自定义类型：结构体，枚举，联合

本章重点

- 结构体
  - 结构体类型的声明
  - 结构的自引用
  - 结构体变量的定义和初始化
  - 结构体内存对齐
  - 结构体传参
  - 结构体实现位段（位段的填充&可移植性）
- 枚举
  - 枚举类型的定义
  - 枚举的优点
  - 枚举的使用
- 联合
  - 联合类型的定义
  - 联合的特点
  - 联合大小的计算

正文开始

结构体

结构体的声明

结构的基础知识

> 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。

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
```

特殊的声明

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。

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？

```
//代码1
struct Node
{
    int data;
    struct Node next;
};
//可行否？
如果可以，那sizeof(struct Node)是多少？
```

正确的自引用方式：

```
//代码2
struct Node
{
    int data;
    struct Node* next;
};
```

注意：

```
//代码3
typedef struct
{
    int data;
    Node* next;
}Node;
//这样写代码，可行否？

//解决方案：
typedef struct Node
{
    int data;
    struct Node* next;
}Node;
```

结构体变量的定义和初始化

有了结构体类型，那如何定义变量，其实很简单。

```
struct Point
{
    int x;
    int y;
}p1;                //声明类型的同时定义变量p1
struct Point p2;    //定义结构体变量p2


//初始化：定义变量的同时赋初值。
struct Point p3 = {x, y};

struct Stu        //类型声明
{
    char name[15];//名字
    int age;      //年龄
};
struct Stu s = {"zhangsan", 20};//初始化

struct Node
{
    int data;
    struct Point p;
    struct Node* next;
}n1 = {10, {4,5}, NULL};            //结构体嵌套初始化
```

```
struct Node n2 = {20, {5, 6}, NULL};//结构体嵌套初始化
```

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`

```
//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));
```

考点 如何计算？ 首先得掌握结构体的对齐规则：

> 1. 第一个成员在与结构体变量偏移量为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 该成员大小的**较小值**。

> - `VS中默认的值为8`
> - `Linux中的默认值为4`
>
> 3. 结构体总大小为最大对齐数（每个成员变量都有一个对齐数）的整数倍。
> 4. 如果嵌套了结构体的情况，嵌套的结构体对齐到自己的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（含嵌套结构体的对齐数）的整数倍。

为什么存在内存对齐?

> 大部分的参考资料都是如是说的：

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。

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。

```
//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。

修改默认对齐数

之前我们见过了`#pragma`这个预处理指令，这里我们再次使用，可以改变我们的默认对齐数。

```
#include <stdio.h>
#pragma pack(8)//设置默认对齐数为8
struct S1
{
    char c1;
    int i;
    char c2;
```

```
};
#pragma pack()//取消设置的默认对齐数，还原为默认

#pragma pack(1)//设置默认对齐数为8
struct S2
{
    char c1;
    int i;
    char c2;
};
#pragma pack()//取消设置的默认对齐数，还原为默认
int main()
{
    //输出的结果是什么？
    printf("%d\n", sizeof(struct S1));
    printf("%d\n", sizeof(struct S2));
    return 0;
}
```

