# C语言

# 一、基础知识

## 1、知识覆盖范围

c/c++ 数据结构 系统编程 接口封装和设计

日志库，配置文件， windows linux IPC库， socket 数据库

windows 项目 linux项目

## 2、内存布局

**数据区：**堆、栈、全局/静态存储区

**全局/静态存储区：**常量区,全局区,静态区

**常量区：**字符串常量区，常变量区

**代码区：**二进制代码，不可寻址

1. **栈区-（先进后出的内存结构）**

由编译器自动分配，存放函数**参数，返回值，局部变量、结构体、指针**程序运行中实时加载释放，都由编译器负责。

**注意：**

函数不要返回局部变量的地址，**栈区内容会被释放，这时的地址是一个野指针**

函数里创建的字符串变量是先放到常量区，然后又复制一份到栈区，函数结束后一样会删除

栈区大小在不同的操作系统中，系统分配给每一个栈区大小空间不同，windows 1M-8M不等

linux 1M-16M

1. **堆区**

容量大，由程序员操作，程序结束（main函数）后由系统回收；

**注意：**

堆空间需要自己手动申请，

连续的内存空间都可以使用下标赋值

int \*p = malloc(sizeof(int) \* 5)

p[0] =100;

音频文件，文本文件，视频文件

如果创建大的数据，应该放到堆区

**（3）全局静态区**

全局静态区，存放全局变量，静态变量，常量。静态区与全局区区别，作用域不同，

static定义文件只能在当前文件使用，全局变量可以在所有文件中使用，

常量区-字符串常量，不能修改。char \*p常量区，可直接通过字符串获取首地址

const修饰全局变量-常量区，不能修改该

const修饰局部变量-栈上，不能直接修改，通过指针修改

## 3、内存管理

**作用域**

int a ; //作用域为{}之间，变量存放在栈中，栈的顺序是从上到下存储，声明不会分配内存空间，定义的时候才会分配内存空间

生命周期

局部变量 int a = 10 函数结束

全局变量 int a = 10 程序结束

静态变量 static int a = 10 程序结束

代码区：

程序指令

数据区：

初始化数据

1. 初始化的全局变量
2. 初始化的静态局部变量
3. 初始化的静态全局变量

未初始化的数据

1. 未初始化的静态局部变量，默认值为0
2. 未初始化的全局变量，默认值为0
3. 未初始化的静态全局变量，默认值0

字符串常量

变量存储类型

Auto自动类

register寄存器类 仅局部变量

static静态类

extern外部类

## 5、计算机中存储的方式

正数反码，补码 源码相同都相同

计算机中数值以补码存储

### 源码，

计算机存储的是二进制数，最高位用于表示数的正负

+0 0000 0000

-0 1000 0000

### 反码：

负数取反（除第一位）

### 补码（用于计算加法，正数加负数）

反码最后一位+1

会出现溢出，系统自动舍去溢出数据

sizeof获取的结果是字节（Byte)

## 基础单位

B-b-M-G

## 8、进制转换

**十进制转二进制(除2反序取余法）**

1987 % 2

**二进制转十进制（权值法）**

第一位：1\*2^0 = 1

第二位：1\*2^1 = 2

第三位：0\*2^2 = 0

第四位：0\*2^3 = 0

... 最后将结果加起来

**八进制/十六进制转二进制使用8241法则**

**八进制转二进制**

一位八进制位转换为3位二进制

**十六进制转二进制**

一位十六进制转换为4位二进制

## 库

已经写好得，成熟得可复用得代码

windows静态库

1. 创建新项目
2. 创建头文件
3. 编写代码-重新编译
4. 将.h .lib文件给其他人调用.h是看有什么函数可调用

动态库调用

1. 隐式调用
2. 使用加载lib

# 二、C语言基础操作

## 1、引入文件

.c源文件。std标准库，io输入输出库

#include <stdio.h> //导入标准输入输出库

#include "good.h" //调用good.h,直接调用即可

#include "..\two\some.h" //调用上一目录的其他文件夹下的头文件

## 2、样例

// 主函数 一个程序有且只有一个主函数

// 函数的返回值类型 int

// argc函数输入的参数个数，

// argv函数输入的参数是什么（文本）

int main(int argc, char \*argv[])

int main(void)

