C语言学习笔记

C语言结构体大小计算(★★★)

1. 除结构体的第一个成员外，其他所有的成员的地址相对于结构体地址(即它首个成员的地址)的偏移量必须为实际对齐单位或自身大小的整数倍(取两者中小的那个)
2. 结构体的整体大小必须为实际对齐单位的整数倍

实际对齐单位

1. 结构体最大成员(基本数据类型变量)
2. 预编译指令#pragma pack(n)手动设置　　n--只能填1 2 4 8 16

普通结构体大小计算(6,7重点查看)

```
//类型1
struct s  {//总共大小12
    int a;      //0x00
    float b;    //0x04
    char c;     //0x09
};
//类型2
struct s{//12
    char a;
    float b;
    char c;
};
//类型3
struct s//12
{
    char a;
    int b;
    char c;
};
//类型4
struct s//16
{
    int a;
    int b;
    double c;
};
//类型5
struct s//24
{
    int a;
    double c;
    int b;
};
//类型6
struct s//8最重要
{
    char a;//0x00
    short b;//0x02
```

```
    int c;//0x04
};
//类型7
struct s//24最重要
{
    char a;//0x00
    float c;//0x04
    short b;//0x08

    double d;//0x10
};
//类型8
typedef struct book {//152
    char   name[50];
    char   author[20];
    char   publish[30];
    char   isbn[20];
    char   local[20];
    int    amount;
    float  price;

    struct book* next;
} BOOK;
```

结构体位域操作大小计算

```
struct s //8
{
    int b : 8;
    char a : 2;
    char c : 2;
};
struct s //8
{
    int b : 8;
    char a : 2;
    short c : 2;
};
struct s //12
{
    char a : 2;
    int b : 8;
    char c : 2;
};

struct s//3
{
    char a : 2;
    char b : 8;
    char c : 2;
};

struct s //2
{
    char a : 2;
    char c : 2;
    char b : 8;
```

```
};
```

结构体套结构体大小计算(将内部套的结构体只看单独变量)

```
struct xx//24   struct s//24
{   {
        char a;         int a;
        float c;            double c;
        short b;            int b;
};          xx d;
    };
```

结构体中存在Union 联合体时,只看联合体内最大类型变量

C语言结构体内指针(★★★)

```
//错误演示
//当你初始化的时候你将结构体中name指针变量赋值了一个常量地址
//当你使用Strcpy(n.name,"tim");时将会更改常量区内容,将会报错.

struct name{
    char *name;
    int age;
};

int main(void){
    struct name n = {"tom",60};
    strcpy(n.name,"mike");
}
//正确演示
//malloc是从heap堆中申请的变量,注意free的先后
struct name{
    char *name;
    int age;
};

int main(void){
struct name n ;
    n.name = (char*)malloc(sizeof(char)*10);
    strcpy(n.name,"tom");
    strcpy(n.name,"mike");
    free(n.name);
}

int main(void){
    struct name* p = (struct name*)malloc(sizeof(struct name)) ;
    p->name = (char*)malloc(sizeof(char)*10);
    strcpy(p->name,"tom");
    strcpy(p->name,"mike");
    free(p->name);
    free(p);
}
```

C语言使用结构还是指向结构的指针和const(★★)

在定义一个结构体有关的函数,到底是使用结构体,还是结构体指针?
指针作为参数,只需要传递一个地址,所有代码效率高
结论就是当一个结构作为函数参数的时候,尽量使用指针,而不是使用结构体变量,这样代码的效率很高

注意事项:
当不需要修改结构体变量内部数据时最好加入const
来防止函数内部结构体变量被修改

```
void printf_student(const struct student *s){
//一般来讲,不要把结构体变量做为函数的参数传入
//当只需要进行读数据时最好使用,const关键字
}
```

C语言union(联合体)(★)

union变量大小取决与内部定义的最大变量类型

```
//&u.a,&u.b使用的地址都是相同的
 union u{
 int a;
 char b;
};
union u{
unsigned  char a;
char b;
};
//u.a = 128;
//显示a = 128,b=-128
```

注意事项:
如果联合体中有指针成员,那么一定要使用完这个指针,并且free之后才能使用联合内其他成员

C语言enum(枚举)(★)

```
#define red 1
#define red 1
#define red 1

enum a{
red,black,yellow = 9,green
};
```

注意事项:
枚举是常量,值不能修改

C语言typedef在函数中的使用(★★★)

https://www.bilibili.com/video/av34711641?p=5
里面讲函数与typedef的共同使用

** #define与typedef的区别**

1. 与#define不同,typedef 仅限于数据类型,而不是表达式或具体的值

2. typedef是编译器处理的,而不是预编译指令
3. typedef比#define更加灵活,直接看typedef好像没有什么用处,使用BYTE定义一个unsigned char.使用typedef可以增加程序的可移植性,更方便理解

```

typedef char* (*STRCAT)(char *,char*);

char *test(STRCAT P,char *s1,char* s2){
    return p(s1,s2);
}
//当声明函数型指针数组时
char *(*p[10])(char *,char*);//复杂难懂
STRCAT arry[10]; //更加方便,也更容易理解

//当函数输出时是函数型指针时,采用typedef
STRCAT get_mystrcat(){
    return mystrcat;
}

/*错误,编译器无法通过
char* (*)(char *,char*) get_mystrcat(){
    return mystrcat;
}*/

//当我们需出如下类型时,使用typedef
int ** test(void){
    int * aa[10];
    return aa;
}

typedef int  (*__a)[10];
__a test1(void){
    int a[10];
    //int (*aa)[10] = &a;
    __a aa = &a;
    return aa;
}
```

