C语言12.16日 结构体, 函数

结构体

有的时候我们为了描述一个对象需要很多个属性, 例如描述一个人, 他有年龄, 性别, 姓名, 身份证ID号等等属性, 当我们需要把这些属性聚合起来, 结构体就出现了

```
struct Human {
  unsigned int age, hight;
  char sexual;
  char name[10];
  int ID[20];
};
struct Human man;
```

我们称结构体中的每一个变量为成员变量或者数据成员或者字段, 成员变量可以是基本数据类型(int, float, char等), 也可以是其他结构体类型, 或者指针类型, 但是 成员变量**不可以是自身结构体类型的实例**

```
struct ELIEN {
    struct Human man;
    struct ELIEN e;
}; // 这样做是错误的 不允许使用不完整的类型
```

定义结构体

结构体定义由关键字 `struct` 和结构体名字组组成

`struct` 语句定义了一个包含一个或多个成员的新的**用户自定**数据类型, struct语句的格式如下:

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

tag 是结构体标签。

member-list 是标准的变量定义，比如 int i; 或者 float f;，或者其他有效的变量定义。

variable-list 结构变量，定义在结构的末尾，最后一个分号之前，您可以指定零个, 一个或多个结构变量。下面是声明 Book 结构的方式:

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book; // book 是结构体struct Books 的一个实例
```

在一般情况下，tag、member-list、variable-list 这 3 部分至少要出现 2 个。以下为实例：

```
//此声明声明了拥有3个成员的结构体，分别为整型的a，字符型的b和双精度的c
//同时又声明了结构体变量s1
//这个结构体并没有标明其标签
struct
{
    int a;
    char b;
    double c;
} s1; // s1是一个匿名的结构体变量

//此声明声明了拥有3个成员的结构体，分别为整型的a，字符型的b和双精度的c
//结构体的标签被命名为SIMPLE,没有声明变量
struct SIMPLE
{
    int a;
    char b;
    double c;
};
//用SIMPLE标签的结构体，另外声明了变量t1、t2、t3
struct SIMPLE t1, t2[20], *t3;

//也可以用typedef创建新类型
typedef struct
{
    int a;
    char b;
    double c;
} Simple2;
//现在可以用Simple2作为类型声明新的结构体变量
Simple2 u1, u2[20], *u3;
```

在上面的声明中，第一个和第二声明被编译器当作两个完全不同的类型，即使他们的成员列表是一样的，如果令 t3=&s1，则是非法的。

结构体的成员可以包含**其他结构体**，也可以包含**指向自己结构体类型的指针**，但是不可以包含自己结构体类型的实例

```
//此结构体的声明包含了其他的结构体
struct COMPLEX
{
    char string[100];
    struct SIMPLE a;
};
```

如果两个结构体互相包含，则需要对其中一个结构体进行不完整声明，如下所示：

```
struct B;     //对结构体B进行不完整声明

//结构体A中包含指向结构体B的指针
struct A
{
    struct B *partner;
    //other members;
};

//结构体B中包含指向结构体A的指针，在A声明完后，B也随之进行声明
struct B
{
    struct A *partner;
    //other members;
};
```

结构体变量的初始化

和其它类型变量一样，对结构体变量可以在定义时指定初始值

```
#include <stdio.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};
struct Books book = {"C 语言", "RUNOOB", "编程语言", 1234}; // initializer_list

void init_Books(struct Books* pbooks, char* title, char* author, char* subject, int id)
{
```

```
    pbooks->title = title;
    pbooks->author = author;
    pbooks->subject = subject;
    pbooks->book_id = id;
}

struct Books book2;
init_Books(&book2, "C 语言", "RUNOOB", "编程语言", 1234)

int main()
{
    printf("title : %s\nauthor: %s\nsubject: %s\nbook_id: %d\n", book.title,
book.author, book.subject, book.book_id);
}
```

访问结构成员

为了访问结构的成员，我们使用**成员访问运算符（.）**。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号

