嵌入式C语言巩固
2024/01/22
嵌入式C语言王海宁-*Rocky*麦子学院
麦子学院

1-1 gcc

1.1.1 概述：

全名：GNU C Compiler（GNU Compiler Collection） 翻译官，翻译组织，把人能看懂的文件转换成机器能看懂的

`gcc -o [输出的文件名] [输入文件]` o： output

gcc是根据文件后缀名来确认调用的"翻译"模块

`gcc -v -o build test.c` 输出编译过程

1.1.2 C语言的编译过程

- 预处理：[gcc -E]
- 编译 ： [gcc -S] 编译出一个.s文件
- 汇编 ： [gcc -c] 将.s文件编译成.o文件
- 链接 ： [gcc -o] 库文件与对象文件链接

1.1.3 C语言中常见错误类型举例

- 预处理错误：预处理过程是对头文件和define展开（宏展开）

`#include "xxxx"` 双引号自定义的文件`#include <xxxx>`尖括号系统定义的文件

> 对于尖括号，预处理器会在系统预设的头文件包含目录搜索头文件
> 对于双引号，预处理程序则在目标文件所在目录内搜索相应文件，如果当前路径没有找到所包含的头文件，则会到系统预设目录进行搜索，也就是说**使用双引号的搜索路径包含了使用尖括号的情形**；

典型错误提示：`No such file or directory` 编译器找不到文件

可能原因：尖括号和双引号乱用、文件目录路径问题

- 编译错误：语法错误
- 链接错误：

  原材料不够： 提示：`undefine reference to 'xxx'` 寻找标签是否实现，链接时是否加入一起链接

  多了：提示：`multiple definition of 'xxx'` 多次实现了标签，只保留一个标签实现

1.1.4 预处理的使用

```
#include    //包含头文件
#define     //宏 替换，不进行语法检查

#define    /*宏名*/   /*宏体*/        //一般宏体加括号 保证运算优先级的安全问题

#ifdef #else #endif                   //条件编译 根据条件开关来确认代码段的执行
/*好处：方便版本调试，将代码段进行预控制
gcc -DABC -o build 003.c    -> 将003.c文件编译输出，输出文件名为build，同时预编译过程中打开ABC宏（即含ABC作为条件编译的代码也进行编译）
*/


//预定义宏(系统定义宏)
__FUNCTION__
__LINE__
__FILE__


#       //字符串化   主要使用场合：将宏中需要赋值字符串时使用
##      //连接符号   替换大量前缀相同的情况下的代码
#define ABC(x)      #x
#define DAY(x)      myday##x
int main()
{
    int myday1 = 10;
    int myday2 = 20;
    printf("the day is %d\n", DAY(1));      //实际打印传递的参数时myday1 输出10
}
```

1-2 C语言常用关键字和运算符

1.2.1 关键字

概念：编译器**预先定义**了一定意义的**字符串**（方便编译器解析的），C语言中一共有32个关键字

杂项关键字

sizeof：编译器提供查看内存空间容量的一个工具

return：返回的概念 函数返回

（1）数据类型关键字

- **char**（一般对应1字节 最大范围256 2^8）

应用场景：硬件处理的最小单位

- **int**

大小：**不固定** 根据编译器来决定 一般为4字节（4byte 2^32 ）\ 2字节（2byte 2^16 65536）

编译器最优的处理大小：系统一个周期，所能接受的最大处理单位

> int a = 010;// a十进制下是8 编译器当成了八进制，由于第一个是0
> int a = 0x10; //a有0x限制 所以是16进制 a十进制下等于16