C语言指针类型对应与函数解析(★★★)

指针类型对应表

int *P1	**a[10]**	**&a**	
int **P2	&a	&P1	
int (*p3)[10]	&a	b[20][10]	对应二维数组
当返回值	类型不会写时	使用typedef	
typedef int(*__a)[10] ;	__a test1(int* a, int* b)	返回int(*)[10]	

函数解析方法

右左法则：首先从最里面的圆括号看起，然后往右看，再往左看。每当遇到圆括号时，就应该掉转阅读方向。一旦解析完圆括号里面所有的东西，就跳出圆括号。重复这个过程直到整个声明解析完毕。笔者要对这个法则进行一个小小的修正，应该是从未定义的标识符开始阅读，而不是从括号读起，之所以是未定义的标识符，是因为一个声明里面可能有多个标识符，但未定义的标识符只会有一个。

函数解析实例

现在通过一些例子来讨论右左法则的应用，先从最简单的开始，逐步加深：

int (*func)(int *p);*

*首先找到那个未定义的标识符，就是func，它的外面有一对圆括号，而且左边是一个*号，这说明func是一个指针，然后跳出这个圆括号，先看右边，也是一个圆括号，这说明(*func)是一个函数，而func是一个指向这类函数的指针，就是一个函数指针，这类函数具有int*类型的形参，返回值类型是int。

int (*func)(int *p, int (*f)(int));*

*func被一对括号包含，且左边有一个*号，说明func是一个指针，跳出括号，右边也有个括号，那么func是一个指向函数的指针，这类函数具有int *和int ()*(int)*这样的形参，返回值为int类型。再来看一看func的形参int (*f)(int)，类似前面的解释，f也是一个函数指针，指向的函数具有int*类型的形参，返回值为int。

int (*func[5])(int *p);*

func右边是一个[]运算符，说明func是一个具有5个元素的数组，func的左边有一个，说明func的元素是指针，要注意这里的*不是修饰func的，而是修饰func[5]的，原因是[]运算符优先级比*高，func先跟[]结合，因此*修饰的是func[5]。跳出这个括号，看右边，也是一对圆括号，说明func数组的元素是函数类型的指针，它所指向的函数具有int*类型的形参，返回值类型为int。

int ((*func)[5])(int *p);*

func被一个圆括号包含，左边又有一个，那么func是一个指针，跳出括号，右边是一个[]运算符号，说明func是一个指向数组的指针，现在往左看，左边有一个*号，说明这个数组的元素是指针，再跳出括号，右边又有一个括号，说明这个数组的元素是指向函数的指针。总结一下，就是：func是一个指向数组的指针，这个数组的元素是函数指针，这些指针指向具有int*形参，返回值为int类型的函数。

int ((*func)(int *p))[5];*

*func是一个函数指针，这类函数具有int*类型的形参，返回值是指向数组的指针，所指向的数组的元素是具有5个int元素的数组。

要注意有些复杂指针声明是非法的，例如：

int func(void) [5];

func是一个返回值为具有5个int元素的数组的函数。但C语言的函数返回值不能为数组，这是因为如果允许函数返回值为数组，那么接收这个数组的内容的东西，也必须是一个数组，但C语言的数组名是一个右值，它不能作为左值来接收另一个数组，因此函数返回值不能为数组。

int func5;

func是一个具有5个元素的数组，这个数组的元素都是函数。这也是非法的，因为数组的元素除了类型必须一样外，每个元素所占用的内存空间也必须相同，显然函数是无法达到这个要求的，即使函数的类型一样，但函数所占用的空间通常是不相同的。

C++学习笔记

C++类和对象

C++中引用的概念

引用的语法 类型 + &别名 = 名称

类如 int & INC = _inc;

引用实际上是const 指针,不能进行第二次的赋值

当函数返回引用时可以是可修改的左值

当在函数 func(const int & arc)时无法将arc所指向的地址内容进行修改

C++函数的高级使用

函数可以设置默认参数
格式:返回值类型 函数名 (形参 = 默认值){}
例如:
int func(int a, int b, int c =10){}
注意事项
1~如果某个位置已经有了默认参数,那么从这个位置以后,从左到右都必须有默认值
例如:
int func(int a, int b, int c =10){}
int func(int a, int b = 10, int c){}

