第19讲：自定义类型：结构体

目录：

正文开始

1. 结构体类型的声明

前面我们在学习操作符的时候，已经学习了结构体的知识，这里稍微复习一下。

1.1 结构体回顾

结构是一些值的集合，这些值称为成员变量。结构的每个成员可以是不同类型的变量。

1.1.1 结构的声明

```
struct tag
{
    member-list;
}variable-list;
```

例如描述一个学生：

```
struct Stu
{
    char name[20];//名字
    int age;//年龄
    char sex[5];//性别
```

```
    char id[20];//学号
}; //分号不能丢
```

1.1.2 结构体变量的创建和初始化

```
#include <stdio.h>
struct Stu
{
    char name[20];//名字
    int age;//年龄
    char sex[5];//性别
    char id[20];//学号
};

int main()
{
    //按照结构体成员的顺序初始化
    struct Stu s = { "张三", 20, "男", "20230818001" };
    printf("name: %s\n", s.name);
    printf("age : %d\n", s.age);
    printf("sex : %s\n", s.sex);
    printf("id  : %s\n", s.id);

    //按照指定的顺序初始化
    struct Stu s2 = { .age = 18, .name = "lisi", .id = "20230818002", .sex = "女" };
    printf("name: %s\n", s2.name);
    printf("age : %d\n", s2.age);
    printf("sex : %s\n", s2.sex);
    printf("id  : %s\n", s2.id);
    return 0;
}
```

1.2 结构的特殊声明

在声明结构的时候，可以不完全的声明。

比如：

```
//匿名结构体类型
struct
{
    int a;
    char b;
```

```
    float c;
}x;

struct
{
    int a;
    char b;
    float c;
}a[20], *p;
```

上面的两个结构在声明的时候省略掉了结构体标签（tag）。

那么问题来了？

```
//在上面代码的基础上，下面的代码合法吗？
p = &x;
```

警告：

编译器会把上面的两个声明当成完全不同的两个类型，所以是非法的。

匿名的结构体类型，如果没有对结构体类型重命名的话，基本上只能使用一次。

1.3 结构的自引用

在结构中包含一个类型为该结构本身的成员是否可以呢？

比如，定义一个链表的节点：

```
struct Node
{
    int data;
    struct Node next;
};
```

上述代码正确吗？如果正确，那`sizeof(struct Node)`是多少？

仔细分析，其实是不行的，因为一个结构体中再包含一个同类型的结构体变量，这样结构体变量的大小就会无穷的大，是不合理的。

正确的自引用方式：

```
struct Node
```

```
{
    int data;
    struct Node* next;
};
```

在结构体自引用使用的过程中，夹杂了`typedef`对匿名结构体类型重命名，也容易引入问题，看看下面的代码，可行吗？

```
typedef struct
{
    int data;
    Node* next;
}Node;
```

答案是不行的，因为Node是对前面的匿名结构体类型的重命名产生的，但是在匿名结构体内部提前使用Node类型来创建成员变量，这是不行的。

解决方案如下：定义结构体不要使用匿名结构体了

```
typedef struct Node
{
    int data;
    struct Node* next;
}Node;
```

2. 结构体内存对齐

我们已经掌握了结构体的基本使用了。

现在我们深入讨论一个问题：计算结构体的大小。

这也是一个特别热门的考点：`结构体内存对齐`

2.1 对齐规则

首先得掌握结构体的对齐规则：

> 1. 结构体的第一个成员对齐到和结构体变量起始位置偏移量为0的地址处
> 2. 其他成员变量要对齐到某个数字（对齐数）的整数倍的地址处。
>
> **对齐数** = 编译器默认的一个对齐数 与 该成员变量大小的**较小值**。

