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

struct 的本质

高级语言如 C++、java，有类 class 的概念。类是面向对象的重要特征，封装属性和方法。用类生成很多对象。每个对象占用一定的内存。
struct 可以表示 class，通过 struct 理解对象的内存布局。

struct 的本质是一块连续的内存，struct 包含多个属性，属性用偏移定位。
struct 的属性，可以是整数、浮点数、数组、字符串，也可以是函数、struct、union。
struct 非常高效。从汇编看出，struct 的各个部分与内存一一对应，没有多余的部分。
struct 值传递，把 struct 变量整体拷贝一份，作为新的变量。
struct 引用传递，只传递 struct 变量的地址。

理解 struct 的关键在于内存布局。各个属性，在内存依次排列，表现为线性结构。嵌套的 struct，其属性被展开，依然为线性结构。

用 C 和汇编分析 struct 属性的内存顺序和内存地址

struct 可以包含多个属性。多个属性的定义有先后顺序。struct 变量在内存中，多个属性的地址的顺序，和属性的定义顺序一致。
属性的读写，使用地址和偏移。struct 变量的首地址，和第一个属性的地址，相等。

编写代码： field_addr.c
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// 一个简单的 struct
typedef struct
{
    int32_t height; // 4 字节
    int32_t weight; // 4 字节
    int64_t speed;  // 8 字节
    int32_t notUse; // 4 字节
} Cat_t;

// 输出信息。
void print_cat(char *title, Cat_t *cat)
{
    printf("\n %s \n", title);
    printf("struct  addr = %p %llu \n", cat, (uint64_t)cat);
```

```
    printf("height  addr = %p %llu  value = %d \n", &cat->height, (uint64_t)&cat->height, cat->height);
    printf("weight  addr = %p %llu  value = %d \n", &cat->weight, (uint64_t)&cat->weight, cat->weight);
    printf("speed   addr = %p %llu  value = %lld \n", &cat->speed, (uint64_t)&cat->speed, cat->speed);
    printf("notUse  addr = %p %llu  value = %d \n", &cat->notUse, (uint64_t)&cat->notUse, cat->notUse);
}

// 数据区
Cat_t cat_red;

int main()
{
    // struct 的大小
    printf("Cat_t size = %d \n", sizeof(Cat_t));

    // 数据区
    cat_red.height = 111;
    cat_red.weight = 222;
    cat_red.speed = 333;
    cat_red.notUse = 0;

    // 堆区
    Cat_t *cat_blue = malloc(sizeof(Cat_t));
    cat_blue->height = 444;
    cat_blue->weight = 555;
    cat_blue->speed = 666;
    cat_blue->notUse = 0;

    // 栈区
    Cat_t cat_grey;
    cat_grey.height = 777;
    cat_grey.weight = 888;
    cat_grey.speed = 999;
    cat_grey.notUse = 0;

    print_cat("in Data area : ", &cat_red);
    print_cat("in Heap area : ", cat_blue);
    print_cat("in Stack area : ", &cat_grey);

    return 0;
}
```

编译代码：

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

运行代码：

```
[root@192 struct]# ./field_addr
Cat_t size = 24
```

```
 in Data area :
struct  addr = 0x601050 6295632
height  addr = 0x601050 6295632  value = 111
weight  addr = 0x601054 6295636  value = 222
speed   addr = 0x601058 6295640  value = 333
notUse  addr = 0x601060 6295648  value = 0

 in Heap area :
struct  addr = 0x1cde010 30269456
height  addr = 0x1cde010 30269456  value = 444
weight  addr = 0x1cde014 30269460  value = 555
speed   addr = 0x1cde018 30269464  value = 666
notUse  addr = 0x1cde020 30269472  value = 0