```
#include <stdio.h>
#include <string.h>

struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        /* 声明 Book1，类型为 Books */
   struct Books Book2;        /* 声明 Book2，类型为 Books */

   /* Book1 详述 */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* Book2 详述 */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "Zara Ali");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
```

```
    Book2.book_id = 6495700;

    /* 输出 Book1 信息 */
    printf( "Book 1 title : %s\n", Book1.title);
    printf( "Book 1 author : %s\n", Book1.author);
    printf( "Book 1 subject : %s\n", Book1.subject);
    printf( "Book 1 book_id : %d\n", Book1.book_id);

    /* 输出 Book2 信息 */
    printf( "Book 2 title : %s\n", Book2.title);
    printf( "Book 2 author : %s\n", Book2.author);
    printf( "Book 2 subject : %s\n", Book2.subject);
    printf( "Book 2 book_id : %d\n", Book2.book_id);

    return 0;
}
```

指向结构的指针

可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似，如下所示：

```
struct Books* p_books = &Book1;
```

通过指针访问某字段(结构体成员)有两种方式:

`(*p).field` 或 `p->field`

```
p_books→title = ..;
(*p_books).title;
```

结构体大小的计算

我们可以使用 sizeof 运算符来计算结构体的大小，sizeof 返回的是给定类型或变量的字节大小。

对于结构体，sizeof 将返回结构体的总字节数，包括所有成员变量的大小以及可能的填充字节。

```
#include <stdio.h>

struct Person {
    char name[20];
    int age;
    float height;
};
```

```c
int main() {
    struct Person person;
    printf("结构体 Person 大小为: %zu 字节\n", sizeof(person));
    return 0;
}
```

函数

函数的基本形式

`ReturnType FunctionName(ArgumentsList) { Body }`

`ReturnType` 和 `ArgumentsList` 可以是你所知道的任何类型, void, int, float struct, union...

但 `ReturnType` 只有一个, 而 `Arguments` 可以有很多, 也可以没有

```c
int test1() {
    return 1;
}
```

函数的参数

为什么函数要有参数?

因为函数内部作为一个**独立, 封闭的空间**, 在计算, 执行的过程中**不可避免地**需要参考外部的信息, 例如:

$$f(x) = x^2 + 2x + 1$$

其中, 我们公式中固定的参数是1, 2, 1, 但不确定的是因变量x

```c
double f(double x) {
    double a = 1.0;
    double b  = 2;
    double c = 1;
```

```
    return a * x * x + b * x + c;
}
```

这也意味着, 我们在实现一个函数的过程中, 只需要参考这个独立, 封闭的空间和小部分我们需要的外部信息(参数列表), 函数内部的执行与外部大部分的信息是无关的, 这降低了我们的思考复杂度.

函数参数的传递

函数参数的传递是**严格按顺序的**, 但是**不是严格按类型的**, 如果调用方调用函数时, 传入的参数与函数所期待的参数类型不同, 会发生**隐式类型转换**, 除非无法进行从源类型到目标类型的类型转换.

(char, unsigned char, short, unsigned short, int, float , double, long long, unsigned longlong)

在大多数情况下值域范围内是可以互相进行类型转换的(*Very Important*)

```
int example2(int i) {
    printf("%d", i);
}
int main() {
    example2(0.5f);
    return 0;
}
```

有没有可以阻止隐式类型转换的办法?

好消息: 有的 --explicit关键字

坏消息: C++特供, C语言不能使用

函数的返回值

一个函数内部可以有多个返回语句, 但是函数总会在**执行流程中遇到的第一个返回语句时**返回调用处(即函数结束并返回结果)

```
int example3(int i) {
    return i;
    return 0;
    return 1;
    int  i  = 0;
    ,,,
}
```

```
int example4(int i) {
    if (i > 0)
        return 1;
    printf("afaf");
    return 0;
}
```

在大多数情况下, 我们有一个原则: 函数的返回语句越少越好(单一出口原则)

