第五章：内存、字符串、结构体、共用体

程序中内存从哪来

- 内存本质是硬件，由OS统一管理，程序运行时需要内存存一些临时变量
- **三种内存申请方式：栈、堆、数据段**
  - **栈内存：普通局部变量、自动**
    - **自动分配、自动回收（操作系统维护一个栈指针，自动移动），不需要程序员干预**
    - **栈空间反复使用，大小固定**
    - **栈内存是脏内存，不会擦掉，因此定义局部变量后一定要初始化（不初始化变量的值是随机的）**
    - **不应当在函数中返回临时变量的地址**
    - **栈会溢出：分配空间超过栈大小、不停递归**
  - **堆内存：独立、手动**
    - **每个进程有自己的栈内存（小块），如果进程需要大空间，则从堆内存中申请**
    - **堆内存反复使用，脏内存（申请空间后要初始化）**
    - **手动申请释放：malloc和free**
      - **使用步骤：申请和绑定→if(NULL==p)检验是否成功→使用申请的内存→释放，释放完写上p=NULL（防止野指针）**
      - **malloc返回void*类型（malloc不确定分配的内存用来存什么类型的数据）指针，实质上是申请的内存空间的首地址，申请失败返回NULL**
      - **malloc按块分配，不能分配任意空间，gcc中按照16字节为单位分配，即malloc(2)和malloc(4)实际上分配同样大小的空间，多的只是被浪费**
      - **malloc（0）返回内容不确定（可能返回NULL或者分配一定空间返回指针），标准C没有规定，由各malloc函数库实现者确定**
      - **malloc内存分配是连续的**
    - **malloc的风险**

        - **在free之前给指针另外赋值，会导致内存丢失（程序吃内存/内存泄漏），直到当前程序结束，操作系统才会回收**
        - **free释放后不应该再使用，不代表不能使用，但堆管理器可能会把这段内存分给其它进程**
        - **malloc申请16字节的空间，但第200字节可能依旧可以访问（C语言不严密）**
    - **数据段：全局变量、静态局部变量**
        - **编译器编译时，将程序分为数据段、代码段、bss段等**
            - **数据段 (.data)：数据区/静态数据区/静态区，存程序中显示初始化非0的全局变量、静态局部变量（局部变量不是程序的数据、是函数的数据，隶属于栈，全局变量隶属于数据段）**
            - **代码段.text：函数的堆叠，表示动作**
            - **bss段 (ZI段zero initial)：存放被初始化为0的数据或者没有初始化的全局变量，本质上也是一种数据段**
        - **有些特殊数据会被分配到代码段，不能被改变**
            - **字符串"linux"**
            - **某些情况下的const常量，const型常量的两种实现方式：**
                - **将const修饰变量放到代码段（常见于各种单片机）**
                - **const修饰变量放在数据段，由编译器检查（gcc中使用）**
- 三种申请方式都可以存储数据，如何选择？
    - 函数内部临时用→栈，局部变量
    - 变量只在程序一个阶段有用→堆
    - 变量存在程序始终→数据段

字符串

- C语言没有string类型，通过字符指针间接实现，指针指向头、固定尾部（用'\0'结尾，'\0'是ASCII编码为0的字符，为NUL/null/空字符，不同于'0'，'0'=48，'\0'=0）、地址相连
    - C语言中不可能存在一个包含'\0'的字符串，'\0'不属于字符串
    - char*p="linux"："linux"分配在代码段，占6个字节，实际上耗费了10字节（4字节存指向字符串的指针）
- **存储多个连续字符的两种方式：字符数组、字符指针（两种不同的存储方式）**

- **字符数组：char a[]="linux"完全等同于char a[]={'l','i','n','u','x','\0'}，右值存在于编译器中，编译器用其初始化数组后丢弃，内存中不保存 "linux"**
- **字符指针：函数中char*p="linux"，定义一个字符指针，占4字节，分配在栈上，还定义了一个字符串"linux"，分配在代码段，然后把 'l' 的首地址赋值给p**
- **总结**

	字符数组	字符指针（优选）
本质	数组	指针
字符串存储位置	栈	栈、数据段、堆
占用空间	字符串长度+1（未定义数组大小）；数组容器大小（定义数组大小）	永远只占4字节

```
//示例：数据分别存在栈、数据段、堆上

char b[5];

int main(void)
{
  //存到栈上（字符数组、字符指针）
  char a[7];
  char *p=a;

  //存到数据段
  char*p=b;

  //存到堆上
  char*p=(char*)malloc(5);
}

