std::allocator<int>>::operator[](&input_vector, i);// vector result[i]
    std::istream::operator>>(&std::cin, input_num);// 输入的5个数字
}
result_back_inserter = std::back_inserter<std::vector<int,std::allocator<int>>>(&result);
input_vector_begin = std::vector<int,std::allocator<int>>::begin(&input_vector);// input begin()
nums_end = std::vector<int,std::allocator<int>>::end(&nums);// iterator end()
nums_begin = std::vector<int,std::allocator<int>>::begin(&nums);// iterator begin()
std::transform<__gnu_cxx::__normal_iterator<int *,std::vector<int,std::allocator<int>>>,__gnu_cxx::__normal_iterator<int *,std::vector<
    nums_begin,
    nums_end,
    input_vector_begin,
    result_back_inserter,
    v10,
    v11,
    v3>;

```

我们定义的vector{1,2,3,4,5}在内存中如下

.rodata:0000000000402620	five_nums	db	1
.rodata:0000000000402621		db	0
.rodata:0000000000402622		db	0
.rodata:0000000000402623		db	0
.rodata:0000000000402624		db	2
.rodata:0000000000402625		db	0
.rodata:0000000000402626		db	0
.rodata:0000000000402627		db	0
.rodata:0000000000402628		db	3
.rodata:0000000000402629		db	0
.rodata:000000000040262A		db	0
.rodata:000000000040262B		db	0
.rodata:000000000040262C		db	4
.rodata:000000000040262D		db	0
.rodata:000000000040262E		db	0
.rodata:000000000040262F		db	0
.rodata:0000000000402630		db	5
.rodata:0000000000402631		db	0
.rodata:0000000000402632		db	0
.rodata:0000000000402633		db	0

跟进std::transform

```
while ( (unsigned __int8)__gnu_cxx::operator!=(int *,std::vector<int,std::allocator<int>>>(&v14, &v13) )
{
    v7 = *(__DWORD *)__gnu_cxx::__normal_iterator<int *,std::vector<int,std::allocator<int>>>::operator*(&v12);
    v8 = (int *)__gnu_cxx::__normal_iterator<int *,std::vector<int,std::allocator<int>>>::operator*(&v14);
    v15 = main::{lambda(int,int)#1}::operator() const((__int64)&a7, *v8, v7);// 关键的lambda
    v9 = std::back_insert_iterator<std::vector<int,std::allocator<int>>>::operator*(&v11);
    std::back_insert_iterator<std::vector<int,std::allocator<int>>>::operator=(v9, &v15);
    __gnu_cxx::__normal_iterator<int *,std::vector<int,std::allocator<int>>>::operator++(&v14);
    __gnu_cxx::__normal_iterator<int *,std::vector<int,std::allocator<int>>>::operator++(&v12);
    std::back_insert_iterator<std::vector<int,std::allocator<int>>>::operator++(&v11);
}
```

一眼注意到最关键的lambda , 其他都是operator* = ++等重载的迭代器相关的操作符

熟悉transform的话显然没有需要我们关注的东西

```
__int64 __fastcall main::{lambda(int,int)#1}::operator() const(__int64 a1, int a2, int a3)
{
    return (unsigned int)(a3 * a2);
}
```

lambda中也只是我们实现的简单乘法运算

```

for ( idx = 0; idx <= 4; ++idx )
{
    if ( *(_DWORD *)std::vector<int,std::allocator<int>>::operator[](&result, idx) != 2 * (idx + 1) )
    {
        v12 = std::operator<<<std::char_traits<char>>(&std::cout, "You failed!");
        std::ostream::operator<<<(v12, &std::endl<char,std::char_traits<char>>);
        exit(0);
    }
}

```

算法很简单，只要输入5个2就会得到win了

vector存vector

这个程序写的有点...没事找事，用于再深入分析一下

比如输入10个数，分别放入size为1 2 3 4的四个vector，并且把4个vector一起放在一个vector中,再进行运算
虽然正常程序不会这么写，但是作为逆向的混淆感觉效果不错

```

#include<iostream>
#include<vector>
#include<algorithm>
#include<numeric>

using namespace std;

int main(int argc, char** argv) {
    std::vector<std::vector<int>>> a;
    a.push_back(std::vector<int>{1, 2, 3});
    a.push_back(std::vector<int>{6, 7});
    for (auto v : a) {
        for (auto n : v) {
            std::cout << n << "\t";
        }
        std::cout << std::endl;
    }
    return 0;
}
//g++ main.cpp -std=c++14 -o test

```

内存结构

为了方便说明，仍然在vs下观察内存结构

▲ a	{ size=2 }	std::vector<std::v...
[capacity]	2	__int64
▶ [allocator]	allocator	std::_Compresse...
▲ [0]	{ size=3 }	std::vector<int,st...
[capacity]	3	__int64
▶ [allocator]	allocator	std::_Compresse...
[0]	1	int
[1]	2	int
[2]	3	int
▶ [原始视图]	{_Mypair=allocator }	std::vector<int,st...

一开始纠结了很久，因为vector开的内存必定是连续的，也就是说{1,2,3}是连续的，{6,7}也是连续的

那么外层vector如果把{1,2,3},{6,7}存在一起，那么当内层vector扩容时，一定会影响到外层vector

最后才明白，外层vector只是存了内层vector的数据结构，而不是直接存了{1,2,3},{6,7}

IDA视角

IDA打开g++编译过后的程序，便于学习演示