- `VS 中默认的值为 8`

- Linux中 gcc 没有默认对齐数，对齐数就是成员自身的大小

3. 结构体总大小为**最大对齐数**（结构体中每个成员变量都有一个对齐数，所有对齐数中最大的）的整数倍。

4. 如果嵌套了结构体的情况，嵌套的结构体成员对齐到自己的成员中最大对齐数的整数倍处，结构体的整体大小就是所有最大对齐数（含嵌套结构体中成员的对齐数）的整数倍。

```c
//练习1
struct S1
{
    char c1;
    int i;
    char c2;
};
printf("%d\n", sizeof(struct S1));

//练习2
struct S2
{
    char c1;
    char c2;
    int i;
};
printf("%d\n", sizeof(struct S2));

//练习3
struct S3
{
    double d;
    char c;
    int i;
};
printf("%d\n", sizeof(struct S3));

//练习4-结构体嵌套问题
struct S4
{
    char c1;
    struct S3 s3;
    double d;
};
printf("%d\n", sizeof(struct S4));
```

2.2 为什么存在内存对齐？

> 大部分的参考资料都是这样说的：

1. **平台原因 (移植原因)：**

不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2. **性能原因：**

数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。假设一个处理器总是从内存中取8个字节，则地址必须是8的倍数。如果我们能保证将所有的double类型的数据的地址都对齐成8的倍数，那么就可以用一个内存操作来读或者写值了。否则，我们可能需要执行两次内存访问，因为对象可能被分放在两个8字节内存块中。

总体来说：结构体的内存对齐是拿**空间**来换取**时间**的做法。

那在设计结构体的时候，我们既要满足对齐，又要节省空间，如何做到：

> 让占用空间小的成员尽量集中在一起

```c
//例如:
struct S1
{
    char c1;
    int i;
    char c2;
};

struct S2
{
    char c1;
    char c2;
    int i;
};
```

`S1` 和 `S2` 类型的成员一模一样，但是 `S1` 和 `S2` 所占空间的大小有了一些区别。

2.3 修改默认对齐数

`#pragma` 这个预处理指令，可以改变编译器的默认对齐数。

```c
#include <stdio.h>

#pragma pack(1)//设置默认对齐数为1
struct S
{
    char c1;
    int i;
    char c2;
};
#pragma pack()//取消设置的对齐数，还原为默认

int main()
{
    //输出的结果是什么?
    printf("%d\n", sizeof(struct S));
    return 0;
}
```

结构体在对齐方式不合适的时候，我们可以自己更改默认对齐数。

3. 结构体传参

```c
struct S
{
    int data[1000];
    int num;
};

struct S s = {{1,2,3,4}, 1000};
//结构体传参
void print1(struct S s)
{
    printf("%d\n", s.num);
}
//结构体地址传参
void print2(struct S* ps)
{
    printf("%d\n", ps->num);
}

int main()
{
```

```
    print1(s);  //传结构体
    print2(&s); //传地址
    return 0;
}
```

上面的 `print1` 和 `print2` 函数哪个好些?

答案是：首选print2函数。

原因：

> 函数传参的时候，参数是需要压栈，会有时间和空间上的系统开销。

> 如果传递一个结构体对象的时候，结构体过大，参数压栈的的系统开销比较大，所以会导致性能的下降。

结论：

结构体传参的时候，要传结构体的地址。

4. 结构体实现位段

结构体讲完就得讲讲结构体实现 `位段` 的能力。

4.1 什么是位段

位段的声明和结构是类似的，有两个不同：

1. 位段的成员必须是 `int、unsigned int 或signed int` ，在C99中位段成员的类型也可以选择其他类型。
2. 位段的成员名后边有一个冒号和一个数字。

比如：

```
struct A
{
    int _a:2;
    int _b:5;
    int _c:10;
    int _d:30;
};
```

A就是一个位段类型。

那位段A所占内存的大小是多少?