2~如果函数的声明有默认参数,函数的实现就不能有默认参数
(声明和实现只能有一个默认参数)
例如
函数声明:int func(int a, int b, int c =10);
函数实现:int func(int a, int b, int c){}
函数实现:int func(int a, int b, int c =10){}

** C++类和对象**

类的结构

Class _class{//类
Public:
Private:

//create属性
//create行为
};

实例化过程

_class temp;//对象

/*访问权限(3种-public-private-protected)
protected :保护权限 成员 类内可以访问,类外不可以访问->子类可以访问父类保护内容
private :私有权限 成员 类内可以访问,类外不可以访问->子类不可以访问父类私有内容
public :公共权限 成员 类内可以访问,类外可以访问
*/

class与struct的区别
class默认private
struct默认public

在类中,可以让另一个类,作为本类中的成员

** 函数的分类及调用**

二种分类方式

1. 按参数分为:有参构造和无参构造
2. 按照类型分:普通构造和拷贝构造

三种调用方式

```
    //1.  括号法
    //假设 Person 为一个class
    Person p1;       //默认构造函数
    Person p2(10)    //有参构造函数
    Person p3(p2)    //拷贝构造函数
    /*注意事项
    调用默认构造函数时不能加()
    例如:Person p1();
    因为这行代码,编译器会认为是一个函数的声明,不会认为是在创建对象*/
```

```
    //2.  显示法
    Person p1;               //默认构造函数
    Person p2 = Person(10); //有参构造函数
    Person p3 = Person(p2); //拷贝构造函数
    Person(10);              //匿名对象
                             //特点:当前行执行结束后,系统会立即回收掉匿名对象
    /*注意事项
    不要利用拷贝构造函数初始化匿名对象
    编译器会认为Person(p3) == Person p3;对象声明
    Person(p3);
    */
```

```
    //3.  隐式转换法
    Person p1 = 10; //有参构造函数
    Person p2 = p1; //拷贝构造函数
```

构造函数调用时机

C++中拷贝构造函数调用时机通常有三种情况

1. 使用一个已经创建完毕的对象来初始化一个新的对象
2. 值传递的方式给函数参数传值
3. 以值方式反回局部对象

构造函数调用机制

默认情况下,C++编译器至少给一个类添加三个函数

1. 默认构造函数(无参,函数体为空)
2. 默认析构函数(无参.函数体为空)
3. 默认拷贝构造函数,对属性进行值拷贝

构造函数调用规则如下

- 如果用户定义有参构造函数,C++不在提供默认无参构造,但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数,C++不会再调用其他构造函数

深拷贝与浅拷贝

浅拷贝:简单的赋值拷贝操作

```
//举例-->演示正常
//通过默认拷贝函数可以实现功能
class Person{
```

```
public:
    person(int _Age){
    h_Age = _Age;
    }
public:
    int h_Age;
};

void test1(void){
    Person p1(18);
    person p2(p1);
}
//举例-->演示错误,浅拷贝不正确
//通过默认拷贝函数无法实现重新堆内申请的的空间,只能复制先前的地址
class Person{
public:
    Person(int _Age, int _height){
    h_Age = _Age;
    h_height = new int(_height);
    }
public:
    int h_Age;
    int * h_height;
public:
    ~Person(){
    delete h_height;
    h_height = NULL;
    }
};

void test1(void){
    Person p1(18,180);
    Person p2(p1);
    //会出错
}
```

深拷贝:在堆区重新申请空间,进行拷贝操作

```
//举例-->演示正确,需要自己写拷贝函数
class Person{
public:
    Person(int _Age, int _height){
    h_Age = _Age;
    h_height = new int(_height);
    }
public:
    int h_Age;
    int * h_height;
public:
    person(const Person &p){//&引用
    h_Age = p.h_Age;
    h_height = new int(*(p.h_height));
    }
public:
    ~Person(){
    delete h_height;
    h_height = NULL;
```

```
    }
};

void test1(void){
    Person p1(18,180);
    Person p2(p1);
}
```

初始化列表

语法:

```
    构造函数():属性1(值1),属性2(值2) ... {}
    例子:Person():age(1),heigt(160) {}
        Person(a,b):age(a),heigt(b) {}
```

类对象作为类成员

当其他类对象作为本类成员,构造时先构造类对象,再构造自身,
析构的顺序与构造相反

静态成员

静态成员就是在成员变量和成员函数前加上关键子static,称为静态成员
静态成员分为

- 静态成员变量
    - 所有对象共享同一份数据
    - 在编译段分配内存
    - 类内声明,类外初始化

```
class Person{
public:
    static func(){
        num = 200;
    }
static int num;
}

int Person::num = 0;
```

- 所有对象共享同一份数据
    - 所有对象共享同一个函数
    - 静态成员函数只能访问静态成员变量

访问静态成员有两种方式

1. 通过对象访问

```
    Person p;
    p.func();
```

2. 通过类名访问

```
    Person::func();
    //当将静态变量设置为private时通过
    //person::func();类外无法访问私有静态变量成员函数
```