```

v15 = __readfsqword(0x28u);
std::vector<std::vector<int,std::allocator<int>>,std::allocator<std::vector<int,std::allocator<int>>>>>::vector(
    &vector_vector,
    argv,
    envp);
std::allocator<int>::~allocator(&vec1);
std::vector<int,std::allocator<int>>::vector(&vec2, &_96, 3LL, &vec1);// vector{1,2,3}
std::vector<std::vector<int,std::allocator<int>>,std::allocator<std::vector<int,std::allocator<int>>>>>::push_back(
    &vector_vector,
    &vec2);
std::vector<int,std::allocator<int>>::~~vector(&vec2);// 析构函数
std::allocator<int>::~~allocator(&vec1);
std::allocator<int>::~allocator(&vec2);
std::vector<int,std::allocator<int>>::vector(&vec1, &_97, 2LL, &vec2);// vector{6,7}
std::vector<std::vector<int,std::allocator<int>>,std::allocator<std::vector<int,std::allocator<int>>>>>::push_back(
    &vector_vector,
    &vec1);
std::vector<int,std::allocator<int>>::~~vector(&vec1);// 析构函数
std::allocator<int>::~~allocator(&vec2);
v10 = &vector_vector;
vector_vector_begin = std::vector<std::vector<int,std::allocator<int>>,std::allocator<std::vector<int,std::allocator<int>>>>>::begin(&vector_vector);
vector_vector_end = std::vector<std::vector<int,std::allocator<int>>,std::allocator<std::vector<int,std::allocator<int>>>>>::end(&vector_vector);

```

结合注释和变量的重命名，逻辑比较清晰

```
vector_vector<vector<int> >.push_back(&vec1)
```

可以理解为外层vector存了内层vector的"指针"

输出部分：

```

while ( (unsigned __int8) __gnu_cxx::operator!=(std::vector<int,std::allocator<int>> *,std::vector<std::vector<int,std::allocator<int>>,std::allocator<std::vector<int,std::allocator<int>>>>::vector(
    &vec_vec_iter,
    &vec_vec_end) ) )
{
    v3 = __gnu_cxx::__normal_iterator<std::vector<int,std::allocator<int>> *,std::vector<std::vector<int,std::allocator<int>>,std::allocator<std::vector<int,std::allocator<int>>>>::vector(
    &vec2, v3);
    vec2_addr = &vec2;
    vec2_iter = std::vector<int,std::allocator<int>>::begin(&vec2);
    vec1 = std::vector<int,std::allocator<int>>::end(vec2_addr);
    while ( (unsigned __int8) __gnu_cxx::operator!=(int *,std::vector<int,std::allocator<int>>>(&vec2_iter, &vec1) ) )
    {
        v4 = (unsigned int *) __gnu_cxx::__normal_iterator<int *,std::vector<int,std::allocator<int>>>::operator*(&vec2_iter);
        v5 = std::ostream::operator<<(&data, *v4);
        std::operator<<std::char_traits<char>>(v5, &unk_402CB9);
        __gnu_cxx::__normal_iterator<int *,std::vector<int,std::allocator<int>>>::operator++(&vec2_iter);
    }
    std::ostream::operator<<(&data, &std::endl<char,std::char_traits<char>>);
    std::vector<int,std::allocator<int>>::~~vector(&vec2);
    __gnu_cxx::__normal_iterator<std::vector<int,std::allocator<int>> *,std::vector<std::vector<int,std::allocator<int>>,std::allocator<std::vector<int,std::allocator<int>>>>::~vector(&vec_vec);
    return 0;
}

```

稍微有些不理解，看起来两个内层vector的迭代器之间有一些优化

vec1 = end(vec2_addr)，这一句没怎么看懂，因为上传附件经常丢失...没有上传例程，通过源码编译比较简单，大佬们有兴趣可以试着逆一下逻辑

不过主线还是清晰的

- 外层vector的迭代器operator ++和operator !=
- 双层循环，内层循环分别得到每个内层vector的*iterator，通过ostream输出

小总结

vector中连续内存里存的是类型的数据结构，比如int的数据结构，vector<int>的数据结构

但无论如何，每个vector用于存数据的内存都是连续的

比如 {1,2,3},vector<int>{1,2},vector<int>{3,4,5}这两个vector

点击收藏 | 2 关注 | 3

[上一篇：要想加入红队，需要做好哪些准备？](#) [下一篇：内核漏洞挖掘技术系列\(3\)——bo...](#)

1. 1 条回复



[新user](#) 2019-04-29 09:05:45

收藏加一 解析很详细 学习了 感谢

0 回复Ta

[登录](#) 后跟帖

[先知社区](#)

[现在登录](#)

[热门节点](#)

[技术文章](#)

[社区小黑板](#)

[目录](#)

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)