- **long、short**

特殊长度的限制符 补充int char的内存分配短板
int char long short整型类型的组合 内存分配空间的思想

- **usigned、signed**

数据类型的限制标志 限制开辟的内存空间的**最高字节**是 **符号位** 还是**数据**

无符号：数据概念（显卡产生的、采集的）
有符号：数字概念（用于加减乘除使用的）

右移操作：>> 右移操作下符号位会干扰数据。所以一般数据会使用unsigned进行规范：unsigned int a

- **float、double**

用于浮点运算，小数

大小：float（4字节byte 4byte 2^32）double（8字节 8byte ）

> 浮点型常量：1.0 1.1 （实际上是double型） 1.0f（实际上是float型）

内存存在形式：和整数有所不同

- **void** 占位标识 ……

（2）自定义数据类型

C编译器默认定义的内存分配不符合实际资源的形式而衍生
自定义 = 基本元素的集合

- **struct**

元素之间的和

```
struct mayabc        //元素和元素之间是递增的关系（地址递增）
{
    unsigned int a;
    unsigned int b;
    unsigned int c;
    unsigned int d;
};                   //这个语句还未申请内存空间 这里只是告诉编译器 具备myabc内存空间定义的能力
struct myabc mybuf;    //正式将struct myabc进行变量化，相当于在内存里开了一块地
```

结构体中的顺序问题：顺序是有要求，位置颠倒，代表的意义变了

- **union** 共用体

共用起始地址的一段代码
技巧型代码

```
union myabc
{
    char a;
    int b;
};                        //类似结构体 先声明
union myabc abc;           //定义内存空间（开辟）
```

- **enum**

enumerate 列举
被命名的**整型常数**的**集合**

一般用于数据的"分类" / 命令的组合 / 被打包的常量集合

- **typedef**

数据类型的别名（外号）
方便代码阅读 常与指针一起交叉使用，在定义一些变量的时候会更加方便

（3）逻辑结构

程序执行方式的建议符号

- **if、else**

分支/选择 条件

根据条件表达式进行判断

- **switch、case、default**

多分支 多重选择性 switch(整型数字)

- **do、while、for**

for：次数 循环次数的概念
while：条件 循环的条件 注重在什么条件下循环
do while：先后关系的问题

- **continue、break、goto**

continue：继续
break：跳出循环
goto：在函数内部跳转方便使用的语法

（4）类型修饰符

对内存资源**存放位置**的限定 资源属性中位置的限定
（在嵌入式开发中，内存分配的具体位置实际上和某些代码和某些执行流程有很大的关系，比如开机画面，画面素材存放的位置不可写）

- **auto**类型

默认情况下都会实现的关键字 编译器解析常见数据类型内存位置的默认值 ---->默认分配的内存都是可读可写的
`int a ----> auto int a; char b -------> auto char b; 如果定义的这些变量的位置在大括号{}内，则默认分配到栈空间`

- **register**

限制变量在寄存器上的修饰符 定义一些需要快速/频繁访问的变量
编译器会尽量的安排CPU的寄存器去存放这个a，如果寄存器不足时，a还是放在存储器中（不保证100%有用）
"&"取地址这个符号对register修饰的变量不起作用 `error：address of register variable ‘a’ requested`

`int a ----> register int a //用register修饰int a 那a的存放位置会在寄存器上存放`

- **static**(内核开发、内核源代码、多文件工程编程中会大量存在)

静态
应用场景：修饰3种数据：

```c
//1 函数内部的变量
int fun(){
 int a;   ======>   static int a;
}

//2 函数外部的变量
int a;       ======>   static int a;
int fun(){
 //
}

//3 函数的修饰符
int fun();   ======>   static int fun();
```

- **const**

常量的定义 （可理解为只读的变量，但是这个量还是可以通过某种方法改变这个量<跟变量存放内存的位置相关>）
`const int a; //则a不能改 const修饰表示是常数不可修改（在编译器中编译的时候不能显式地修改）`

内存泄漏

- **extern**

外部声明 主要应用全局变量函数中使用

- **volatile**

告诉编译器编译方法的关键字，不优化编译 volatile的使用一般和**地址**关系比较大/ 在嵌入式底层/驱动开发中经常使用
修饰变量的值的修改，不仅仅可以通过软件，也可以通过其他方式（硬件外部的用户）

防止在汇编阶段优化修改原本的程序执行流程

> B站嵌入式C语言

1.2.2 运算符

（1）算术操作运算符

- **+、-**
- ***、/、%**
  `int a = b * 10;` CPU可能多个周期，甚至要利用软件的模拟方法去实现乘法
  `int a = b + 10;` 加减类CPU一个周期就可以处理