结论：

> 结构在对齐方式不合适的时候，我么可以自己更改默认对齐数。

百度笔试题：

> 写一个宏，计算结构体中某变量相对于首地址的偏移，并给出说明

考察：`offsetof`宏的实现

注：这里还没学习宏，可以放在宏讲解完后再实现。

结构体传参

直接上代码：

```
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。

位段

结构体讲完就得讲讲结构体实现`位段`的能力。

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`。
>
> 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的大小是多少？

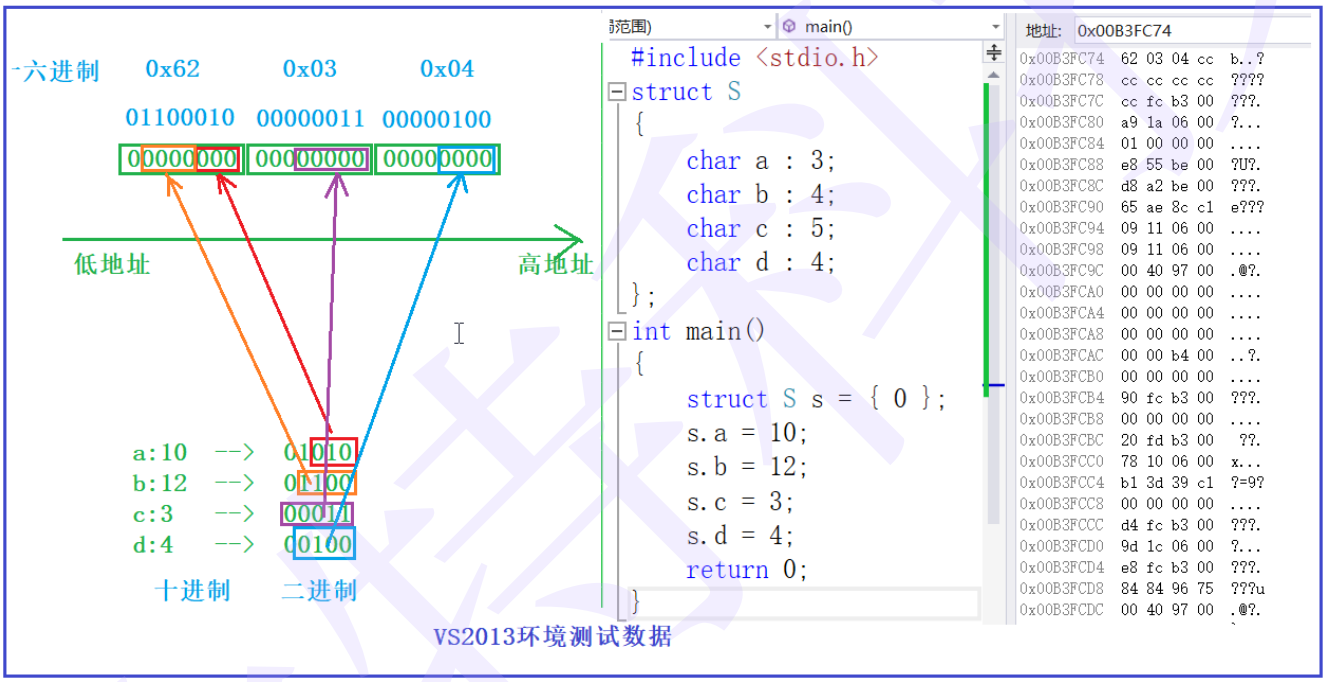
```
printf("%d\n", sizeof(struct A));
```

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`（属于整形家族）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。

```c
//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？
```



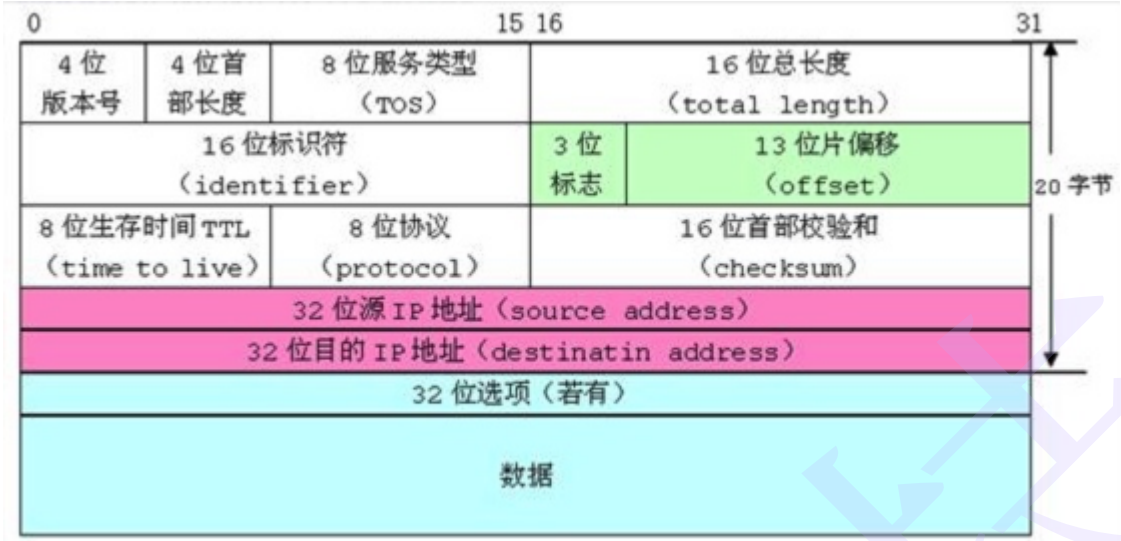
位段的跨平台问题

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，但是可以很好的节省空间，但是有跨平台的问题存在。

位段的应用



枚举

枚举顾名思义就是一一列举。

把可能的取值一一列举。

比如我们现实生活中：

> 一周的星期一到星期日是有限的7天，可以一一列举。
>
> 性别有：男、女、保密，也可以一一列举。
>
> 月份有12个月，也可以一一列举
>
> 颜色也可以一一列举。

这里就可以使用枚举了。

枚举类型的定义

