自定义数据类型

@TOC

前言

C语言中，提供了本身的数据类型，如：char、int、long、long long、float、double等，那么还有给我们提供自定义类型的实现方法。

- **结构体**
- **枚举**
- **联合**

结构体

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。

结构体类型：

```
struct tag
{
 member-list;  //成员变量列表（类型是可以不同）
}variable-list;
```

==例如，定义一个学生：==

```
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
```

==例如，定义书：==

```
//声明结构体类型
struct Book{
    char name[20];  //书名
    char id[20];  //索书号
    float price;  //定价
    char author[15];  //作者
}b1, b2;  //顺带直接创建两个结构体变量，其类型为struct Book
```

当然，也可以不用再创建好类型时进行创建变量。另外，在main函数外创建结构体类型，紧接着创建变量，如struct Book结构体类型的b1，b2变量其属于全局变量。

还有一种匿名结构体类型，在声明结构的时候，可以不完全的声明，如下：

```
//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;
```

考虑如下的操作：

```
#include <stdio.h>
//结构体数据类型
//匿名结构体类型（只能在声明时使用一次）
struct
{
    int a;
    char b;
    float c;
}x;  //全局变量指针

//匿名结构体类型
struct
{
    int a;
    char b;
    float c;
}* p;

int main() {
    //考虑这样的代码
    p = &x;
}
```

其实这是行不通的。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。

结构体自引用

代码如下：

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

显然这种做法是没有结果的。

考虑用结构体指针：

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

```
typedef struct Node {
    int data;  //数据域
    struct Node* next;  //指针域

}Node;  //重新命名为Node

int main() {
    Node n;  //创建节点
    return 0;
}
```

这样的做法是可以的，其实也不一定要重命名，这取决于实际的应用场景，不进行结构体类型的重命名，那么在main函数里创建 n 时就要用struct Node n;

结构体变量的定义和初始化

```
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
```

补充：关于初始化

初始化可以按自己定义的顺序来赋值。

```
struct Stu {
    char name[10];
    int age;
};

int main() {
    //也可以不按顺序初始化，自己来按想要的初始化顺序来
    struct Stu s = { .age = 18,.name = "张三" };

    printf("%s %d\n", s.name, s.age);
    return 0;

}
```

结构体内存对齐

现在深入讨论一个问题：**计算结构体的大小。**

考虑如下代码：

```
struct s1
{
    char c1;
    int i;
    char c2;
};

struct s2 {
    char c1;
    char c2;
    int i;
};
//结构体内存对齐

int main() {
    printf("%d %d\n", sizeof(struct s1), sizeof(struct s2));
    return 0;
}
```

结果如下：

```
struct s1
{
    char c1;
    int i;
    char c2;
};

struct s2 {
    char c1;
    char c2;
    int i;
};
//结构体内存对齐

int main() {
    printf("%d %d\n", sizeof(struct s1), sizeof(struct s2));
    return 0;
}
```

```
Microsoft Visual Studio 调试控制台
12 8

E:\Github_Gitee\C\Struct\x64\Debug\Struct.exe (进程 13088)已退出，代码为 0。
按任意键关闭此窗口. . .
```


结果其实比较意外，在给结构体分配内存时，

```
//offset这个东西，type，member。查看内存偏移量
#include <stddef.h>
int main() {
    struct s1 ss1;
    printf("%d ", offsetof(struct s1, c1));
    printf("%d ", offsetof(struct s1, i));
    printf("%d ", offsetof(struct s1, c2));
    printf("\n");
    printf("%d %d\n", sizeof(struct s1), sizeof(struct s2));
    return 0;
}
```

```
int main() {
    struct s1 ss1;
    printf("%d ", offsetof(struct s1, c1));
    printf("%d ", offsetof(struct s1, i));
    printf("%d ", offsetof(struct s1, c2));
    printf("\n");
    printf("%d %d\n", sizeof(struct s1), siz
    return 0;
}
```

```
Microsoft Visual Studio 调试控制台
0 4 8
12 8

E:\Github_Gitee\C\Struct\x64\D
按任意键关闭此窗口. . .
```