%：求模
一个数和某个数求模的时候，最后的结果一定是这个数的0~n-1倍
`n % m = res  [0 ~ m-1]`
使用场景：取一个范围的数 得到一个n进制的一个个位数 得到一些循环数据结构的下标

（2）逻辑运算

真与假的选择 逻辑运算的最后结果不是真就是假 即二值性
返回结果：1/0　　1：真　　0：假

- **||、&&

`A || B ====> B || A 两个结果不相等`

原因：在C语言编译器中他首先根据翻译顺序先获取A的值，确认1/0，这时就没有必要看B了，所以执行完A后就不执行B，同样&&也相同

`A && B ====> B && A 两个结果不相同，原因参考||运算顺序 即C语言执行中会有一个预判断的过程`

- **<、<=、>、>=** 小于等于 大于等于 简单的数学概念
- **!**

逻辑操作中的取反，这里的取反和逐位取反不同

对比位运算中取反

```
int a = 0x00;        //a的值为0
! a   if(!a){}       //则if内代表真 执行括号内代码
~a    0xff           //~ 逐位取反 结果为0xff
```

- **?:** 典型if else的变形

（3）位运算

- **<<、>>**

左移：乘法 *2 二进制下的移位 整体往左/右移了一位 以0补位

`m << 1; //m * 2; ========> m << n; //m * 2^n`

`int a = b * 32; ====C编译器会自动将这个语句转换成========> b << 5 //这种情况下CPU一个周期就可以完成这个运算`

负数计算机底层显示形式：补码 补码 = 原码除符号位外逐位取反 + 1

右移：除法 除以2

`m >> n =====>等价于m / 2^n`

右移补位的数字与符号变量有关，正数右移则最高位补0， 负数右移则最高位补1

- **&、|、^**

&：（在硬件中常被称为清0器）
屏蔽：a & 0
取出：a & 1

|：（设置为1/高电平的方法，设置set）
A | 0：等于A，保留了A的值；
A | 1：等于1， 设置了A；

设置一个资源的bit 5为高电平，其他位不变 -> a = (a | (0x1 << 5))

`int a; ======> a = a | 100000 ======> a = (a | (0x1 << 5)) ======> a = (a | (0x1 << n));//即将第n位设置为1`

清除第五位 -> a = (a & ~(0x1 << 5))

`int a; a = a & 011111 ======> a = (a & ~(0x1 << 5));这样可以避免不同位平台下移植数据错误 ======> a = (a & ~(0x1 << n))`

- **^、~**

^：异或，相同为0，相异为1； 1 ^ 1 = 0, 0 ^ 0 = 0; 1 ^ 0 = 1, 0 ^ 1 = 1;
异或常用于交换两个数：

```
int a = 30, b = 20;
a = a ^ b;      //异或符号交换两个数；
b = a ^ b;
a = a ^ b;
```

~：逐位取反，注意不同位数系统下的取反，比如32位和64位下的取反操作

（4）赋值运算

- **=**
- **+=、-=、&=、……**

（5）内存访问符号

- **()**：限制符（即运算优先级的问题，`（a + b）* c`），函数访问符号（`int fun(); fun();`）
- **[]**：内存访问的ID符号，a[1],a[2]……，可以看作是第一个ID号、第二个ID号
- **{}**：函数体的限制符、结构体空间定义、空间的限制括号
- **->、.** ：连续空间，也就是自定义空间中不同成员变量的访问方法 ， ->：地址访问；. ：变量访问
- **&、*** ：&搭配变量名即为取地址`&p,取p的地址`， *搭配变量名代表指针`*p,p是指针变量`