{

printf(“hallo world”);

return 0;

}

//关闭警告-windows

#define \_CRT\_SECURE\_NO\_WARNINGS

## windows隐藏cmd一闪

#include <windows.h>

WinExec(“calc”, SW\_HIDE);

## 7、强制转换

int a =123;

float b = (float)a

# 宏定义

\_\_FILE\_\_

\_\_LINE\_\_

**定义头文件，防止头文件多次加载**

#ifndef \_\_luck\_\_MainMenu\_\_

#define \_\_luck\_\_MainMenu\_\_

// luck项目名称，MainMenu文件名称

#endif

**C语言宏定义**

#ifdef \_\_aplusplus

extern “C” {

#endif

.....

#ifdef \_\_aplusplus

}

#endif

-----------------------

//宏函数-不是真正的函数,宏只是进行了文本替换

//宏函数在一定场景下效率高，

#define MYADD(x,y) ((x)+(y))

int main()

{

MYADD(a,b)

//相当于替换为(a)+(b)

}

# 编译

## 编译过程

头文件不参与编译，每一个.c文件叫做编译单元

编译器独立编译每一个.c文件

extern int a; //指明这个变量是全局变量

***C语言代码编译分为4步***

**预处理：gcc -E a.c -o a.i**

预处理：宏定义，头文件展开，条件编译等，删除注释，不检查语法 （.i)

**编译：gcc -S a.i -o a.s**

编译：检查语法，将预处理后的文件编译成汇编语言(.s)

**汇编：gcc -c a.s -o a.o**

汇编：生成二进制文件(.o)

**链接：gcc a.o -o a.exe**

链接：将需要使用的库链接到可执行文件中

linux平台下更新链接库lld heal.exe

gcc -o file 指定生成输出的文件名

-E 只进行预处理

-S 只进行预处理和编译

-c 只进行预处理，编译和汇编

## 多文件联合编程

1. 在同一个文件夹中，所有函数都是全局函数，所以，在任意文件中定义一个函数，均可以被其他方法调用,
2. 使用.h文件，.h文件用于声明函数所在的位置

// 函数、变量的声明

// 系统库的调用

extern int max(int a, int b);

1. 主函数引用.h文件

编辑器会自动引入文件，对于非编辑器使用命令

gcc -o 可执行程序.exe main.c **其他文件.c 其他头文件.h**

## 防止头文件被多次包含

#pragma once //版本高

# SOMEFILE可定义

#ifndef \_\_SOMEFILE\_H\_\_

#define \_\_SOMEFILE\_H\_\_

// 声明语句

#endif

## 预处理

预处理-编译-汇编-链接，

预处理是简单文本替换

#defined 定义文本没有作用域说法，会一直到文本结束

# 变量与常量

## 变量

int 整形 %d --4字节-32比特

float/double 浮点型 %f --8字节-64比特

short 65535-长度 --2字节-16比特

long

字符类型：char --1字节-8比特

每一个字符都有自己的编码ACSII，如果字符串按数字输出，将输出ACSII编码

## 常量

1、定义方式

const int PI = 3.14

#define PI 3.14 #符号常量，宏定义

注意：

1. 通过#define定义的常量是根据值来匹配数据类型
2. const 定义的常量是不安全的，可以通过指针来修改（C语言中）
3. 字符串常量

# 五、数组

int a[] = {1,2,3}; //声明一个数组

1. **数组本身就是指针**

printf(“%p”, a); //打印数组a的地址

1. **数组名就是指向数组下标0的指针**

// 这样的值都是一样的

printf(“%p”, a);

printf(“%p”, &a[0]); //a[0]取的是值，必须加&;