```

- sizeof和strlen
  - **sizeof是关键字/运算符，返回 变量/数据 占用的内存字节数，包括'\0'**
  - **strlen是C语言库函数，size_t strlen(const char*s)，返回字符串的长度，不包含'\0'，从开头往后数，一直到第一个 '\0'**
  - 为什么要有sizeof?
    - 不同平台下各数据类型占的字节数不同（int在32位系统4字节，16位系统2字节）

- 自定义类型UDT（数组、结构体、函数类型等）的存在

```
//结构体UDT（存在对齐问题）
struct A
{
  char a;
  int b;
};
struct A a;
sizeof(a);//值为8

//数组UDT
 char str[]="hello"
 sizeof(str)//值为6,sizeof(数组名)返回数组大小
 sizeof(str[0])//值为1
 strlen(str)//值为5

 //指针UDT
 char str[]="hello"
 char *p=str
 sizeof(p)//值为4
 sizeof(p[0])//值为1
 strlen(str)//值为5
```

- 字符串、sizeof、strlen

```
//1、字符数组定义方式

char a[5]="lin";
sizeof(a);//5
strlen(a);//3

char a[5]={1,2};
sizeof(a);//5
strlen(a);//2

char a[5];//局部变量未初始化，里面的值随机
sizeof(a);//5
strlen(a);//5

char a[5]={0};//0代表'\0'
sizeof(a);//5
strlen(a);//0，第一个元素就是'\0'

```

```
char a[5]="windows";
sizeof(a);//5
strlen(a);//5,越界初始化会直接截掉

//2、字符指针定义方式
char*p="linux";
sizeof(p);//4，测的永远是指针，这种定义方式必须用strlen
strlen(p);//5，strlen(指针名)得到指向字符串大小
```

结构体

- 定义：先定义类型再定义变量，或者同时进行（若同时进行，后面依旧可以定义变量，没有副作用）

```
//同时定义类型和变量，后面可以直接用

struct student
{
  char name[20];
  int age;
}s2;

int main(void)
{
  struct student s2;//依旧可以定义变量，没有副作用
  s2.age=21;
}
```

  - 一般都是在函数外部定义结构体，在函数内部也可以，但几乎没人这么用

- 由来：结构体从数组发展而来，数组是最简单的数据结构，为弥补数组缺陷（必须明确给出大小，且不可修改；元素类型必须一致）
- 结构体变量中的元素访问：使用 .（变量访问）或者 →（指针访问）访问，两者实质一样，编译器都转成指针

```
struct myStruct
{
  int a;
  double b;
  char c;
};

```

```c
int main(void)
{
  struct myStruct s1;
  s1.a=12;//内部原理：int*p=(int *)&s1; *p=12;
  s1.b=4.4;//double*p=(double*)(&s1+4); *p=4.4;
  //这里加4因为int占四个字节，编译器会自动计算偏移量
  s1.c='a';//char*p=(char*)(&s1+12); *p='a';

  //指向结构体中元素b的指针
  double *p=(double *)((int*)&s1+4);//方法1
  double *p=&(s1.b);//方法2
  printf("%f",*p);//4.4
}
```

- 对齐访问：对齐排布提高访问效率，空间换性能（实际编程不需要考虑元素对齐，编译器会自动处理）
  - 32位编译器，默认4字节对齐（结构体大小为4的倍数且考虑每个元素自身对齐规则）
  - gcc支持但不推荐的对齐指令

```c
//方法一：

#pragma pack(n)  // n=1/2/4/8，结构体内部每个元素n字节对齐，放在希望对齐的结构体之前
#pragma pack()  // 取消对齐，放在希望对齐的结构体之后
//规定了一个区间，C语言广泛支持，gcc不推荐


//示例：

#pragma pack(1)
struct student
{
    int a;
    char b;
    short c;
};
#pragma pack()
```

  - gcc推荐的对齐指令

```c
//方法二：

```

```
__attribute__((packed))//取消默认的四字节对齐规则
__attribute__((aligned(n)))//整个结构体变量整体n字节对齐，而不是结构体内部各元素n字节对齐

//示例：

struct student
{
    int a;
    char b;
    short c;
}__attribute__((packed));
//7

typedef struct teacher
{
    int a;
    char b;
    short c;
}__attribute__((aligned(8))) t1;
//8
```

offsetof宏和container_of宏（DJI）

- offsetof宏：为了得到结构体中某元素相对于结构体首地址的偏移量（实际上编译器自动计算）
- container_of宏：已知结构体中某元素指针，反推结构体变量指针，继而得到结构体中其它元素的指针