```
int example5(int i ) {
    unsigned int n = i;
    for(int j = 0; j < n; ++j) {
        while (n -- ) {
            if (n < 0) break;
            if( ,,, ) break;
        }
    if (... ) break;
    }
    return 0;
}
```

返回值为非void类型的函数**必须显式地**写明返回语句 `return` , 如果实际上的返回值的类型**不是所声明类型**, 那么会经历一次从**实际类型到所声明的返回类型的隐式类型转换:** (*Very Important*)

```
int example() {
    return 1.5f;
}

int main() {
    printf("%d", example())
    return 0;
}
```

如果一个函数指明了返回类型(非void类型)但是没有**显式**地写明返回语句, 或者函数有**某个出口**没有显式写明返回语句的

返回值是上一个变量eax寄存器里的值(危险行为, 因为eax寄存器并不一定保存适合返回类型的值)

函数没有显示写明返回语句的情况:

```
int example7(){
    int a = 5*6;
    return a;
}
```

函数的某个出口没有显式写明返回语句的情况:

```
int example8(int i ) {
    unsigned int n = i;
    for(int j = 0; j < n; ++j) {
        while (n -- ) {
            if (n < 0) return n ;
            if( ,,, ) break;
        }
    if (... ) break;
    }

}
```

动态内存管理

需求, 一个可以支持动态扩容的数组容器, 满足一下操作: 增, 删, 改, 查

```
#include <stdio.h>
#include <stdlib.h>

typedef float T;

typedef struct{
    size_t size;
    size_t capacity;
    T* arr;
}Array;

void init_Array(Array* pArray, size_t capacity) {
    pArray->size = 0;
    pArray->capacity = capacity;
    pArray->arr = (T*)malloc(sizeof(T) * capacity);
}

void resize(Array* pArray, size_t new_size) {
    size_t size = pArray->size;
    size_t oldcapacity = pArray->capacity;
    if (new_size < size) {
```

```
        return;
    }
    T* oldarr = pArray->arr;
    pArray->arr = (T*)malloc(sizeof(T) * new_size);
    pArray->capacity = new_size;
    for (int i = 0; i < size; ++i) {
        pArray->arr[i] = oldarr[i];
    } // memcpy(pArray->arr, oldarr, sizeof(T) * size);
    free(oldarr);
}

#include <assert.h>


void append(Array* pArray, T value) {
    size_t size = pArray->size;
    size_t capacity = pArray->capacity;
    if (size == capacity) {
        resize(pArray, 2 * capacity);
    }
    size = pArray->size;
    capacity = pArray->capacity;
    assert(size < capacity);

    pArray->arr[pArray->size++] = value;
}

void delete(Array* pArray, size_t pos) {
    size_t size = pArray->size;
    size_t capacity = pArray->capacity;
    if (pos >= size) {
        // printf(...);
        return;
    }
    assert(pos < size);
    T* arr = pArray->arr;
    for (int i = pos + 1; i < size; ++i) {
        arr[i - 1] = arr[i];
    }
    --pArray->size;
    size = pArray->size;
    capacity = pArray->capacity;
    if (size < capacity / 3) {
        resize(pArray, capacity / 2);
    }
}

int main() {
```

```
    Array arr;
    init_Array(&arr, 10);
    for (int i = 0; i < 100; ++i) {
        append(&arr, i);
    }
    printf("size: %lld, capacity: %lld \n", arr.size, arr.capacity);
    for (int i = 0; i < 1000; ++i) {
        append(&arr, i);
    }
    printf("size: %lld, capacity: %lld\n", arr.size, arr.capacity);
    for (int i = 1000 + 100; i >= 0; --i) {
        delete(&arr, i);
    }
    printf("size: %lld, capacity: %lld\n", arr.size, arr.capacity);
    free(arr.arr);
}
```

练习

20. 有效的括号 - 力扣（LeetCode）

26. Remove Duplicates from Sorted Array - 力扣（LeetCode）

17. 从尾到头打印链表 - AcWing题库

21. 合并两个有序链表 - 力扣（LeetCode）

16. 替换空格 - AcWing题库