1.2.3 逻辑操作

（1）顺序执行

（2）分支执行

（3）循环执行

1-3 C语言内存空间的使用

C语言操作的最终对象是内存资源

1.3.1 指针

（1）指针概述

内存资源的地址、**内存资源**的门牌号/标识/代名词 访问**内存空间**的最直接的关键词
指针变量：存放指针这个概念的盒子

> C语言编译器对指针这个特殊的概念有两个疑问？
> 1、分配一个盒子，盒子要多大 在32bit系统中，指针就4个字节
> 2、盒子里存放的地址 所指向内存的读取方法是什么？ 每次读取的字节数限制（char 1字节 int 可能是4字节）

指针指向内存空间，一定要保证合法性！

> `char *p;    //用char指针去读内存，本身不争取，应该改为 unsigned char *p  //和符号位有关`

（2）指针 + 修饰符（const、voliatile、typedef）

- **const**

常量 只读（const修饰的东西不能变）

`const char *p [推荐写法] 等价于 char const *p` -> p为核心节点 这时向前看有* 则编译器认为p为指针变量，则p指向的方向是可变的 再往前看，指向的内容被const修饰，所以指向的内容只读不能写，即p指向的空间是只读的，同时p可以指向任意的只读空间【当然也有可变的方法】；
简单理解为这种指针指向类似于字符串常量的东西 希望**指向的内容是固定**的 **描述"字符串类型"的门牌号**
`"hello worlk", "aaa"`

`char *const p[推荐写法]  等价于 char *p const` -> p被const修饰 再被*修饰 那么指针p指向的方向不能改变，即指向固定地址，但地址中的内容可以进行刷新；
简单理解为硬件资源的定义，比如LCD、显卡他有一个缓存的概念；CPU向一块缓存地址不断的获取数据，缓存地址芯片公司已经确认死不能改变，然后在这地址上不断刷新数据，刷不同的值，显示器就得到不同的像素值/数据，显示屏就显示不同的数据，所以更多的用于指向固定的地址，地址上的内容是可以不断被修改的（硬件资源）

`const char *const p` -> 指向的地址和地址上的内容都不能变 ROM设备 在硬件中描述比较多

```c
//C语言中看到任何代码都要想到它在内存中的表现形式是如何
//segmentation fault段错误：指针指向的内容被非法访问/书写
char *p = "hello world\n"        //实际上是 const char *p = "hello world\n"  这段字符串内存的属性是const char *p

*p = 'a';        //非法操作
//另一种情况
char buf[] = {"hello world!\n"};
char *p2 = buf;
*p2 = 'a';       //正确操作
printf("the %s\n",p)  //打印除aello world!
```

- **volatile**

防止优化指向内存地址 更多的是对指针指向内容的修饰 就是从硬件资源或其他设备里取出真实的值

- **typedef**

别名 将一些复杂的声明别名化

（3）指针 + 运算符

- **加减运算++、--、+、-**

指针的加法运算是指针**指向内存的下一个单位** `int *p = 0x12; p + 1 则对应 [0x12 + 1 * sizeof(*p)]`
指针的加法运算，实际上加的是一个单位，单位的大小可以使用`sizeof(p[0])`来获取 减法逻辑相同

p++ p--：自加自减：更新地址 内存向下/向上偏移的方式

- **[]**

一般使用方式：变量名[n]，n： ID/标签 地址内容（指针）的标签访问方式 非线性访问 访问的是内容（取出标签里的内容）

- **逻辑操作符**

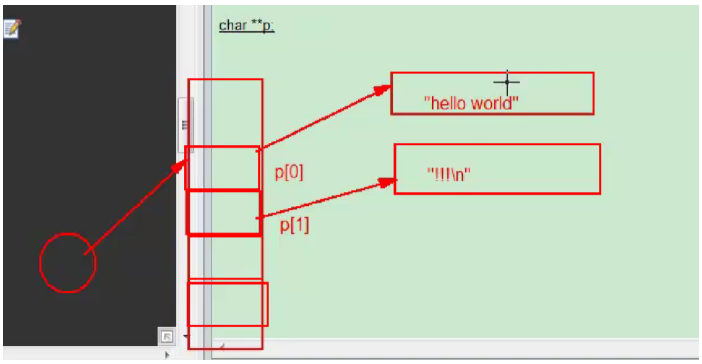
最常见的操作是"==、!="