1. 直接打印数组名，会打印数组的首地址

// 这3中方式的值都是一样的，都是a的地址

printf(“%p”,&a);

printf(“%p”, a);

printf(“%p”, &a[0]);

## 一维数组(连续空间）

**定义数组**

int Array[20]； //定义20长度的数组

**初始化的时候赋值**

int Arr[10]={0,1,....} //定义一个10个大小的数组,连续空间

int Array[]={1,2,3,4} //定义并赋值

**普通赋值**

Arr[0] = 100;

**获取数组的个数**

int len = sizeof(Array) / sizeof(Array[0])

printf(“%d”,Arr[1]) //输出

## 一维数据内存存储

根据操作系统与定义的数据类型不同，数据占用的空间不同，

比如int是4位（byte)

数组是内存中连续存在的空间，（VS中调试显示的是16进制，并有数据对齐等问题）

printf(“%p”, &Arr[1]) //以无符号十六进制格式打印

数组名=数组的第一个元素，下标0

## 二维数组

int A[10][2];

# 字符串

## 数组

字符串本质上其实就是以'\0作为'结尾的特殊字符数组，所以定义字符串时，必须保证字符串存储的最后一个元素为'\0'。

**%s会接受\0之前的所有字符**

**错误赋值：**

**原因：因为name是一个数组（字符串数组），所以name实际上是一个指针（8位或16位等），指针是不能随意赋值的，要赋值也要赋值地址;**

**char name[10];**

**name = “ssss” ; “表示字符串**

***正确赋值-定义赋值***

Char name[]=”holle” //**字符串**，编译器会自动添加 \0

Char name[]={“h”,”o”,”l”,”l”,”e”} //字符串数组，**没有\0**

// \0字符串结束标准，字符串的内容要循环修改，将字符一个一个赋值，不能直接等于赋值

char name[20]={‘a’,’b’,’\0’}

**数组常量字符串**

char \*name[]={ //常量区

“aaa”,”bbb”,”ccc”

}

***正确赋值-常量赋值（****字符串常量不能修改）*

char \*name = “字符串常量” //定义赋值

***// 只能赋值1次***

char \*name;

name = “aaaaa”;

## 字符串结构体

struct Person{

         char name[20];

         int age;

         float score;

} p = {"lilei",20,80}; 定义赋值

### 定义后赋值

typedef struct {

         char name[20];

         int age;

         float score;

}Person;

Person person = {"lilei",20,80};

Person person2 = {.age=20,.score=80}; //部分初始化

### 单独初始化

struct Person{

         char name1[20];

         char \*name2;

         int age;

         float score;

} p;

// p.name1 = "lilei";//错误用法

strcpy(p.name1, "lilei");//正确用法-正常数组赋值方法

p.name2 = "lilei"; // 指针常量数组，才能用“=”赋值

p.age = 20;

p.score = 80;

## 字符串数组操作

#include<string.h>

char str\_str[] = “helloWold”;

char str\_arr[] = {“a”} //字符数组需要传递 这个数组的大小

**数组传递字符串**

void print(char \*str\_str) //字符串只传递一个变量就可，因为字符串结束标志为\0

{

int len = strlen(arr) //通过str函数获取字符串长度

}

void print\_arr(char \*str\_arr, int len)

{

printf(“%s”, str\_arr); //打印字符串，到\0结束

}

# f六、函数

形参与实参的区别表现

函数里的变量，在函数结束的时候会清空内存，只有放到变量中不会被清除

## 1、声明

extern int add(int a, int b);

int add(int a, int b); //extern可省略

## 2、定义/实现-带有{}

int add(int a, int b)

{

return 0; //必须有返回值

}

## 3、调用

int main()

{

add(1,2);