 in Stack area :
struct  addr = 0x7ffeddcccd70 140732619607408
height  addr = 0x7ffeddcccd70 140732619607408  value = 777
weight  addr = 0x7ffeddcccd74 140732619607412  value = 888
speed   addr = 0x7ffeddcccd78 140732619607416  value = 999
notUse  addr = 0x7ffeddcccd80 140732619607424  value = 0
```

分析结果：
Cat_t 包含 4 个属性，依次为 height 4 字节、weight 4 字节、speed 8 字节、notUse 4 字节。
在数据区、堆区、栈区，分别创建 1 个 Cat_t 变量。4 个属性的地址顺序，和定义的顺序一致。比如，堆区，4 个属性的地址依次为 30269456、30269460、30269464、30269472，地址为升序。Cat_t 变量的首地址，与第一个属性的地址，相同。比如，栈区，首地址为 140732619607408，第一个属性 height 的地址为 140732619607408，两者相同。

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

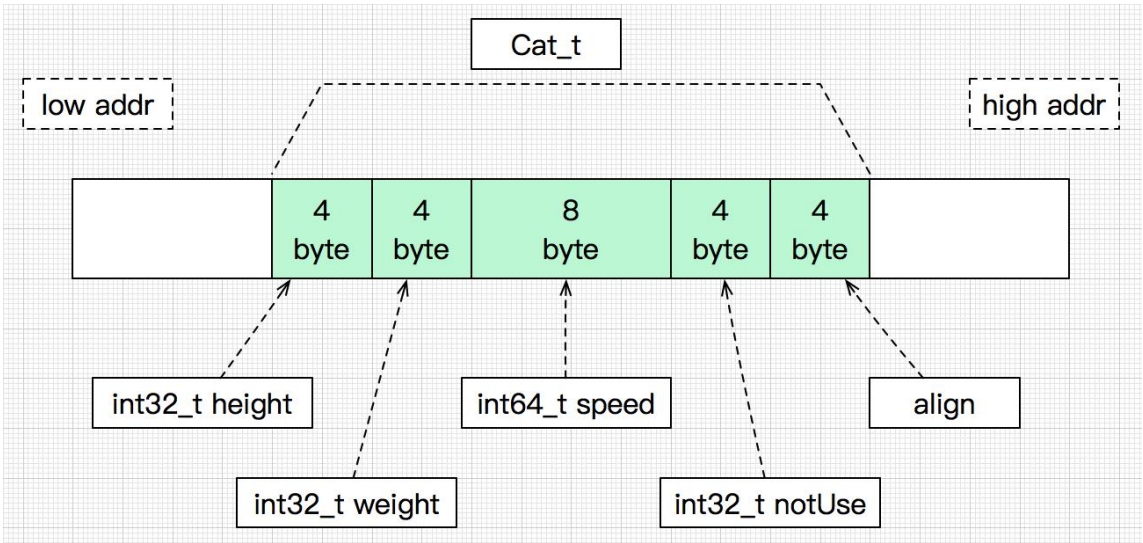
内存区域	源码	汇编代码	分析

数据区	cat_red.height = 111;
	cat_red.weight = 222;
	cat_red.speed = 333;
	cat_red.notUse = 0;
		movl $111, cat_red(%rip)
		movl $222, cat_red+4(%rip)
		movq $333, cat_red+8(%rip)
		movl $0, cat_red+16(%rip)
			使用 rip 相对寻址,依次为 4 个属性赋值。地址偏移依次为 0，4，8，16。

堆区	Cat_t *cat_blue = malloc(sizeof(Cat_t));
	cat_blue->height = 444;
	cat_blue->weight = 555;
	cat_blue->speed = 666;
	cat_blue->notUse = 0;
		call malloc
		movq %rax, -8(%rbp)
		movq -8(%rbp), %rax
		movl $444, (%rax)
		movq -8(%rbp), %rax
		movl $555, 4(%rax)
		movq -8(%rbp), %rax
		movq $666, 8(%rax)
		movq -8(%rbp), %rax
		movl $0, 16(%rax)
			调用 malloc 获得一个已分配内存的指针，返回值在 rax。
			使用 rax 加上偏移,依次为 4 个属性赋值。地址偏移依次为 0，4，8，16。

栈区	Cat_t cat_grey;
	cat_grey.height = 777;
	cat_grey.weight = 888;
	cat_grey.speed = 999;
	cat_grey.notUse = 0;
		movl $777, -32(%rbp)
		movl $888, -28(%rbp)
		movq $999, -24(%rbp)
		movl $0, -16(%rbp)
			使用 rbp 加上偏移，依次为 4 个属性赋值。第一个属性的地址为 rbp-32 ，地址最小。以第一个属性的地址为起点，地址偏移依次为 0，4，8，16。

Cat_t 大小为 24 个字节。地址对齐，在末尾填充了 4 个字节。
Cat_t 变量占用一块内存，这块内存可以在数据区、堆区、栈区等。



用 C 和汇编分析用地址遍历 struct 的属性

struct 变量占用一块内存。找到首地址、每个属性的偏移，就能用指针遍历全部属性。数组类型的属性，在内存中被展开。

编写代码： scan_field.c

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

typedef struct
{
    char name[8];     // 字符数组。8 字节
    float height;     // 浮点数。4 字节
    int32_t speed;    // 整数。4 字节
    int64_t score[2]; // 数组。16 字节
} Dog_t;

// 数据区
Dog_t dog;
```

```
int main()
{
    // 初始化
    sprintf(dog.name, "BoQiTa");
    dog.height = 22.33F;
    dog.speed = 555;
    dog.score[0] = 666;
    dog.score[1] = 777;

    // 变量地址
    void *ptr = &dog;
    printf("addr = %llu  struct  \n\n", (uint64_t)ptr);

    // 用偏移取属性
    char *name_ptr = ptr + 0;
    printf("addr = %llu  name = %s \n", (uint64_t)name_ptr, name_ptr);

    // 用偏移取属性
    float *height_ptr = ptr + 8;
    printf("addr = %llu  height = %f \n", (uint64_t)height_ptr, *height_ptr);

    // 用偏移取属性
    int32_t *speed_ptr = ptr + 12;
    printf("addr = %llu  speed = %d \n", (uint64_t)speed_ptr, *speed_ptr);

    // 用偏移取属性
    int64_t *score0 = ptr + 16;
    printf("addr = %llu  score[0] = %lld \n", (uint64_t)score0, *score0);

    // 用偏移取属性
    int64_t *score1 = ptr + 24;
    printf("addr = %llu  score[1] = %lld \n", (uint64_t)score1, *score1);

    return 0;
}
```