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE*)0)->MEMBER))

详细解析：
//(TYPE*)0, 0地址转换成指针，指针指向结构体变量，实际上该结构体变量可能不存在，从0地址开始虚拟出来一个结构体变量
//((TYPE*)0)->MEMBER，通过指针访问MEMBER元素
//&((TYPE*)0)->MEMBER)，得到MEMBER的地址，由于结构体从0开始，相当于偏移

#define container_of(ptr,TYPE,MEMBER) \
        ({const typeof( ((TYPE*)0)->MEMBER) * __mptr = (ptr); \
        (TYPE *)((char*)__mptr-offsetof(TYPE,MEMBER));})

详细解析：
//ptr是指向结构体中元素的指针，宏返回的是指向整个结构体变量的指针，即为TYPE*类型
//__mptr是typeof(((TYPE*)0)->MEMBER)类型的指针，typeof(a)表示由变量名a得到a的类型，这里得到MEMBER的数据类型，ptr是某元素指针，传进来的时候类型已经丢了，用这种方式重新获取
```

```c
//__mptr减偏移量得到整个结构体的指针，再强制类型转换成 TYPE*

struct student
{
  char a;//偏移0
  int b;//偏移4
  short c;//偏移8
};

int main(void)
{
  struct student s1;
  struct student *pS;
  short *p=&(s1.c);

  int offsetofa=offsetof(struct student,a);//0
  //TYPE是结构体类型,MEMBER是结构体中元素

  pS=container_of(p,struct student,c);
}
```

- 掌握要求层级：会用→看到会解析→会写

共用体/联合体

- 定义、元素使用与struct一样，但各元素共用同一内存空间（同一内存空间多种解析方式）
- 占用内存取决于占用内存最大的元素，不存在内存对齐问题
- 共用体中元素都是从低地址开始访问
- 使用场景：通信中不知道对方会发一个什么类型的数据包过来

```c
union pack
{
    int a;
    char b;
    float c;
};

int main(){
    union pack data;
    data.a=65533;
    printf("%f\n",data.c);//使用float方式解析

    int a=65533;
```

```
    printf("%f\n",*((float*)&a))//使用指针代替共用体，C语言中可以没有共用体，但共用体更好理解
}
```

- 大小端模式（big endian、little endian）
  - 大端高位低地址，小端高位高地址；存储读取要按照同样的方式
  - 有些CPU公司用大端（51单片机），有些用小端（ARM），大部分用小端，写代码时需要自己测试

```
//方法一：union测试大小端
union myunion
{
  int a;
  char b;
};
char is_little_endian(void)//返回1则为小端
{
  union myunion u1;
  u1.a=1;
  return u1.b;
}//共用体中元素都是从低地址开始访问

int main(){
    printf("%d\n",is_little_endian());
}

//方法二：指针测试大小端（本质）
int main(void)
{
  int a=1;//小端存，1在低地址
  char b=*((char*)(&a));//将a强行截断，只取低位，为1则为小端
}
```

  - 不可行的测试方式

```
//1、位与方式
int a=1;
char b=a&0x01;//预期b为1则为小端，实际无法测试，大小端都得1
//原因：C语言对位与运算做了适配,一定是高字节对高字节，与内存无关

//2、移位方式
int a,b;
```

```
a=1;
b=a>>1;//预期b为全0说明是小端模式，实际无法测试，大小端都为0
//原因：右移运算永远移出低字节

//3、强制类型转换
int a;
char b;
a=1;
b=(char)a;//预期强制类型转换截断低位。实际不可以
//原因：C语言做了移植

```

- 通信系统中的大小端：一般，先发低位→小端，先发高位→大端，实际上通信协议会明确定义（使用或自己定义通信协议都要事先明确）

枚举

- 实际上是个常量集，定义了一些符号，每个符号和一个int常量绑定
- 不定义从0开始计数，用户定义了则依次计数，也可全部自己定义
- 宏定义和枚举都是为了符号代替数字，大部分可以替换，区别：枚举编译时计算、一次定义一批、可实现分类、可自动依次计数；宏定义预编译时计算、一次定义一个

```
enum return_value//定义枚举类型
{
  ERROR,
  RIGHT,
};//枚举值是全局的，可以直接单独用，如果两个枚举变量有相同的符号会报错


#define ERROR 0
#define RIGHT 1
```