自定义类型：结构体，枚举，联合

笔记本： My Notebook
创建时间： 2023/10/21 20:52 **更新时间：** 2023/10/28 14:57
作者： lwb

结构体的声明
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

```
//定义结构体变量的方式
struct Stu
{
        // 成员变量
        char name[20];
        char tele[20];
        char sex[10];
        int age;
}s4,s5,s6;// 全局变量
struct Stu s3; // 全局变量
int main()
{
        struct Stu s1; // 局部变量
        struct Stu s2;
        return 0;
}
```

```
//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;
//以上的两个结构在声明的时候省略掉了结构体标签(tag)
//编译器会把上面两个声明当成完全不同的两个类型
```

结构体的自引用
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

```
struct Node
{
    int data; // 4个字节
    struct Node* next; // 4/8个字节
}
```

结构体内存对齐
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中默认的值是8
- Linux中没有默认对齐数，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（每个成员都有一个对齐数）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

```
struct S1
{
        char c1;
        int a;
        char c2;
};
struct S2
{
        char c1;
        char c2;
        int a;
};
int main()
{
        struct S1 s1 = { 0 };
        printf("%d\n", sizeof(s1)); // 12
        struct S2 s2 = { 0 };
        printf("%d\n", sizeof(s2)); // 8
}
```

为什么存在内存对齐?
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。

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？
让占有空间小的成员尽量集中在一起。

修改默认对齐数
#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。

```
//设置默认对齐数位4
#pragma pack(4)
struct S
{
        char c1; //1
        // 7
        double d; // 8
};
#pragma pack()
//取消设置的默认对齐数
int main()
{
        struct S s;
        printf("%d\n", sizeof(s));
        return 0;
}
```

写一个宏，计算结构体中某变量相对于首地址的偏移
考察：offsetof宏的实现

```
#include <stddef.h>
struct S
{
        char c;
        int i;
        double d;
};
int main()
{
        printf("%d\n", offsetof(struct S, c)); // 0
        printf("%d\n", offsetof(struct S, i)); // 4
        printf("%d\n", offsetof(struct S, d)); // 8
        return 0;
}
```

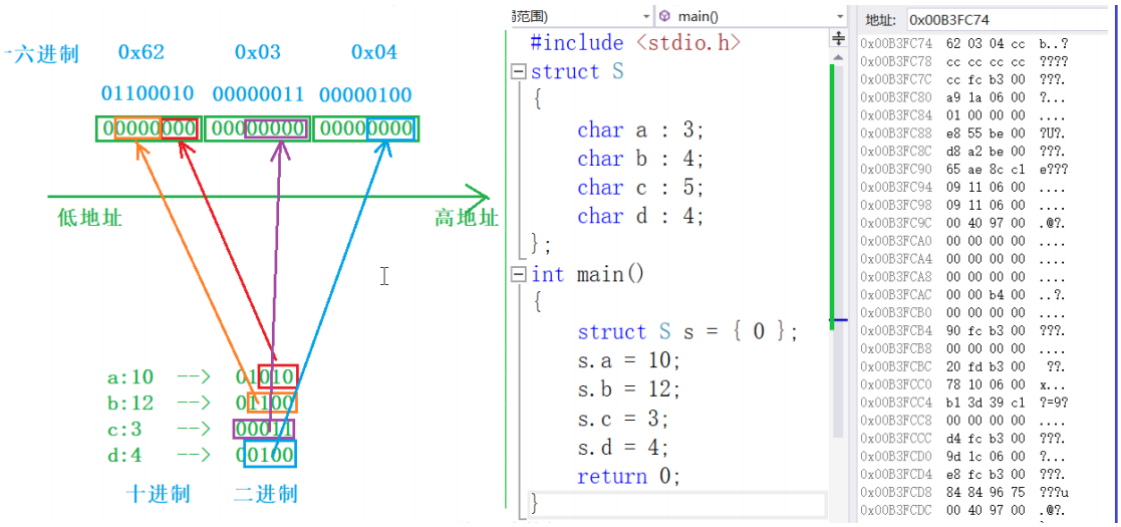
结构体传参
函数传参的时候，参数是需要压栈，会有时间和空间上的系统开销
如果传递一个结构体对象的时候，结构体过大，参数压栈的系统开销比较大，所以会导致性能的下降
结论： 结构体传参的时候，要传结构体的地址