编译代码：

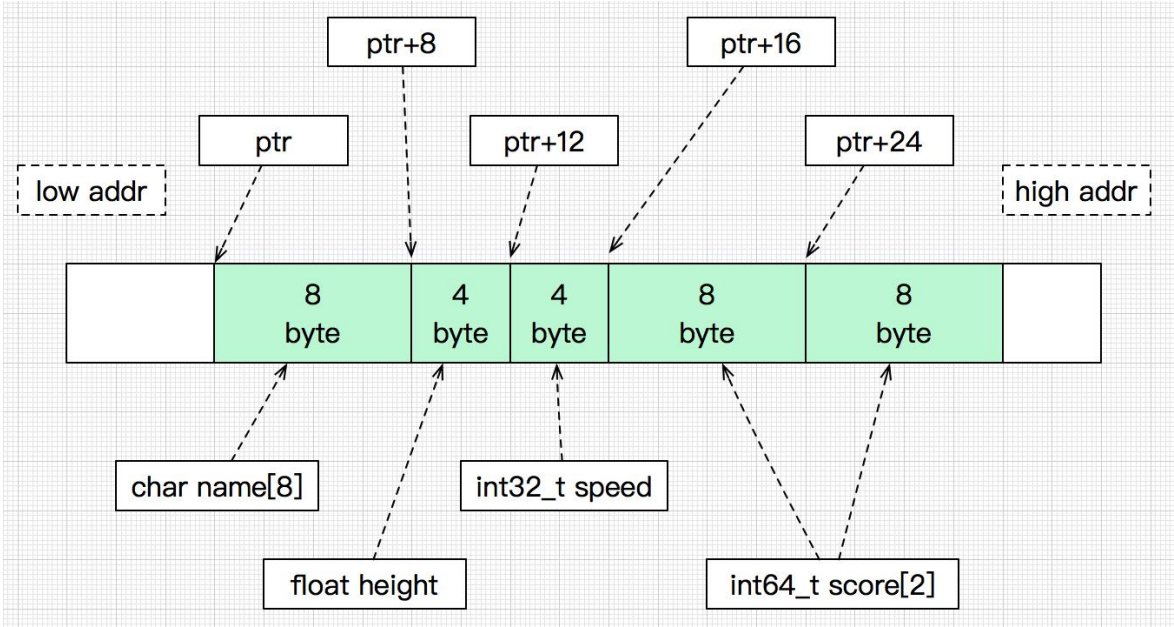
```
gcc scan_field.c -o scan_field
gcc scan_field.c -S -o scan_field.s
```