跟一个特殊值进行比较 0x0 NULL 0x0：地址的无效值，结束标志
`if(p == 0x0){};    if(p == NULL)` 可以用于判断申请到的内存是否为空

指针必须是同类型的比较才有意义（主要在编译过程中比较 编译器的工作）

（4）多级指针

存放地址的地址空间 更多的是在描述内存与内存之间的**线性**关系
`int **p` ： 分开解析 *p 和 int 和 *



二维指针实际存了一堆地址，这堆地址又各自指向其他东西

1.3.2 数组

内存分配的一种形式 数组名不是指针，数组名是一个常量符号/标签 不可用于赋值操作 数组就是一个内存空间

（1）数组空间

数组的定义：定义了一个空间
属性：大小、读取方式
形式：数据类型 数组名[m]， m的作用域是在申请的时候起作用，建议符 `int a[100]`

（2）数组空间的初始化

空间的赋值：按照标签逐一处理赋值 `int a[10]; a[0] = xxx; a[1] = yyy; //a的范围是0-9`
　　空间定义时，就告知编译器的初始化情况，空间的第一次赋值，初始化操作

int a[10] = 空间；C语言本身，CPU内部本身一般不支持空间和空间的拷贝
int a[10] = {10, 20, 30}; ====> 反汇编说明 这个语句实际上是一个拷贝的过程 还是逐一赋值：a[0] = 10;a[1] = 20;a[2] = 30;a[4] = 0……

数组空间的初始化和变量的初始化本质上是不同的，尤其在嵌入式裸机开发中，空间的初始化往往需要库函数的辅助

```
1 char buf[10] = {'a', 'b', 'c'};      //bufff当成内存来看没有问题；buff当成一个字符串来看，最后要加上一个'\0'
2 char buf[10] = {"abc"};              //C编译器在翻译过程中看到双引号就自动加上了'\0' 这里其实是4字节 而第一行形式是3字节
3 //双引号可以理解为在内存中存在的
4 char buf[10] = "abc";     // 通过初始化的方法将这个内存拷贝到这个变量当中 常量向变量去拷 buf[2] = 'e' 合法
5 char *p = "abc";          //指针指向常量区 所以p[2] = 'e' 非法，出现段错误
```

strcpy, strncpy

一块空间，当成字符空间，提供了一套字符拷贝函数

strcpy(): （严重的内存泄露函数，长数据拷贝到短内存，多出来的数据覆盖内存的其他位置，导致数据丢失）

字符拷贝函数的原则：

内存空间和内存空间的逐一赋值的功能的一个封装体

一旦空间中出现了0这个特殊值，函数就即将结束

strncpy(): 相比strcpy，限制了拷贝的位数必须和所拷贝到的内存匹配，不可越界

非字符空间

开辟一个存储数据采集的盒子，一般定义这些内存时都用`unsigned char buf[xx]` 一定要用unsigned关键字

非字符空间拷贝三要素

1、src，拷贝的基地址　2、dest，拷到哪里去　3、拷贝的个数

memcpy(): `void *memcpy(void *dest, const void *src,size_t n);　//函数声明` n：代表多少个字节单位