```
enum Day//星期
{
    Mon,
    Tues,
    Wed,
    Thur,
    Fri,
    Sat,
    Sun
};
enum Sex//性别
{
    MALE,
    FEMALE,
    SECRET
};
enum Color//颜色
{
    RED,
```

```
    GREEN,
    BLUE
};
```

以上定义的`enum Day`，`enum Sex`，`enum Color`都是枚举类型。 {}中的内容是枚举类型的可能取值，也叫`枚举常量`。

这些可能取值都是有值的，默认从0开始，一次递增1，当然在定义的时候也可以赋初值。 例如：

```
enum Color//颜色
{
        RED=1,
        GREEN=2,
        BLUE=4
};
```

枚举的优点

> 为什么使用枚举？

我们可以使用`#define`定义常量，为什么非要使用枚举？ 枚举的优点：

1. 增加代码的可读性和可维护性
2. 和#define定义的标识符比较枚举有类型检查，更加严谨。
3. 防止了命名污染（封装）
4. 便于调试
5. 使用方便，一次可以定义多个常量

枚举的使用

```
enum Color//颜色
{
        RED=1,
        GREEN=2,
        BLUE=4
};

enum Color clr = GREEN;//只能拿枚举常量给枚举变量赋值，才不会出现类型的差异。
clr = 5;               //ok??
```

联合（共用体）

联合类型的定义

联合也是一种特殊的自定义类型 这种类型定义的变量也包含一系列的成员，特征是这些成员公用同一块空间（所以联合也叫共用体）。 比如：

```
//联合类型的声明
union Un
{
        char c;
        int i;
};

//联合变量的定义
union Un un;
//计算连个变量的大小
printf("%d\n", sizeof(un));
```

联合的特点

> 联合的成员是共用同一块内存空间的，这样一个联合变量的大小，至少是最大成员的大小（因为联合至少得有能力保存最大的那个成员）。

```
union Un
{
        int i;
        char c;
};
union Un un;

// 下面输出的结果是一样的吗？
printf("%d\n", &(un.i));
printf("%d\n", &(un.c));

//下面输出的结果是什么？
un.i = 0x11223344;
un.c = 0x55;
printf("%x\n", un.i);
```

面试题：

> 判断当前计算机的大小端存储

联合大小的计算

- 联合的大小至少是最大成员的大小。
- 当最大成员大小不是最大对齐数的整数倍的时候，就要对齐到最大对齐数的整数倍。

比如：

```
union Un1
{
        char c[5];
        int i;
};
union Un2
{
        short c[7];
        int i;
};
//下面输出的结果是什么？
printf("%d\n", sizeof(union Un1));
printf("%d\n", sizeof(union Un2));
```

本章完