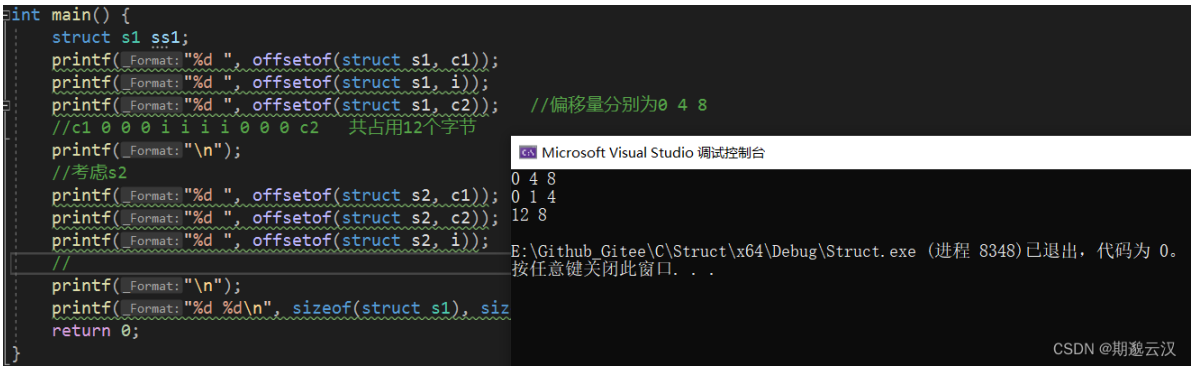

在这里，偏移量分别为0 4 8，在内存中这就是三个变量其实存放位置的相对位置。

```
printf("%d ", offsetof(struct s1, c1));
printf("%d ", offsetof(struct s1, i));
printf("%d ", offsetof(struct s1, c2));    //偏移量分别为0 4 8
//c1 0 0 0 i i i i 0 0 0 c2   共占用12个字节
printf("\n");
```


考虑s2

```
//考虑s2
    printf("%d ", offsetof(struct s2, c1));
    printf("%d ", offsetof(struct s2, c2));
    printf("%d ", offsetof(struct s2, i));
```