}

## 函数参数

**函数传值一般情况，数组/结构体不同**

### 传值

值传递在**函数运行结束**的时候会销毁

int func(int a, int b){

printf(“a=%d”, a); //a传递的是值

}

**//函数需要值传递**

**int main()**

**{**

**int a=10, b=20;**

**func(a, b);**

**}**

### 传地址

地址传递在**进程结束**时结束销毁

// 地址传递-可以将值带回, 如果a传递的不是地址（&），编译器会报错

int func(int **\*a**, int **\*b**) { // int\* a代表定义

**printf(“a=%d”, \*a); //因为a是地址，所以要用\***

}

**int main()**

**{**

**int a=10, b=20;**

**func(&a, &b); //取地址**

**}**

## 数组做函数参数

// void func\_01(int \*a) //对于数组，与下面相同

void func\_01(int a[])

{

printf(“%d”, a[0]); // 数组a[0],1

}

int main()

{

int a[] = {1,2,3}

func\_01(a); **//调用函数**

// func\_01(&a); //等同于上面

}

## 函数指针做参数

int func(int a){...}

void doLogin( **int(\*pfunc)(int)** ){//调用函数指针

funname = pfunc(1)

}

## 函数指针构成的数组

void func1(){}

void func2(){}

void (\*func\_array[2])();

func\_array[0] = func1;

func\_array[1]=func2;

func\_array[0]() //函数调用

## 函数指针

函数名 = 函数得入口地址

把指针转化为函数指针-将Null转换为函数指针类型

int (\*pFunc3)(int, char) = (int(\*)(int, char))Null;

# 指针

为什么要有不同类型指针

在同一个编译器下，一个指针变量所占用的内存空间是固定的，

但是一个类型的大小是不固定的，int 4字节，char 1字节，指针要根据不同的类型读取不同的长度

指针指向空，\*p=NULL;

**1、标记**

\* 在**定义时**强调这是一个指针类型，其他时候代表**取值**

\* 取值（只有对**指针类型变量**才可以使用） & 取地址（任意变量）

**2、指针步长**

**指针步长由指针类型决定（char 1,int 4)**

指针类型的主要作用是确定从内存中根据指针类型确定读取几个字节

**3、指针大小**

无论是什么类型得指针，存的都是地址，内存地址都是无符号整形，都是4字节大小

**4、指针变量存放非地址数据没有实际上的意义**

p=a // p保存的是a中的值，相当于普通变量

printf(\*p) //这里会溢出，因为P变量存的值是5,\*p是找 内存中指向5的位置

## 使用例文

int a=5,b,c

int \*p //**\*声明指针类型** int类型的指针\*p，4个字节

p = &a //p中放的是地址（a的地址，也可以放123456，但没有实际操作意义）

**// p = 123456; 对123456这个地址操作，可能会内存溢出**

print(“p=%p”, p); //x0083212 打印a的地址，因为p存放存放的是&a地址

\*p = 15 // 1、取p中地址指向的位置的值，2、将其值改为15,相当于**a=15**

c = \*p //将指针p中的值 赋值给c， c=15

b = &a // 将地址赋给b,但b不是指针变量，不能使用\*

print(“\*b=%d”, \*b) //错误，b不是指针类型

void \* p = &a //万能指针，void

\*（int \*)p = 100 //当取值得时候要进行类型转换

## 溢出

\*p=&a // 这里会溢出，不知道指到哪里了

**\*必须是指针类型才能使用**

int p, b=5;

p=&b; //a中存放地址

printf(\*p); //语法错误，只有指针类型才能使用 \*

printf(p); // 打印b的地址

## 移动指针

**步长由指针类型确定**

int \*p = stu; //stu是个数组

\*p // 指向下标为0的数据

\*(p+1) // 指针向后移动2个位置，指向下标为1的数据

\*((int \*)p +1) //强转int类型，根据int类型确定步长，移动1步，取值

## **指向数组的指针**

int stu[2] = {1,2};

int \*p;

p = &stu[0] // stu[0]存放的是首元素，so，要使用 &stu[0]