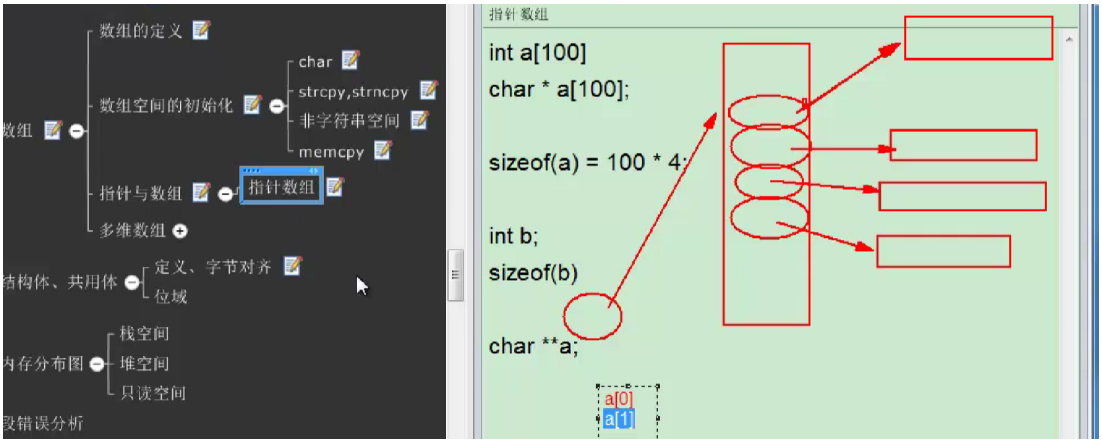
```
int buf[10];
int sensor_buf[100];
memcpy(buf, sensor_buf, 10 * sizeof(int));  //注意事项，buf[]是由int修饰的

unsigned char buf1[10];
unsigned char sensor_buf[100];
memcpy(buf, sensor_buf, 10 * sizeof(unsigned char)); //如果使用strncpy(),strcpy()函数进行拷贝的话，当数据中有0就会停止拷贝，数据完整性不能保证，
```

（3）指针与数组

指针数组：数组里面存放了一堆地址

```
char * a[100];       //类似多级指针
sizeof(a);       //a = 100 * 4,  a是一个常量/标签，这个常量的大小表示申请空间时申请的大小，申请了100个，每个4字节，4表示地址占用的字节数
```



（4）多维数组

数组名的保存

定义一个指针，指向`int a[10]`(线性空间)的首地址 ———> `int *p1 = a;`

定义一个指针，指向`int b[5][6]`的首地址

`int *p[5]` 编译器语句分析：先看到p，往右[5]，则为数组 空间为5，往左”*“，则五个空间存放”*“，类型为int型

`int (*p)[5]` 由于括号的原因：先看到*p p变为地址，往右看到[5] 则地址是5个形式 int方式来读取

```c
int a[10];
int b[5][6];    //5行6列

int *p1 = a;
int (*p2)[6] = b;       //可以理解为int [6](*p) 一次读6个

int b[2][3][4];
int (*p)[3][4];
```

1.3.3 结构体、共用体

（1）定义、字节对齐

结构体：事实上是数据类型/基本数据类型/其他构造数据类型打包的工具 **打包**

字节对齐：为了效率，希望牺牲一点空间换取时间的效率 4字节对齐也就是32bit对齐 大部分系统默认的情况
最终结构体的大小一定是4的倍数（根据系统不同而不同）

结构体内的成员变量的顺序不一致也会影响它的大小

```c
struct abc{
    char a;         //char 1字节
    short e;        //short 2字节
    int b;          //int 4字节   但该结构体总花费内存空间为8字节
};
struct my{
    char a;
    int b;
    short e;         //该结构体实际花费内存空间12字节
};
struct abc buf;
struct my  buf1;
printf("the buf is %lu:%lu\n", sizeof(buf));
```