运行代码：

```
[root@192 struct]# ./scan_field
addr = 6295648  struct

addr = 6295648  name = BoQiTa
addr = 6295656  height = 22.330000
addr = 6295660  speed = 555
addr = 6295664  score[0] = 666
addr = 6295672  score[1] = 777
```

分析结果：
为了便于分析，提前把 Dog_t 的属性对齐。
指针变量，指向首地址。属性地址=首地址+偏移。使用这种方法，遍历全部属性。



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

功能	源码	汇编代码	分析

变量初始化	sprintf(dog.name, "BoQiTa");
	dog.height = 22.33F;
	dog.speed = 555;
	dog.score[0] = 666;
	dog.score[1] = 777;
		movl $1766944578, dog(%rip)
		movw $24916, dog+4(%rip)
		movb $0, dog+6(%rip)
		movl .LC0(%rip), %eax
		movl %eax, dog+8(%rip)
		movl $555, dog+12(%rip)
		movq $666, dog+16(%rip)
		movq $777, dog+24(%rip)
			依次给属性赋值。
			给字符串 name 赋值，没有调用 sprintf 函数，而使用 mov 指令，拷贝 int 值。
			给浮点数 height 赋值，22.33F 占用 4 个字节，使用 movl。
			给整数 speed 赋值，使用 movl。
			给整数 score[0]、score[1]赋值，使用 movq。

首地址	void *ptr = &dog;	movq $dog, -8(%rbp) movq -8(%rbp), %rax	把变量 dog 的地址，保存到栈上 -8(%rbp)。 后续从-8(%rbp)读取首地址。
name 的地址	char *name_ptr = ptr + 0;	movq -8(%rbp), %rax	属性地址=首地址
height 的地址	float *height_ptr = ptr + 8;	movq -8(%rbp), %rax addq $8, %rax	属性地址=首地址+8
speed 的地址	int32_t *speed_ptr = ptr + 12;	movq -8(%rbp), %rax addq $12, %rax	属性地址=首地址+12
score[0]的地址	int64_t *score0 = ptr + 16;	movq -8(%rbp), %rax addq $16, %rax	属性地址=首地址+16
score[1]的地址	int64_t *score1 = ptr + 24;	movq -8(%rbp), %rax addq $24, %rax	属性地址=首地址+24

问题： `sprintf(dog.name, "BoQiTa")` ，在汇编代码中，为什么可以不调用函数？
ASCII 码，1 个字符占用 1 个字节，字符和数字有对应关系。"BoQiTa"加上末尾字符，占用 7 个字符。
连续的 4 个字符，占用 4 个字节，可以用 1 个 int 表示，使用 `movl $1766944578, dog(%rip)` 。
连续的 2 个字符，占用 2 个字节，可以用 1 个 short 表示，使用 `movw $24916, dog+4(%rip)` 。
拼接最后的末尾字符'\0'，使用 `movb $0, dog+6(%rip)` 。

用 C 和汇编分析父子结构与首地址

struct 嵌套，使用同一个内存布局，多个属性在内存上展开。汇编代码，使用首地址和偏移。如果嵌套的 struct 是第一个属性，则嵌套 struct 的首地址与外层 struct 的首地址相同。

编写代码： head.c

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

// 动物
struct Animal
{
    char animal_name[16];
    int32_t animal_color;
};

// 鸟
struct Bird
{
    struct Animal animal;   // 第 1 个属性
    double bird_fly_height; // 第 2 个属性
};

// 动物的方法
void animal_sleep(struct Animal *animal)
{
    printf("Animal addr = %p \n", animal);
    printf("Animal %s with color %#x is sleeping \n\n",
           animal->animal_name, animal->animal_color);
}

// 鸟的方法
void bird_fly(struct Bird *bird)
{
    printf("Bird addr = %p \n", bird);
    printf("Bird %s is flying at height %f \n\n",
           bird->animal.animal_name, bird->bird_fly_height);
```