p = stu // stu是数组，数组本身就是地址

p = &stu **// 推荐**

\*p=stu // 将stu放到变量p中

\*p=&stu // 将stu的地址放到P中

## 指向函数的指针

int \*funaddr = （int\*）4199952 //4199952是一个函数的入口地址

void(\* myfunc)(参数列表) = funaddr

myfunc(); //调用函数

**定义一个函数**

int func(int a){} //自定义函数

**自定义一个函数指针**

int (\* pfun)(int a)； // 定义一个函数指针

**指向一个函数**

pfun = func; // 指向一个函数

pfun = &func; // 指向一个函数

**调用**

pfun（10） // 调用函数

(\*pfun)(10)//调用

## 万能指针 / 野指针

野指针 -> 指到不知道的地方,

void \* p = &a; //不知道a是什么类型

// 要是修改这个万能指针的化，需要强制转换成指定类型

\*（char \*)p = ‘a’; //char类型是一个字节

//

int a[10] = {0};

void \*p = a;

\*（（int \*)p + 1) = 200 //这里+1相当于加了一个int类型的字节，地址会+4

## 二级指针

void aaa(int \*\*val){ \*val = 0xxxx}

int main()

{

int \* p = Null;

aaa(&p);

}

# 结构体

为什么必须使用结构体类型

不同的结构体，大小是不同的，依据结构体内部定义的变量而变

结构体的存在是为了节省代码，相同的变量，不用多次声明，通过结构体声明多个具有相同属性的变量

结构体需要根据数据类型进行数据对齐

所有数据类型的大小，在内存中存储的地址一定是类型的倍数

结构体定义的数据位置不同，大小也不同

创建结构体一定按照由小到大定义

char - short - int - long - double - float

**结构体变量，结构体数组，结构体指针**

**没有直接接打印结构体得方法，如果是指针指向一个结构体，这个指针变量保存得是结构体第一个元素得地址，**

## 栈空间使用方法一

1. **声明结构体**

struct Player{

int a;

}

1. **定义结构体 p1**

struct Player p1; // p1包含Player中声明的所有变量

1. **初始化结构体变量赋值**

struct Player p1 = {1};

struct Player p1 = {.a = 1};

p1.a = 1;

**4、使用结构体**

printf(“%s”,p1.a); // 给结构体的属性赋值

## 使用方式二

struct Player{

int a;

}p1,p2

// 定义了2个结构体变量（Player）:p1,p2

// 相当于struct Player p3;

// 相当于struct Player p4;

p1.a = 1

-----------------------------------------------------------------

// 可以这样定义，但这样定义将不能定义p1,p2之外的结构体变量

struct { //这个结构体并没有标明其标签

int a;

}p1,p2

## 使用方式三

*：少写一个变量名*

typedef struct { // 用typedef创建新类型,新类型名为Player

int a;

}Player

Player p1; //定义一个结构变量

p1.a = 1;

---------------------------------------------------------

typedef struct \_Player{

int a;

}Player;

Player p1;

p1.a = 1; // 给结构体的属性赋值

struct \_Player p2 //结构体定义

## 指针使用结构体

typedef struct ***\_Player***{

int a;

char \*b;

struct ***\_Player*** \*d; //结构体名主要用于在结构体内定义

}Player;

player p1;

p1.a = 111;

Player \* nowp //声明一个Player类型的结构体

**// 数组名在表达式中会被转换为数组指针，而结构体变量名不会，无论在任何表达式中它表示的都**

**// 是整个集合本身，要想取得结构体变量的地址，必须在前面加 &**

**// nowp = p1 这种写法是错误的**

nowp = &p1; // 将结构体指针，指向p1（p1是一个player类型的结构体）

nowp->a = 2222; **// -> 指针的成员操作符与.意思相同**

## 堆空间开辟结构体

struct tec

{

char \*name;

int age;

}t;

