正常程序中会包含许多的复杂数据结构，就内存布局而言，数组可以算是最简单的复合数据结构来。传统意义上的数组指包含同一数据类型的连续元素的连续内存块。用数组中元素的数量乘以每个元素的大小，即可直接计算出数组的大小。

访问数组元素的方式可以是整数常量偏移，也可能是索引*数组元素大小的动态计算的方式。

全局分配的数组

如果一个数组在程序的全局数据区内分配（例如，在．data或．bss节），编译器在编译时可获知该数组的基址。由于基址固定，编译器可以计算出使用固定索引访问的任何数组元素的固定地址。 例如：

```
int global_array[3];

int main(){
    int index = 2;
    global_array[0] = 10;
    global_array[1] = 20;
    global_array[2] = 30;
    global_array[index] = 40;
}
```

上面这段代码的反汇编代码可能如下所示（部分展示）：

```
mov dword_40B720,10
mov dword_40B724,20
mov dword_40B728,30

mov eax,[ebp+idx]
mov dword_40B720[eax*4],40
```

尽管这个程序只有一个全局变量，但前三行反汇编行似乎表明，它使用了3个全局变量。倒数两行处对偏移量的计算（eax×4）是暗示全局数组dword_40B720存在的唯一线索，不过，数组的名称与第一行的全局变量的名称相同。

可以看到IDA在知道数组从0040B720开始，因此使用了固定索引(0,1,2)来计算数组中对应元素的具体地址（0040B720、0040B724、0040B728），此时可以通过上一章对数组的格式化操作、将dword 40B720转换成一个三元素数组。

因此，使用常量索引访问全局数组时，在对应的反汇编代码清单中，对应的数组元素将以全局变量的形式出现。

其次，使用可变索引值将带领我们来到数组的开头，因为在计算要访问的数组元素的具体地址时，我们需要用数组的基址加上相应的偏移量，这时基址即呈现出来，并且通过观察与索引相乘的数可以知道数组中元素的大小。

栈中分配的数组

如果数组是作为栈变量分配的，似乎编译器在编译时无法获得绝对地址，而且即使是使用常量索引的访问也必须在运行时进行某种计算。但实际上，编译器几乎以完全相同的方式处理栈分配的数组和全局分配的数组。

对于栈中数组的常量索引的数组访问，编译器将通过类似ebp+var_x的类似局部变量的访问方式进行访问数组，对于可变索引值的访问，将采取类似[ebp+eax*4+var_base]的方式进行访问，可以发现除了基址的表示不一样，其他均类似。

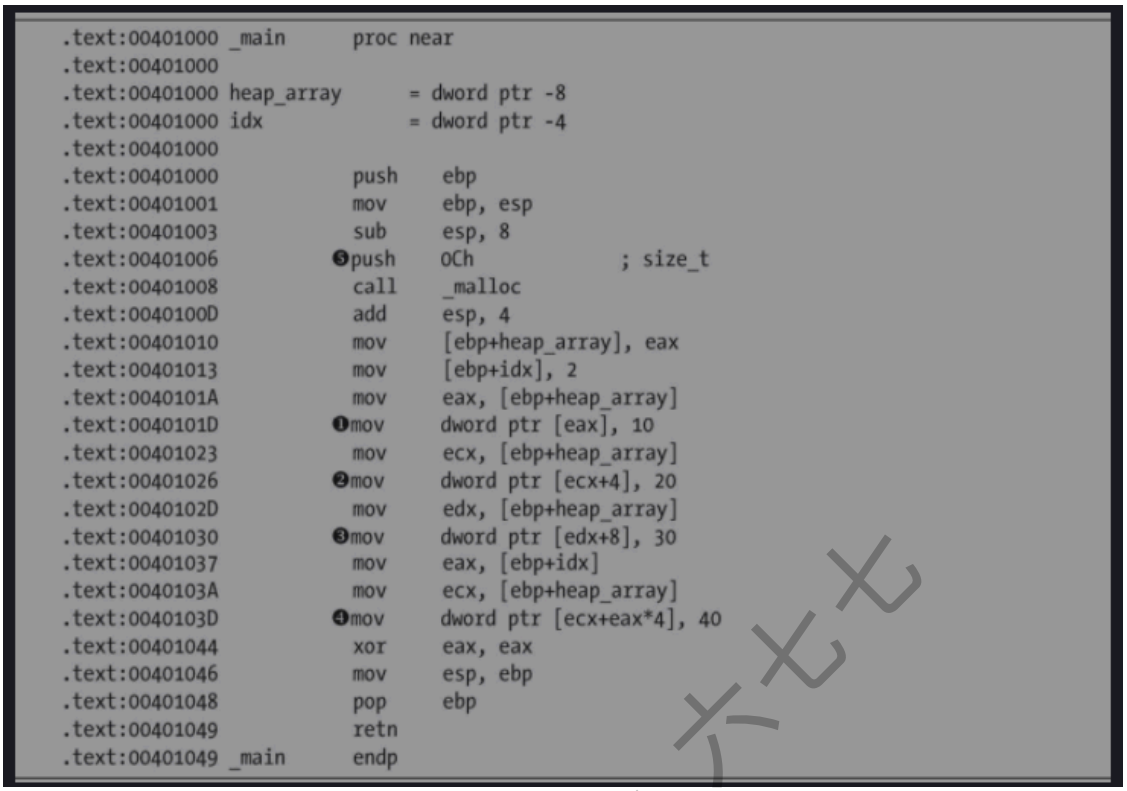
堆中分配的数组

堆分配的数组是使用一个动态内存分配函数（如C中的malloc或C++中的new）分配的。从编译器的角度讲，处理堆分配的数组的主要区别在于，它必须根据内存分配函数返回的地址值，生成对数组的所有引用。

```
int main(){
    int heap_array = (int*)malloc(3 * sizeof(int));
    int index = 2;
    heap_array[0] = 10;
    heap_array[1] = 20;
    heap_array[2] = 30;
    heap_array[index] = 40;
}
```