```
}

// 变量。
struct Bird lucy;

int main()
{
    // 初始化
    sprintf(lucy.animal.animal_name, "Lucy");
    lucy.animal.animal_color = 0xF1D6C8B7;
    lucy.bird_fly_height = 22.33D;

    printf("struct Animal  size = %d \n\n", sizeof(struct Animal));
    printf("struct Bird    size = %d \n\n", sizeof(struct Bird));

    // 调用动物的方法
    animal_sleep((struct Animal *)&lucy);

    // 调用鸟的方法
    bird_fly(&lucy);

    return 0;
}
```

编译代码：

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

运行代码：

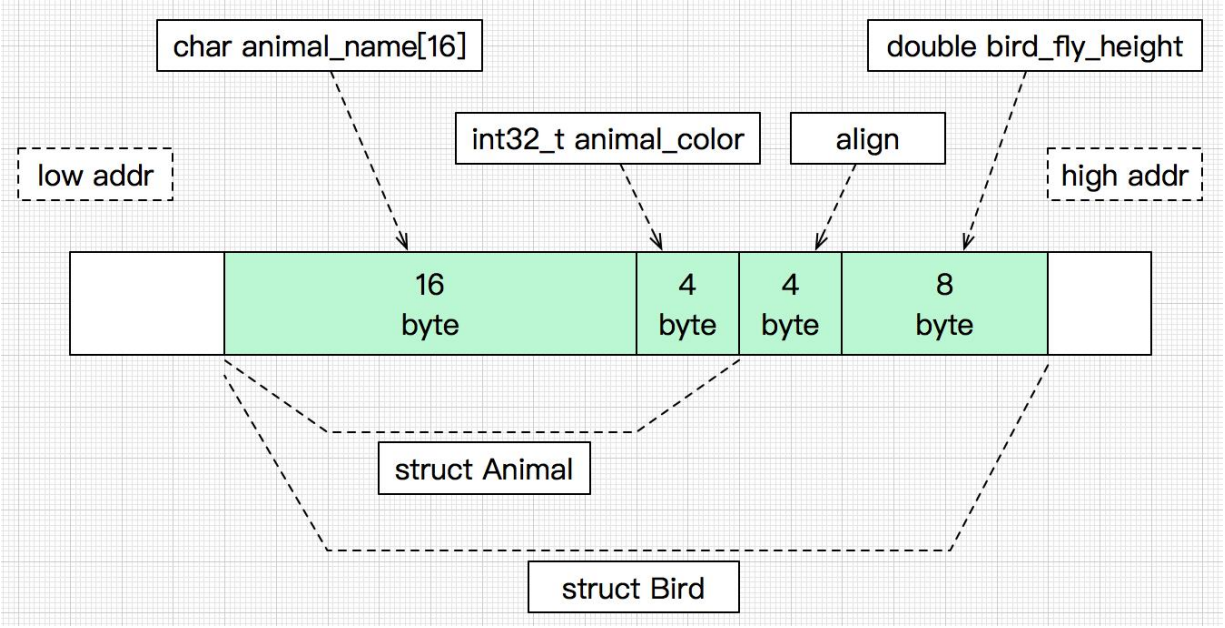
```
[root@192 struct]# ./head
struct Animal  size = 20