位段
1.位段的声明必须是int、unsigned int 或者 signed int
2.位段的成员名后边有一个冒号和一个数字

```
// A就是一个位段类型
struct A
{
    int _a:2;
    int _b:5;
    int _c:10;
    int _d:30;
};
```

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。

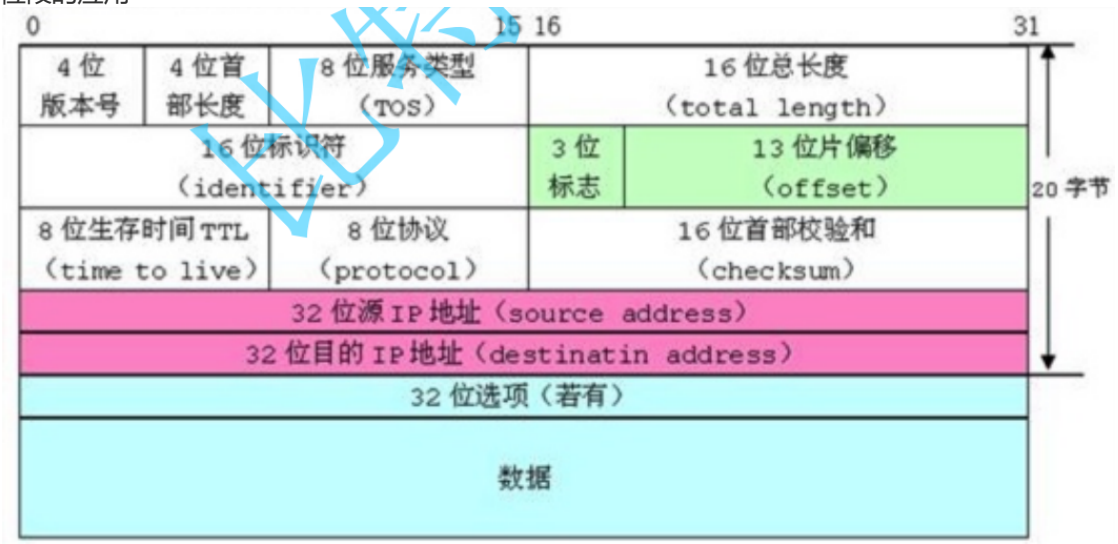
```
//位段：二进制位 - 可以节省空间
struct S
{
        int a : 2;
        int b : 5;
        int c : 10;
        int d : 30;
};
// 47bit - 6个字节 * 8 = 48位
int main()
{
        struct S s;
        printf("%d\n", sizeof(s)); // 8
        return 0;;
}
```



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
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

位段的应用



枚举

```
// 枚举的定义
enum Day//星期
{
Mon,
Tues,
Wed,
Thur,
Fri,
Sat,
Sun
};
// 这些可能取值都是有值的，默认从0开始，依次递增1，在声明枚举类型的时候也可以赋初值
```

枚举的优点
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定义的标识符相比，枚举有类型检查，更加严谨
3.防止了命名污染
4.便于调试
5.使用方便，依次可以定义多个常量

联合(共用体)
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，特征是这些成员共用同一块空间。

```
union Un
{
        char c;
        int i;
};
int main()
{
        union Un u;
        printf("%d\n", sizeof(u)); // 4
        printf("%p\n", &(u.c)); //0000000AD6D1FB64
        printf("%p\n", &(u.i)); //0000000AD6D1FB64
        printf("%p\n", &u); //0000000AD6D1FB64
        return 0;
}
```

联合的特点：
联合的成员是共用同一块内存空间，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)

```
// 第一种方式
int check_sys()
{
        //返回1表示小端
        //返回0表示大端
        int a = 1;
        return *(char*)&a;
}

//第二种方式
int check_sys()
{
        union
        {
               char c;
               int i;
        }u;
        //返回1表示小端
```

```
        //返回0表示大端
        u.i = 1;
        return u.c;
}
int main()
{
        if (check_sys() == 1)
        {
                printf("小端");
        }
        else
        {
                printf("大端");
        }
        // 低地址-------------------高地址
        // ....[][][11][22][33][44][][]... -大端字节序存储模式
        // ....[][][44][33][22][11][][]... -小端字节序存储模式
        // 大小端字节序问题
        return 0;
}
```

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

```
union Un
{
        int a;
        char arr[5]; // 对齐数为元素char的对齐数
};
int main()
{
        union Un u;
        printf("%d\n", sizeof(u));
        return 0;
}
```