//开辟堆空间的结构体，sizeof可以直接计算结构体大小

struct tec \* p = (struct tec \*)malloc(sizeof(t));

p->name = (char\*)malloc(sizeof(char) \*21)

strcpy(p->name, “ss”);

// 开辟3个连续空间的结构体

struct tec \* **d** = (struct tec \*)malloc(sizeof(t) \* 3);

**d[0]**->name = (char\*)malloc(sizeof(char) \* 21)

strcpy(**p[0]**->name, “ss”);

## 结构体数组-没确定使用方式

typedef struct \_player{

int a;

}Player;

Struct Player info[30] // 使用结构体Player 声明一个30容量的info

info[1].a=1;

## 联合体，

共用一块内存空间，只有最后一个值赋值时准确的

共用体根据最大的数据类型长度，确定大小

union vars

{

int a;

}job；

## 结构体数组

struct stu{

int a;

}

struct stu s[2] = {{1},{2}}

## 结构体交换

struct stu temp = s[j];

s[j] = s[j + 1];

s[j+1] = temp;

## 结构体做函数参数函数

typeof struct {

int num;

}stu

//struct stu s = stu s 其struct可省略

void func\_01(**stu s**){ s.num = 100; }

void func\_02(**stu \*s**){ s->num = 100 } //注意不能加\*,加\*就是再取地址了

void func\_03(stu &s){ s.num = 100 } C++

int main(）

{

stu s;

s.num =100;

//---------------------------------

//这种传递会拷贝一份s给函数，效率低

//所有的值传递都不能改变原本的值

func\_01(s);

//---------------------------------

**// 推荐使用**

// stu\* p声明指针类型p

stu \*p = &s; //通过结构体指针传递

func\_02(p);

//----------------------------------

**// 备选方法 func\_02(&s); C++**

func\_03(s); //C++中的引用传递

}

## 结构体错误的使用方法

SNode L;

create\_head\_s(&L);

void create\_head\_s(SNode\* L2)

{

// 这样是错误的

//1、函数体外已经声明了一个变量L，L已经是一个结构提，

//L2 等同于 L，这时再给L2赋值，就会替换掉函数体外的L

L2 = (SNode\*)malloc(sizeof (SNode));

}

## 结构体嵌套结构体

struct a

{ int a; }abc;

struct b

{

struct a c;

}

int main()

{

struct strb b;

}

## 8、枚举类型

enum SYS{ // 枚举类型的值是根据顺序排列的

ADD=1, // 1

QUERY=2, // 2

}

enum SYS input; // 声明一个枚举变量input

scanf(“%d”, &input) //输入信息放到input中,提示用户输入1、2、3、4

if (input == ADD) {} // 枚举类型会自动管理相关变量

enum colores{

red,bule //默认key red=0，bule=1

}clo;

switch(clo){}

# 文件操作

文件分类磁盘文件和设备文件

FILE \* f1; //定义一个文件指针

f1=fopen(“路径”, “w”)

文件结尾是eof 打印现象是大空格

feof(f1) //判断是否到eof

有利于磁盘寿命，有利于效率，从缓冲区读取

二进制文件读写

# typedef起别名

typedef unsigned long ull;

ull a =10// unsigned long a;

struct student

{

int a;

}

typedef struct student ss;

typedef struct stucent{ int a;}stu;

stu ss;

typedef char \* PCHAR;

PCHAR p1, p2; //p1，p2都是char \*

char \*p1, p2; //p1时char \* p2是char

# void

无类型，定义变量会报错

# 产生随机数

#include<time.h>

#include<stdlib.h>

int main()

{

#添加随机数种子,不添加随机数种子，随机的数都是一样的

srand((unsigned int)time(NULL))

rand() //产生随机数

rand() % 100 //0~99的随机数

rand() % 11 + 50 //50~60之间的数

rand() % 10 + 50 //50~69之间的数

}

# 使用内置函数拷贝

//主要是因为，C语言中字符串是存放在数组中的，拷贝字符串需要把数组中每一个元素拷贝一遍

#include <string.h>

strcpy(a,b)

