作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

数组的本质

数组的本质是一块连续的内存，包含多个类型相同、大小相同的元素。
数组的大小，等于元素的大小乘以元素的个数。
数组的元素，可以是整数、浮点数、字符串，也可以是 struct、union、数组等。
多维数组，本质是一维数组。

理解数组的关键，在于内存布局。读写数组，使用首地址+偏移。可以使用下标，也可以使用指针。
数组非常高效。从汇编看出，数组的各个部分与内存一一对应，没有多余的部分。

数组没有值传递。可以用手动复制数组的每个元素，实现值传递。
数组引用传递，只传递数组变量的地址。

用 C 和汇编分析数字型数组

编写代码： num_array.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// 32 位整数的数组
int32_t int32_arr[3] = {100, 200};

// 64 位整数的数组
int64_t int64_arr[3] = {555L, 666L};

int main()
{
    // 大小
    printf("int32_arr  size = %d \n", sizeof(int32_arr));
    printf("int64_arr  size = %d \n", sizeof(int64_arr));

    // 写属性
    int32_arr[0] = 101;
    int32_arr[1] = 202;
    int32_arr[2] = 303;
    int64_arr[0] = 555000L;
    int64_arr[1] = 666000L;
    int64_arr[2] = 777000L;

    // 读属性
```

```
    int32_t int32_tmp = int32_arr[2];
    int64_t int64_tmp = int64_arr[2];
    printf("int32_arr[2]    value = %d \n", int32_tmp);
    printf("int64_arr[2]    value = %lld \n", int64_tmp);

    // 读写使用指针。
    int32_t *int32_ptr = int32_arr;
    int64_t *int64_ptr = int64_arr;
    *(int32_ptr + 1) = 222;
    *(int64_ptr + 1) = 666111L;
    int32_t int32_tmp2 = *(int32_ptr + 1);
    int64_t int64_tmp2 = *(int64_ptr + 1);
    printf("*(int32_ptr + 1)    value = %d \n", int32_tmp2);
    printf("*(int64_ptr + 1)    value = %lld \n", int64_tmp2);

    return 0;
}
```

编译代码：

```
gcc num_array.c -o num_array
gcc num_array.c -S -o num_array.s
readelf -a num_array > num_array.elf.txt
```

运行代码：

```
[root@192 array]# ./num_array
int32_arr  size = 12
int64_arr  size = 24
int32_arr[2]    value = 303
int64_arr[2]    value = 777000
*(int32_ptr + 1)    value = 222
*(int64_ptr + 1)    value = 666111
```

分析结果：
int32_arr 有 3 个元素，元素大小为 4 字节，数组大小为 12 字节。
int64_arr 有 3 个元素，元素大小为 8 字节，数组大小为 24 字节。

查看符号。查看文件 num_array.elf.txt，找到符号 int32_arr、int64_arr。