对于堆中数组的访问，与前两就有不同之处了。

不同之处在于，虽然都是需要通过一个基地址和偏移量算出数组中对应元素的地址，只不过堆分配需要从保存了malloc返回的地址的局部变量中读取。



```
.text:00401000 _main      proc near
.text:00401000
.text:00401000 heap_array      = dword ptr -8
.text:00401000 idx             = dword ptr -4
.text:00401000
.text:00401000           push    ebp
.text:00401001           mov     ebp, esp
.text:00401003           sub     esp, 8
.text:00401006         ❺push     0Ch             ; size_t
.text:00401008           call    _malloc
.text:0040100D           add     esp, 4
.text:00401010           mov     [ebp+heap_array], eax
.text:00401013           mov     [ebp+idx], 2
.text:0040101A           mov     eax, [ebp+heap_array]
.text:0040101D         ❶mov      dword ptr [eax], 10
.text:00401023           mov     ecx, [ebp+heap_array]
.text:00401026         ❷mov      dword ptr [ecx+4], 20
.text:0040102D           mov     edx, [ebp+heap_array]
.text:00401030         ❸mov      dword ptr [edx+8], 30
.text:00401037           mov     eax, [ebp+idx]
.text:0040103A           mov     ecx, [ebp+heap_array]
.text:0040103D         ❹mov      dword ptr [ecx+eax*4], 40
.text:00401044           xor     eax, eax
.text:00401046           mov     esp, ebp
.text:00401048           pop     ebp
.text:00401049           retn
.text:00401049 _main      endp
```

堆分配的数组有一个非常有用的特点。如果能够确定数组的总大小和每个元素的大小，我们可以轻松计算出该数组所包含的元素的数量。对堆分配的数组而言，传递给内存分配函数的参数（0x0C在❺处传递给了malloc）即表示分配给数组的字节总数，用这个数除以元素大小即可得到数组中元素的个数。

数组的结论

关于数组的使用，我们能够得出的唯一确定的结论是：只有当变量被用作数组的索引时，我们才最容易确定数组的存在。要访问数组中的元素，首先需要用索引乘以数组元素的大小，计算出相应元素的偏移量，然后将得到的偏移量与数组的基址相加，得到数组元素的访问地址。

但是在使用常量索引值访问数组元素时，它们很少能够证明数组的存在，并且看起来与用于访问结构体成员的代码非常类似。

结构体成员访问

C结构体，这里通称为结构体，是异类数据集合，可将数据类型各不相同的项组合到一个复合数据类型中。结构体的一个显著特点在于，结构体中的数据字段是通过名称访问，而不是像数组那样通过索引访问。不好的是，字段名称被编译器转换成了

数字偏移量。

所以，在反汇编代码清单中，访问结构体字段的方式看起来与使用常量索引访问数组元素的方式极其相似。

如果编译器遇到一个结构体定义，它会计算出结构体中字段所耗用字节的累计值，以确定结构体中每个字段的偏移量，除此之外，编译器还会进行内存对齐来提高读写效率。

这里又涉及到一个额外的知识点，结构体的内存大小计算方式：

- 各成员变量存放的起始地址相对于结构的起始地址的偏移量必须为该变量的类型所占用的字节数的倍数；
- 各成员变量在存放的时候根据在结构中出现的顺序依次申请空间 同时按照上面的对齐方式调整位置空缺的字节自动填充；
- 同时为了确保结构的大小为结构的字节边界数(即该结构中占用最大的空间的类型的字节数)的倍数，所以在为最后一个成员变量申请空间后还会根据需要自动填充空缺的字节。

全局分配的结构体

和全局分配的数组一样，编译器在编译时可获知全局分配的结构体的地址。这使得编译器能够在编译时计算出结构体中每个成员的地址，而不必在运行时进行任何计算。

对于全局分配的结构体，IDA中所显示的与全局数组一样，看起来访问的是全局变量。

栈分配的结构体

同样，访问结构体中的字段不需要进行任何算术计算，因为在编译时，编译器能够确定栈帧内每个字段的相对偏移量。在这种情况下，我们同样会被误导，认为程序使用的是5个变量，而不是一个碰巧包含5个字段的变量。

堆分配的结构体

如果一个结构体在程序堆中分配、那么、在访问其中的字段时、编译器别无选择、

只有生成代码来计算每个字段在结构体中的正确偏移量。这是结构体的地址在编译时未知所导致的后果。

对于全局分配的结构体，编译器能够计算出一个固定的起始地址。对于栈分配的结构体，编译器能够计算出结构体起始位置与相关栈帧的帧指针之间的固定关系。

对于堆分配的数组，通过malloc申请的内存大小，可以推断结构体的大小，通过指针的偏移量可以推算出结构体中成员的大致布局。