# const int

使用const定义的值是不安全的,因为可以通过指针修改

#define LV 100 //这样定义是无法修改

const int \* p //不能改变指针指向的数据，但可以修改p中的值（地址）

int \* const p = &a //可以修改指针指向的数据，但不能修改p的值（地址）

改变指针类型=改变指针步长

读取结构体

struct A {char a;int a}

struct A a = {‘a’, 20}

# 内存对齐

cpu按块读取，这一块是4字节

内建数据类型已经确定

提高存取数据速度，

# ->与.的区别

-> （通过指针访问）（堆区数据访问）

结构体指针

malloc创建的类、结构体

this->a 类中调用属性

new 一个对象

----------------------

. （栈区数据访问）

C++中类外访问类方法、属性

结构体的访问

# 其他常用函数

## 输入输出

// scanf的变量必须添加地址&

int a;

scanf(“xx=%d”, &a);

对于数组来说，

scanf(“%d”, &a)与scanf(“%d”, a)都正确，因为数组的特殊机制，数组a=数组首地址，&a与a相同

## 调用一个外部程序

在程序中调用一个外部程序

#include <stdlib.h>

int system(const char \* command);

参数：外部可执行程序名称

返回值：失败 -1

#include <stdio.h>

#include <stdlib.h>

int main()

{

system(“ls -all”);

}

## sizeof()

*分配内存空间，以字节为单位（B）*

1. **对于数组(array)，sizeof(array) 会计算整个数组的大小**

int a[] = {1,2,3,4};

printf(“%d”, sizeof(a)); //16，会计算整个数组大小

1. **对于结构体（struct），sizeof(struct)会计算整个结构体大小**

*不用这样写 sizeof(a) \* 10 （a是一个结构体）*

SNode\* z;

sizeof(\*z); // 打印z对应的结构体大小

sizeof(z); //打印z的大小，z是一个指针，固定大小

----------------------------------------------------------------------

1. **数组做函数参数传递时，（数组指针会退化为指针）**

void func\_01(int a[])

{

printf(“%d”, a[0]); // 数组a[0],1

printf(“%d”, sizeof(a)); //会打印指针大小（32位4，64位8 ）

}

int main()

{

int a[] = {1,2,3}

func\_01(a); **//调用函数**

}

传递的是指针（因声明是一个指针），所有sizeof(array)只能读取这个指针的大小

**2、结构体做函数参数传递时（结构体指针退化为普通指针）**

typeof struct {

int num;

int aa[30];

}stu

void func\_01(stu s) //这种传递会拷贝一份s给函数，效率低

void func\_01(stu \*s)

{

printf(“%d”, s->num); // 结构体stu的num ,10

printf(“%d”, sizeof(s)); //会打印指针大小（32位4，64位8 ）

//这样只能计算整个数组的个数，不能计算一共有几个有效数据大小

int a = sizeof(s->aa)/sizeof(s->aa[0]);

// 如果用for可以直接使用int i的个数

}

int main()

{

stu s;

s.num = 10;

func\_01(s); **//调用函数**

}

**sizeof（）计算函数内数组个数时**

不能建议计算！！！！

通过函数增加数组个数参数，判断数组个数

## 计算数组长度C/C++

//这个如果计算字符串，会算上\0

sizeof(arr) / sizeof(arr[0])

## 动态申请空间malloc

#include<stdlib.h>

int \*a, b;

scanf(“%d”, &b); //b是输入的变量

a = (int \*)malloc(b \* sizeof(int)) //生成b个int类型大小

printf(“\*a);

**动态分配空间**

# C语言-Player结构体动态分配空间

Player \* None=（Player \*）malloc(sizeof(Player));

# C++分配空间

Player \* None = new Player;

////释放空间，是一个指针

free（None）