```
Symbol table '.symtab' contains 65 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    51: 0000000000601050    24 OBJECT  GLOBAL DEFAULT   24 int64_arr
    62: 0000000000601040    12 OBJECT  GLOBAL DEFAULT   24 int32_arr
```

符号 int64_arr，地址为 0000000000601050，大小为 24。
符号 int32_arr，地址为 0000000000601040，大小为 12。

数组的定义和初始化。查看文件 num_array.c、num_array.s，这里截取部分代码。

操作	源码	汇编代码	分析

定义和初始化	`int32_t int32_arr[3] = {100, 200};`	`int32_arr:`
		`.long 100`
		`.long 200`
			数组有 3 个元素。
			指定了 2 个元素。第 3 个元素默认值为 0。

		.zero 4	.long 表示 4 字节整数。 .zero 4 表示 4 个 bit 为 0 。
定义和初始化	int64_t int64_arr[3] = {555L, 666L};	int64_arr: .quad 555 .quad 666 .zero 8	数组有 3 个元素。 指定了 2 个元素。第 3 个元素默认值为 0。 .quad 表示 8 字节整数。 .zero 8 表示 8 个 bit 为 0 。

用下标读写。查看文件 num_array.c、num_array.s，这里截取部分代码。

操作	源码	汇编代码	分析

写属性	int32_arr[0] = 101;
	int32_arr[1] = 202;
	int32_arr[2] = 303;
		movl $101, int32_arr(%rip)
		movl $202, int32_arr+4(%rip)
		movl $303, int32_arr+8(%rip)
			使用首地址+偏移。
			每个元素 4 个字节。
			3 个元素的偏移为 0、4、8。

写属性	int64_arr[0] = 555000L;
	int64_arr[1] = 666000L;
	int64_arr[2] = 777000L;
		movq $555000, int64_arr(%rip)
		movq $666000, int64_arr+8(%rip)
		movq $777000, int64_arr+16(%rip)
			使用首地址+偏移。
			每个元素 8 个字节。
			3 个元素的偏移为 0、8、16。

读属性	int32_t int32_tmp = int32_arr[2];	movl int32_arr+8(%rip), %eax movl %eax, -4(%rbp)	使用首地址+偏移。 每个元素 4 个字节。 int32_arr[2]的偏移为 8。
读属性	int64_t int64_tmp = int64_arr[2];	movq int64_arr+16(%rip), %rax movq %rax, -16(%rbp)	使用首地址+偏移。 每个元素 8 个字节。 int64_arr[2]的偏移为 16。

用指针读写。查看文件 num_array.c、num_array.s，这里截取部分代码。

操作	源码	汇编代码	分析
定义指针	int32_t *int32_ptr = int32_arr; int64_t *int64_ptr = int64_arr;	movq $int32_arr, -24(%rbp) movq $int64_arr, -32(%rbp)	读出变量的地址，写到栈上。

写属性	*(int32_ptr + 1) = 222;	movq -24(%rbp), %rax
		addq $4, %rax
		movl $222, (%rax)
			把变量地址读到 rax。
			把 rax 加 4。每个元素 4 个字节。
			把数字 222 写到 rax 指向的内存。

写属性	*(int64_ptr + 1) = 666111L;	movq -32(%rbp), %rax
		addq $8, %rax
		movq $666111, (%rax)
			把变量地址读到 rax。
			把 rax 加 8。每个元素 8 个字节。
			把数字 666111 写到 rax 指向的内存。

读属性	int32_t int32_tmp2 = *(int32_ptr + 1);	movq -24(%rbp), %rax
		movl 4(%rax), %eax
		movl %eax, -36(%rbp)
			把变量地址读到 rax。
			把 rax+4 指向的值，写到 eax。
			把 eax 写到栈上。

读属性	int64_t int64_tmp2 = *(int64_ptr + 1);	movq -32(%rbp), %rax
		movq 8(%rax), %rax
		movq %rax, -48(%rbp)
			把变量地址读到 rax。
			把 rax+8 指向的值，写到 rax。
			把 rax 写到栈上。

用 C 和汇编分析 struct 型数组

数组使用偏移定位元素。struct 使用偏移定位属性。struct 型数组把 2 种偏移累加，形成最终的偏移，用来定位 struct 的属性。

编写代码： struct_array.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

struct Desk
{
    int32_t length; // 4 个字节
    int16_t width;  // 2 个字节
};

// 数组
struct Desk desk_arr[3] = {{.length = 5000, .width = 600}};

int main()
{
    printf("struct size = %d \n", sizeof(struct Desk));
    printf("array  size = %d \n", sizeof(desk_arr));

    // 写属性
    desk_arr[2].length = 1000;
    desk_arr[2].width = 200;

    // 读属性
    int32_t length = desk_arr[2].length;
    int16_t width = desk_arr[2].width;
    printf("desk_arr[2].length = %d \n", length);
    printf("desk_arr[2].width  = %d \n", width);

    return 0;
}
```

编译代码：

```
gcc struct_array.c -o struct_array
gcc struct_array.c -S -o struct_array.s
readelf -a struct_array > struct_array.elf.txt
```

运行代码：

```
[root@192 array]# ./struct_array
struct size = 8
array  size = 24
```

```
desk_arr[2].length = 1000
desk_arr[2].width  = 200
```