struct Bird    size = 32

Animal addr = 0x601060
Animal Lucy with color 0xf1d6c8b7 is sleeping

Bird addr = 0x601060
Bird Lucy is flying at height 22.330000
```

分析结果：
struct Animal 的大小为 20 字节，属性占用 20 字节，说明没有填充空白字节。
struct Bird 的大小为 32 字节，属性占用 28 字节，说明填充 4 个空白字节。
struct Bird 包含 struct Animal。并且，struct Animal 是第一个属性。Animal 的地址和 Bird 的地址相同，都为 0x601060。
内存布局，可以看出 2 个 struct 的嵌套结构。



查看文件 head.c、head.s，这里截取部分代码。

函数	源码	汇编代码	分析

animal_sleep	void animal_sleep(struct Animal *animal)
	{
	printf("Animal addr = %p \n", animal);
	printf("Animal %s with color %#x is sleeping \n\n",
	animal->animal_name, animal->animal_color);
	}
		animal_sleep:
		movq %rdi, -8(%rbp)
		movq -8(%rbp), %rax
		movl 16(%rax), %edx
		movq -8(%rbp), %rax
		movq %rax, %rsi
		movl $.LC1, %edi
		movl $0, %eax
		call printf
			入参 struct Animal *animal 为首地址，引用传递，使用 rdi。把 rdi 拷贝到栈上-8(%rbp)。
			属性 animal_name，其地址等于首地址，从-8(%rbp)读出地址，把该地址赋给 rsi。
			属性 animal_color,其地址等于首地址+16，把属性的值赋给 edx。

bird_fly	void bird_fly(struct Bird *bird)
	{
	printf("Bird addr = %p \n", bird);
	printf("Bird %s is flying at height %f \n\n",
	bird->animal.animal_name, bird->bird_fly_height);
	}
		bird_fly:
		movq %rdi, -8(%rbp)
		movq -8(%rbp), %rax
		movq 24(%rax), %rax
		movq -8(%rbp), %rdx
		movq %rax, -16(%rbp)
		movsd -16(%rbp), %xmm0
		movq %rdx, %rsi
		movl $.LC3, %edi
		movl $1, %eax
		call printf
			入参 struct Bird *bird 为首地址，引用传递，使用 rdi。把 rdi 拷贝到栈上-8(%rbp)。
			属性 animal_name，其地址等于首地址，从-8(%rbp)读出地址，把该地址赋给 rsi。
			属性 bird_fly_height，其地址等于首地址+24，先把属性的值赋给 rax，然后拷贝到栈上 -16(%rbp)，之后拷贝到 xmm0。

源码的层级关系 bird->animal.animal_name，在汇编层面被展开，对应汇编代码 `movq -8(%rbp), %rdx` 和 `movq %rdx, %rsi` 。
汇编代码，只看到首地址和偏移。animal_name 是第一个属性，偏移为 0，其地址等于首地址。

问题：为什么变量 struct Bird lucy 在函数 animal_sleep(struct Animal *animal)中，功能正常？
汇编代码，使用首地址和偏移。从内存布局看出，struct Bird 和 struct Animal 的首地址相同。
源码 `animal_sleep((struct Animal *)&lucy)` ，对应的汇编代码为

```
    movl    $lucy, %edi
    call    animal_sleep
```

源码 `bird_fly(&lucy)` ，对应的汇编代码为

```
    movl    $lucy, %edi
    call    bird_fly
```

2 个函数调用，都使用 `movl $lucy, %edi` ，把变量 lucy 的地址传入 edi。
强制类型转换 `(struct Animal *)&lucy` ，对应汇编代码 `movl $lucy, %edi` 。

用 C 和汇编分析属性偏移与属性读写

属性偏移，作用于内存地址计算。读属性，使用属性偏移。写属性，使用属性偏移。由属性找到所属 struct，使用属性偏移。

编写代码： offset_field.c

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

struct River
{
    int32_t river_width; // 4 字节
    int32_t river_depth; // 4 字节
};

struct Fish
{
    int64_t fish_length; // 8 字节
    struct River river;  // 8 字节

    void (*fish_swim)(struct Fish *fish); // 8 字节
};

// 获得某个属性的偏移。极简版。
#define field_off(field_name) \
    (uint64_t)(&(((struct Fish *)0)->field_name))

// 获得某个属性的偏移。与 field_off 功能等价。
```

```
// 1 地址 0 转为 struct 指针。
// 2 属性指针指向属性。
// 3 属性指针转为 64 位整数。
#define field_off_2(field_name, off_ptr)     \
    do                                       \
    {                                        \
        struct Fish *tmp = (struct Fish *)0; \
        void *addr = &(tmp->field_name);     \
        uint64_t off = (uint64_t)addr;       \
        *off_ptr = off;                      \
    } while (0)

void func_river(struct River *river)
{
    // 属性的偏移。
    uint64_t off = field_off(river);
    // struct Fish 的地址
    uint64_t base_addr = (uint64_t)river - off;
    // struct Fish 指针
    struct Fish *fish = (struct Fish *)base_addr;

    printf("\n");
    printf("River addr = %llu \n", (uint64_t)river);
    printf("Fish  addr = %llu \n", base_addr);
    printf("\n");
    printf("River width = %d  depth = %d \n", river->river_width, river->river_depth);
    printf("Fish  length = %lld \n", fish->fish_length);

    // 调用方法
    fish->fish_swim(fish);
}

void func_fish_swim(struct Fish *fish)
{
    printf("Fish {length=%lld} is swimming in River {width=%d depth=%d} \n",
           fish->fish_length, fish->river.river_depth, fish->river.river_width);
}

// 变量
struct Fish fish;

int main()
{
    // 用 2 种方式，取属性的偏移
    uint64_t river_off1 = field_off(river);
    printf("river_off1 = %llu \n", river_off1);
    uint64_t river_off2;
    field_off_2(river, &river_off2);
    printf("river_off2 = %llu \n", river_off2);
```

```
    uint64_t river_depth_off = field_off(river.river_depth);
    printf("river_depth_off = %llu \n", river_depth_off);

    // 变量
    fish.fish_length = 100;
    fish.river.river_width = 3;
    fish.river.river_depth = 2;
    fish.fish_swim = func_fish_swim;

    // 方法。入参为属性指针。
    func_river(&(fish.river));

    return 0;
}
```