结果就是c1，c2，i 的偏移量分别为0 1 4



偏移量分别为0 1 4

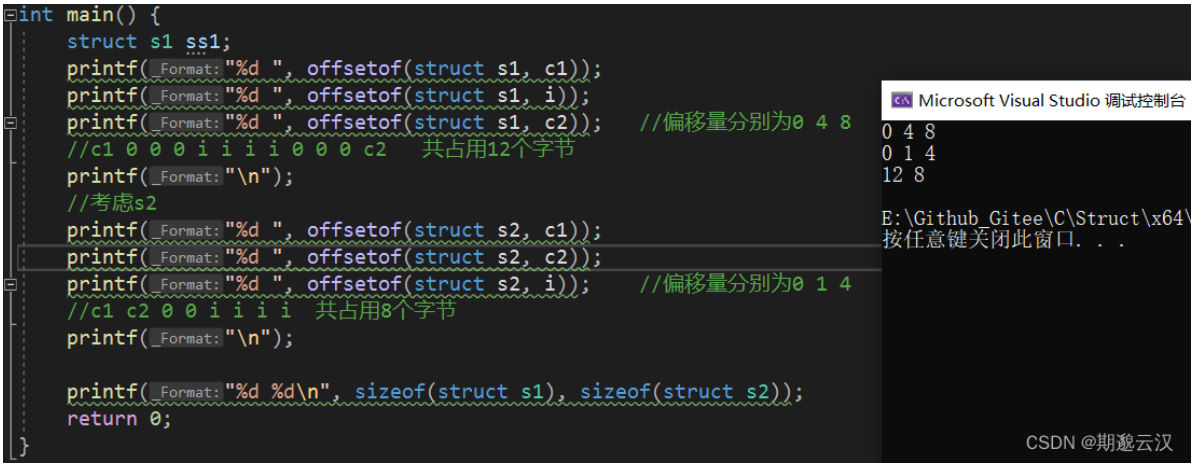
`c1 c2 0 0 i i i i` 共占用8个字节

> **分析完后完整代码如下**：

```
struct s1
{
    char c1;
    int i;
    char c2;
};

struct s2 {
    char c1;
    char c2;
    int i;
};
//结构体内存对齐
//offset这个东西，type，member。查看内存偏移量
#include <stddef.h>
int main() {
    struct s1 ss1;
    printf("%d ", offsetof(struct s1, c1));
    printf("%d ", offsetof(struct s1, i));
    printf("%d ", offsetof(struct s1, c2));    //偏移量分别为0 4 8
    //c1 0 0 0 i i i i 0 0 0 c2    共占用12个字节
    printf("\n");
    //考虑s2
    printf("%d ", offsetof(struct s2, c1));
    printf("%d ", offsetof(struct s2, c2));
    printf("%d ", offsetof(struct s2, i));    //偏移量分别为0 1 4
    //c1 c2 0 0 i i i i  共占用8个字节
    printf("\n");

    printf("%d %d\n", sizeof(struct s1), sizeof(struct s2));
    return 0;
}
```



总结对齐规则

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***
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?

大部分的参考资料如下：

- *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。

- 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。

总结

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。`

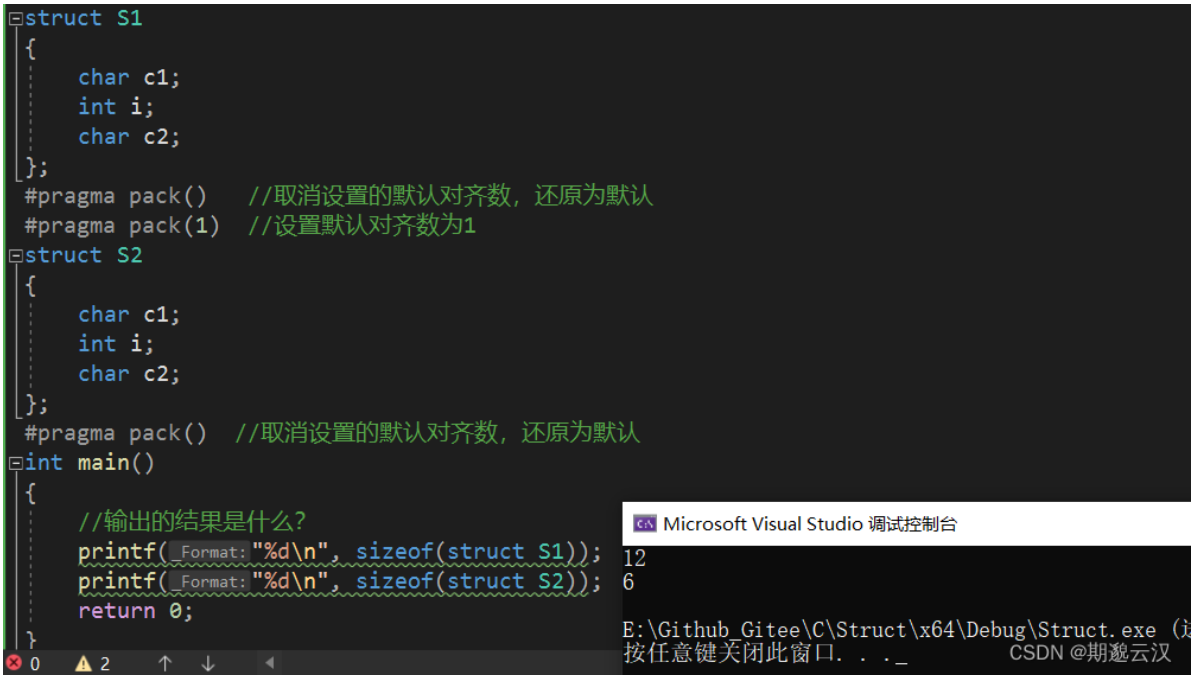
```
struct s1
{
 char c1;
 int i;
 char c2;
};
struct s2
{
 char c1;
 char c2;
 int i;
};
```

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1
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。