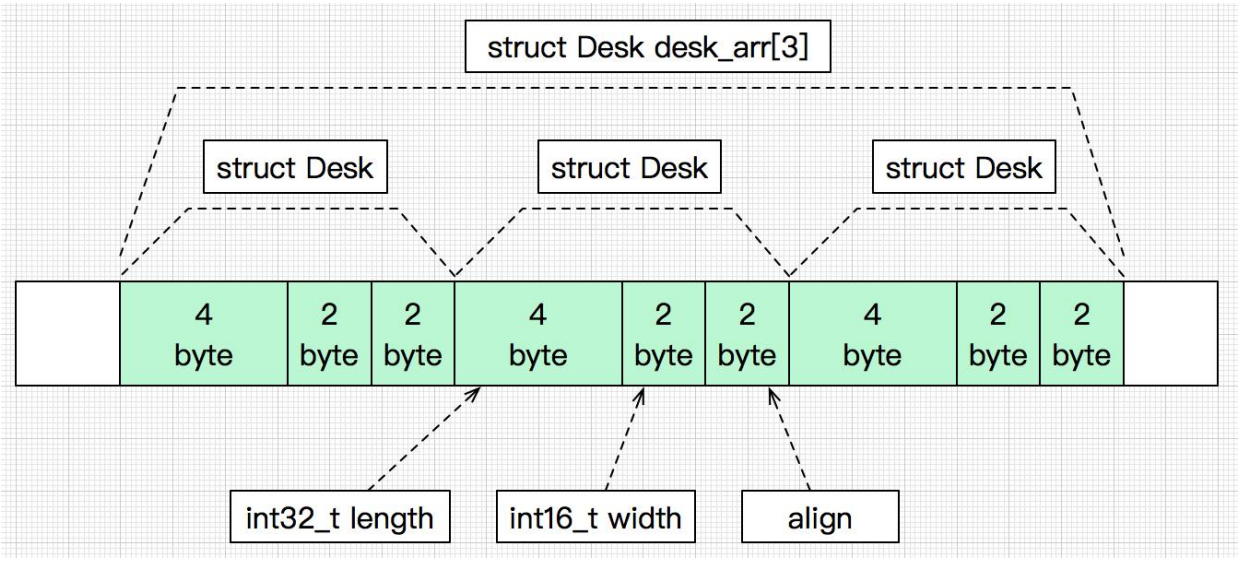
分析结果：
struct Desk 的大小为 8 字节。对齐填充了 2 个字节。
数组 desk_arr，有 3 个元素，总大小为 24 字节。

查看符号。查看文件 struct_array.elf.txt，找到符号 desk_arr。

```
Symbol table '.symtab' contains 64 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    47: 0000000000601040    24 OBJECT  GLOBAL DEFAULT   24 desk_arr
```

符号 desk_arr，地址为 0000000000601040，大小为 24。

内存布局。3 个 struct 依次排列在内存。每个 struct 的多个属性依次排列在内存。



数组的定义和初始化。查看文件 struct_array.c、struct_array.s，这里截取部分代码。

操作	源码	汇编代码	分析

定义和初始化	`struct Desk desk_arr[3] = {{.length = 5000, .width = 600}};`	`desk_arr:`
		`  .long  5000`
		`  .value  600`
		`  .zero  2`
		`  .zero  16`
			数组有 3 个元素。
			指定了 1 个元素。后面 2 个元素默认值为 0。
			.long 表示 4 字节整数。
			.value 表示 2 字节整数。
			.zero 2 表示 struct 填充 2 个 bit。
			.zero 16 表示后面 2 个 struct 的 16 个 bit 为 0。

用下标读写。查看文件 struct_array.c、struct_array.s，这里截取部分代码。

操作	源码	汇编代码	分析
写属性	`desk_arr[2].length = 1000;` `desk_arr[2].width = 200;`	`movl  $1000, desk_arr+16(%rip)` `movw  $200, desk_arr+20(%rip)`	使用首地址+偏移。 每个元素占用 8 个字节。 第 3 个元素的偏移为 16。 struct 的 2 个属性的偏移依次为 0、4。 desk_arr[2].length 的偏移为 16+0=16。

			desk_arr[2].width 的偏移为 16+4=20。
读属性	`int32_t length = desk_arr[2].length;`	`movl desk_arr+16(%rip), %eax` `movl %eax, -4(%rbp)`	使用首地址+偏移。 desk_arr[2].length 的偏移为 16+0=16。
读属性	`int16_t width = desk_arr[2].width;`	`movzwl desk_arr+20(%rip), %eax` `movw %ax, -6(%rbp)`	使用首地址+偏移。 desk_arr[2].width 的偏移为 16+4=20。

用 C 和汇编分析值传递与引用传递

获取数组变量的地址，直接用数组名称，不需要使用&前缀。和一般的变量取地址有区别。

编写代码： refer_array.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// 原始数组
int32_t num_array[2] = {10000, 20000};

// 参数为名称
void add_by_name(int32_t num_array[2])
{
    num_array[0] += 5;
    num_array[1] += 5;
}