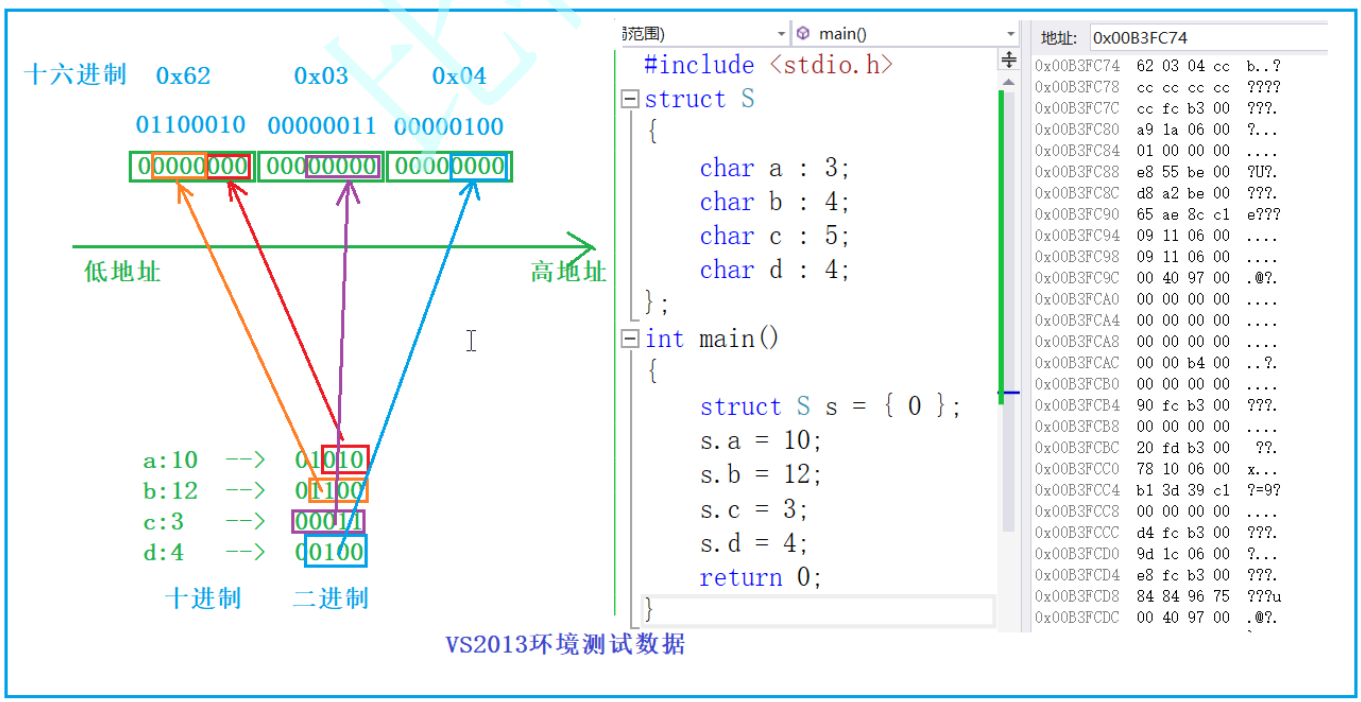
```
printf("%d\n", sizeof(struct A));
```

4.2 位段的内存分配

1. 位段的成员可以是 `int` `unsigned int` `signed int` 或者是 `char` 等类型
2. 位段的空间上是按照需要以4个字节（`int`）或者1个字节（`char`）的方式来开辟的。
3. 位段涉及很多不确定因素，位段是不跨平台的，注重可移植的程序应该避免使用位段。

```
//一个例子
struct S
{
    char a:3;
    char b:4;
    char c:5;
    char d:4;
};
struct S s = {0};
s.a = 10;
s.b = 12;
s.c = 3;
s.d = 4;

//空间是如何开辟的?
```