编译代码：

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

运行代码：

```
[root@192 struct]# ./offset_field
river_off1 = 8
river_off2 = 8
river_depth_off = 12

River addr = 6295640
Fish  addr = 6295632

River width = 3  depth = 2
Fish  length = 100
Fish {length=100} is swimming in River {width=2 depth=3}
```

分析结果：

计算某个属性的偏移，使用宏 `field_off(field_name)` 、 `field_off_2(field_name, off_ptr)` 。这 2 个宏，本质相同，功能等价。从结果看出，2 个宏计算出的 river 属性的偏移都为 8 字节。

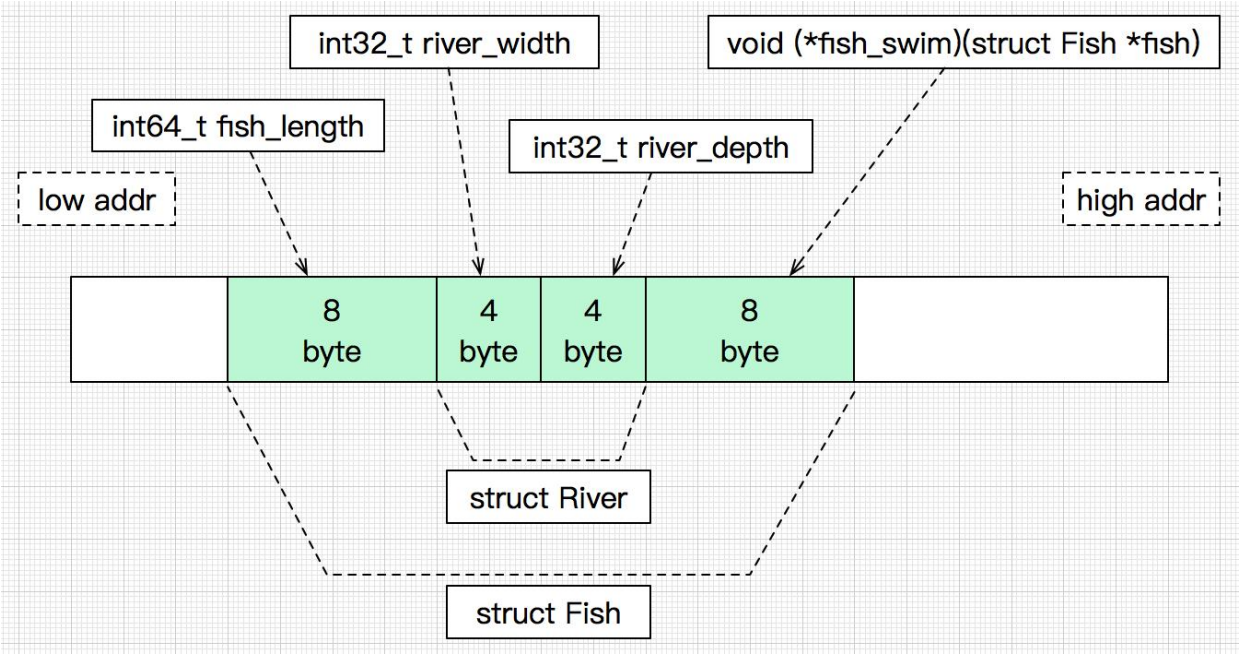
查找某个属性的偏移，源码为

```
#define field_off(field_name) \
    (uint64_t)(&(((struct Fish *)0)->field_name))
uint64_t river_off1 = field_off(river);
```