// 参数为指针
void add_by_ptr(int32_t *ptr)
{
    *(ptr + 0) += 800;
    *(ptr + 1) += 800;
}

void print_array(char *title, int32_t *ptr)
{
    printf("\n %s \n", title);
    printf(" arr[0] = %d \n", *(ptr + 0));
    printf(" arr[1] = %d \n", *(ptr + 1));
}

int main()
{
```

```
    add_by_name(num_array);
    print_array("add_by_name :", num_array);

    add_by_ptr(num_array);
    print_array("add_by_ptr :", num_array);

    return 0;
}
```

编译代码：

```
gcc refer_array.c -o refer_array
gcc refer_array.c -S -o refer_array.s
```

运行代码：

```
[root@192 array]# ./refer_array

 add_by_name :
 arr[0] = 10005
 arr[1] = 20005

 add_by_ptr :
 arr[0] = 10805
 arr[1] = 20805
```

分析结果：
2 个函数，都使数组的内容发生变化。
函数 `void add_by_ptr(int32_t *ptr)` ，入参使用引用传递。
函数 `void add_by_name(int32_t num_array[2])` ，入参从定义看是值传递，其实是引用传递。

分析函数。查看文件 refer_array.c、refer_array.s，这里截取部分代码。

操作	源码	汇编代码	分析
函数定义	`void add_by_name(int32_t num_array[2])`	`add_by_name:` `movq %rdi, -8(%rbp)`	传递入参使用 rdi。
函数定义	`void add_by_ptr(int32_t *ptr)`	`add_by_ptr:` `movq %rdi, -8(%rbp)`	传递入参使用 rdi。
函数调用	`add_by_name(num_array);`	`movl $num_array, %edi` `call add_by_name`	把 num_array 的地址,写到 edi，作为入参。
函数调用	`add_by_ptr(num_array);`	`movl $num_array, %edi` `call add_by_ptr`	把 num_array 的地址,写到 edi，作为入参。

从汇编代码看出，2 个方法，源码定义不同，但是传递入参的方式相同。都使用 rdi 传递 num_array 的地址，都是引用传递。

用 C 和汇编分析数组越界访问

数组占用一块内存。数组访问使用首地址+偏移。如果数组后面的临近内存可以正常访问，则使用首地址+偏移可以越界访问。

编写代码： border_array.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// 数组
int32_t array[2] = {0xA1A2A3A4, 0xB1B2B3B4};

// 16 位整数
int16_t bb16 = 0xC1C2;
int16_t cc16 = 0xD1D2;

// 32 位整数
int32_t dd32 = 0xE1E2E3E4;

// 查看变量的信息
void print_param(char *name, uint64_t addr, int size)
{
    printf("%8s    addr = %#X  %llu    size = %d \n", name, addr, addr, size);
}

void print_int32(int32_t num)
{
    printf("  int32 = %13d  %#13X \n", num, num);
}

int main()
{
    print_param("array", (uint64_t)array, sizeof(array));
    print_param("bb16", (uint64_t)&bb16, sizeof(bb16));
    print_param("cc16", (uint64_t)&cc16, sizeof(cc16));
    print_param("dd32", (uint64_t)&dd32, sizeof(dd32));

    // 读数组的元素
    int32_t num0 = array[0];
    int32_t num1 = array[1];

    // 读数组之外的元素
    int32_t num2 = array[2];
    int32_t num3 = array[3];
```

```
    printf("\n");
    print_int32(num0);
    print_int32(num1);
    print_int32(num2);
    print_int32(num3);
    return 0;
}
```

编译代码：

```
gcc border_array.c -o border_array
gcc border_array.c -S -o border_array.s
```