VS2013环境测试数据

4.3 位段的跨平台问题

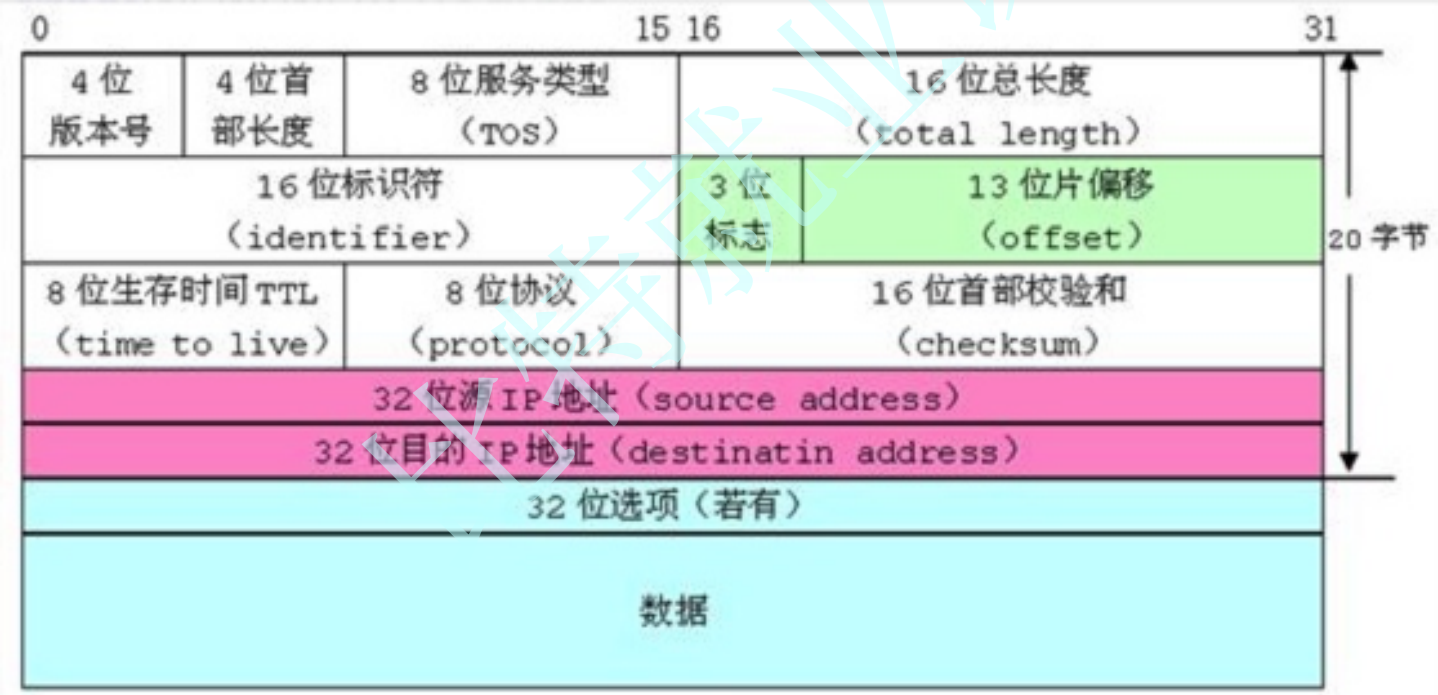
1. int 位段被当成有符号数还是无符号数是不确定的。
2. 位段中最大位的数目不能确定。（16位机器最大16，32位机器最大32，写成27，在16位机器会出问题。
3. 位段中的成员在内存中从左向右分配，还是从右向左分配，标准尚未定义。
4. 当一个结构包含两个位段，第二个位段成员比较大，无法容纳于第一个位段剩余的位时，是舍弃剩余的位还是利用，这是不确定的。

总结：

跟结构相比，位段可以达到同样的效果，并且可以很好的节省空间，但是有跨平台的问题存在。

4.4 位段的应用

下图是网络协议中，IP数据报的格式，我们可以看到其中很多的属性只需要几个bit位就能描述，这里使用位段，能够实现想要的效果，也节省了空间，这样网络传输的数据报大小也会较小一些，对网络的畅通是有帮助的。



4.5 位段使用的注意事项

位段的几个成员共有同一个字节，这样有些成员的起始位置并不是某个字节的起始位置，那么这些位置处是没有地址的。内存中每个字节分配一个地址，一个字节内部的bit位是没有地址的。

所以不能对位段的成员使用&操作符，这样就不能使用scanf直接给位段的成员输入值，只能是先输入放在一个变量中，然后赋值给位段的成员。

```
1 struct A
2 {
```

```
    int _a : 2;
    int _b : 5;
    int _c : 10;
    int _d : 30;
};


int main()
{
    struct A sa = {0};
    scanf("%d", &sa._b);//这是错误的

    //正确的示范
    int b = 0;
    scanf("%d", &b);
    sa._b = b;
    return 0;
}
```

完