汇编代码为 `movq $8, -8(%rbp)` ，非常简洁，直接把偏移 8 写在代码里。

编译时，编译器已经计算出 river 属性的偏移是 8 字节。

内存布局。全部属性在内存上展平。River 的地址与 Fish 的地址相差 8 字节。`void (*fish_swim)(struct Fish *fish)` 表示指针属性，占用 8 字节。



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

属性	源码	汇编代码	分析
fish_length	fish.fish_length = 100;	movq $100, fish(%rip)	使用偏移 fish+0。 第一个属性的地址与首地址相同。
river_width	fish.river.river_width = 3;	movl $3, fish+8(%rip)	使用偏移 fish+8。
river_depth	fish.river.river_depth = 2;	movl $2, fish+12(%rip)	使用偏移 fish+12。
fish_swim	fish.fish_swim = func_fish_swim;	movq $func_fish_swim, fish+16(%rip)	使用偏移 fish+16。 把函数 func_fish_swim 的地址，赋给 fish_swim 属性。

读属性

函数 `void func_river(struct River *river)` 的内容为：入参使用 `struct River *river` ，用宏 field_off 算出 river 属性的偏移，然后得出 `struct Fish *fish` ，最后遍历 river、fish 的全部属性。

调用 struct 的函数 `fish->fish_swim(fish)` ，对应的汇编代码为

```
    movq    -24(%rbp), %rax     算出 fish 的地址，写到 rax
    movq    16(%rax), %rax      使用首地址+16，得出 fish_swim 的地址，然后把 fish_swim 的值写到 rax 。
    movq    -24(%rbp), %rdx
    movq    %rdx, %rdi
    call    *%rax               调用 fish_swim 指向的函数。
```

用 C 和汇编分析值传递与引用传递

值传递，依次拷贝每个属性，对应的汇编代码多。

引用传递，只需要拷贝地址，对应的汇编代码少。

编写代码： refer.c

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

struct Cat
{
    int64_t speed;
    int64_t color;
    int64_t age;
};

// 原始值
struct Cat tom = {
    .speed = 555,
    .color = 33,
    .age = 22};

// 复制
struct Cat copy;

// 指针
struct Cat *refer;

int main()
{
    // 值传递
    copy = tom;

    // 引用传递
    refer = &tom;

    printf("tom    addr = %p \n", &tom);
    printf("copy   addr = %p \n", &copy);
    printf("refer  addr = %p  value = %#llx \n", &refer, (uint64_t)refer);

    return 0;
}
```

编译代码：

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

运行代码：

```
[root@192 struct]# ./refer
tom    addr = 0x601040
copy   addr = 0x601080
refer  addr = 0x601070  value = 0x601040
```

分析结果：
3 个变量的地址不相同。refer 是一个指针。其值为 0x601040，等于 tom 的地址。

查看文件 refer.c、refer.s，这里截取部分代码。

属性	源码	汇编代码	分析
初始化	struct Cat tom = { .speed = 555, .color = 33, .age = 22};	tom: .quad 555 .quad 33 .quad 22	3 个属性，依次排列，各占用 8 个字节。 .quad 表示 8 个字节整数。

值传递	copy = tom;	movq tom(%rip), %rax
		movq %rax, copy(%rip)
		movq tom+8(%rip), %rax
		movq %rax, copy+8(%rip)
		movq tom+16(%rip), %rax
		movq %rax, copy+16(%rip)
			使用首地址+偏移，依次拷贝每个属性。
			把 tom+0 地址的 8 个字节，拷贝到 copy+0 地址。
			把 tom+8 地址的 8 个字节，拷贝到 copy+8 地址。
			把 tom+16 地址的 8 个字节，拷贝到 copy+16 地址。

引用传递	refer = &tom;	movq $tom, refer(%rip)	只需要拷贝地址。 把 tom 的地址，赋给 refer。