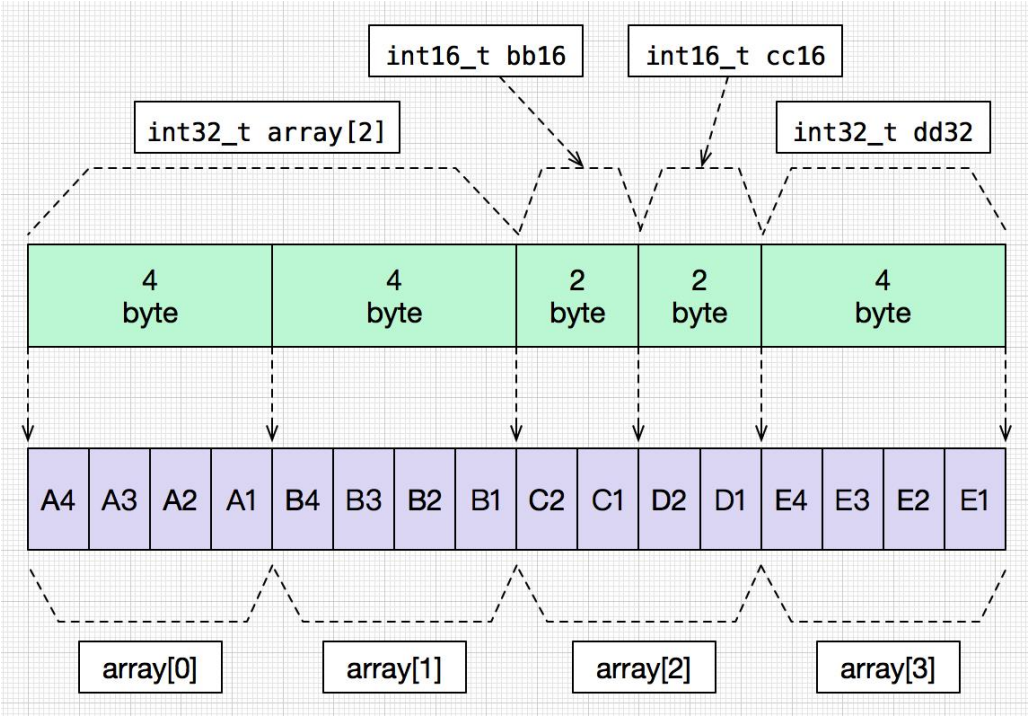
运行代码：

```
[root@192 array]# ./border_array
   array    addr = 0X60103C  6295612    size = 8
    bb16    addr = 0X601044  6295620    size = 2
    cc16    addr = 0X601046  6295622    size = 2
    dd32    addr = 0X601048  6295624    size = 4

  int32 =   -1583176796     0XA1A2A3A4
  int32 =   -1313688652     0XB1B2B3B4
  int32 =    -774716990     0XD1D2C1C2
  int32 =    -505224220     0XE1E2E3E4
```

分析结果：
4 个变量，在内存中依次排列，地址分别为 6295612、6295620、6295622、6295624。
使用数组的下标，可以访问数组之外的变量。

内存布局。字节序为小端模式。数组的元素以 4 个字节为单位。
array[0]，4 个字节为 A4 A3 A2 A1 ，表示整数为 0XA1A2A3A4。
array[1]，4 个字节为 B4 B3 B2 B1 ，表示整数为 0XB1B2B3B4。
array[2]，4 个字节为 C2 C1 D2 D1 ，表示整数为 0XD1D2C1C2。
array[3]，4 个字节为 E4 E3 E2 E1 ，表示整数为 0XE1E2E3E4。



查看文件 border_array.c、border_array.s，这里截取部分代码。

操作	源码	汇编代码	分析
数组	int32_t array[2] = {0xA1A2A3A4, 0xB1B2B3B4};	array: .long -1583176796 .long -1313688652	数组有 2 个元素。 .long 表示 4 字节整数。
16 位整数	int16_t bb16 = 0xC1C2; int16_t cc16 = 0xD1D2;	bb16: .value -15934 cc16: .value -11822	.value 表示 2 字节整数。
32 位整数	int32_t dd32 = 0xE1E2E3E4;	dd32: .long -505224220	.long 表示 4 字节整数。

读数组的元素	int32_t num0 = array[0];
	int32_t num1 = array[1];
		movl array(%rip), %eax
		movl %eax, -4(%rbp)
		movl array+4(%rip), %eax
		movl %eax, -8(%rbp)
			使用首地址+偏移。
			数组的元素，每个占用 4 个字节。
			array[0]的偏移为 0。
			array[1]的偏移为 4。

读数组之外的元素	int32_t num2 = array[2];
	int32_t num3 = array[3];
		movl array+8(%rip), %eax
		movl %eax, -12(%rbp)
		movl array+12(%rip), %eax
		movl %eax, -16(%rbp)
			使用首地址+偏移。
			数组的元素，每个占用 4 个字节。
			array[2]的偏移为 8。
			array[3]的偏移为 12。

从汇编代码看出，数组越界访问依然使用首地址+偏移。

数组 array 在定义时有 2 个元素。array[2]、array[3] 越界访问，如果对应的地址有效，则访问正常。