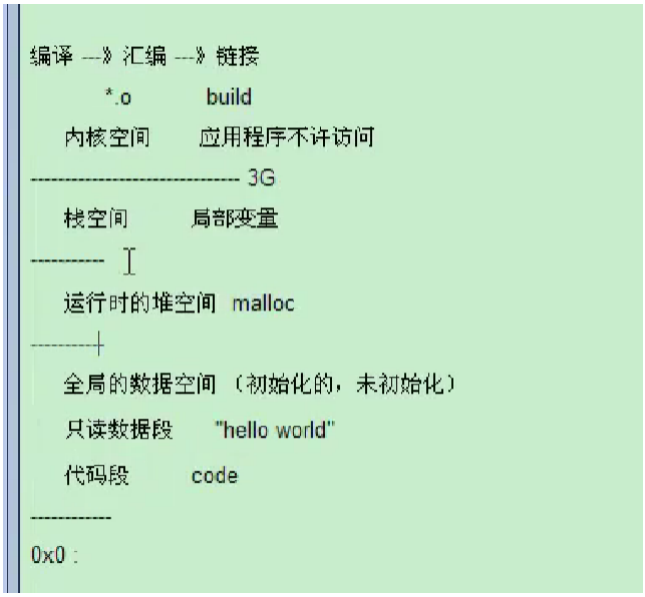
（2）位域

1.3.4 内存分布图

内存的属性：1、大小　　2、在哪里
C语言编译：编译----->汇编--------->链接

> 最终执行程序可以认为是一个大的容器，将.o认为是一个个原材料，我们需要将原材料组合在不同的位置上，所谓的链接就是将.o放在合适的位置上

定义a 在a的不同定义方式上`int a; a = 0x10; int a;//全局` 或者说定义的位置不同，那么a的位置也就不同



（1）栈空间

操作系统为我们提供局部变量的存储器 运行时函数内部使用的变量，函数一旦返回，就释放，生存周期在函数内

（2）堆空间

运行时，可以自由，自我管理的分配和释放的空间，生存周期由程序员来决定

分配：

malloc()，一旦成功，返回分配好的地址给我们，只需要接受，对于这个新地址的读法，由程序员灵活把握，输入参数指定分配的大小，单位就是Byte

释放：

free()，如果没有释放，则会出现内存泄漏

```
char *p;
p = (char *)malloc(100);
if(p == NULL)
{
    error;
}

free(p);
```

（3）只读空间

静态空间，整个程序结束时释放内存，生存周期最长

可以认为C语言进行程序设计时，在系统运行之前就已经分配的数据空间

代码段本身是只读的

数据段：

1.3.5 段错误分析

1-4 C语言函数的使用

```
1 void fun(const char *p, ...);        //p所指向内容是只读的，用在函数形参中，说明空间不可修改该，只读空间，为了空间看看
2 void fun(char *p,...);           //不保证空间是否只读，该空间可能修改
3 //常见例子函数：sprintf(); strcpy();
```

1.4.1 函数概述

一堆代码的集合，用一个标签去描述它 复用化

函数名：--------> 标签，一段连续空间的代名词，有标签就可以定位地址
函数和数组不同，**函数具备三要数**（数组只具备大小）

1、 函数名
2、 输入参数
3、 返回值

在定义函数时，必须将三要素告诉编译器 `int fun(int, int, char){xxx};`

如何用指针保存函数

```
1 char *p;
2 char (*p)[10];
3 //(*p)，p升级为地址，然后往右看括号，具备函数三要数，修饰该地址，所以p读内存的方法是按照函数三要数来读 指针函数
4 int (*p)(int, int, char);
```

定义函数，调用函数

```c
//函数指针的使用示例 函数指针去接收其他函数的地址值，然后可以通过这个地址直接使用该函数
//函数指针可以用于回调函数
int (*myshow)(const char *, ...);
printf("the printf is %p\n", printf);            //编译器打印结果是0x8048320
myshow = (int (*)(const char *, ...))0x8048320;  //将这个结果赋值给指针
myshow("==========\n");                          //程序成功执行
return 0;
//在此基础上可以利用一串数组保存函数指针，方便集合不同函数的调用
//核心思想：函数命就是地址
```

1.4.2 输入参数

承上启下的功能

调用者：上层

函数名(要传递的数据)　　//实参，实实在在的要传递的数据

被调者：下层

函数的具体实现

函数的返回值 函数名(接收的数据)　//形参，接受数据的数据类型形式 一种变量

{xxxxxx};

实参 ——> 形参 传递方式：拷贝

（1）值传递

保护概念：上层调用者保护自己空间值不被修改的能力。（主函数声明的变量值传递到外部函数后，主函数内的该变量值不变）

`int a, b; swap(a, b);`

（2） 地址传递

上层，调用者 让下层子函数修改自己空间值的方式 连续空间的传递

`int a, b; swap(&a, &b);`

- **作用**

1 修改　　int * char *

2 空间传递

2.1 子函数看看空间函数的情况　const char *p / const int *p

2.2 子函数方向修改上层空间里的内容（比如编码解码、摄像头采集）

（3）连续空间的传递

- **数组**

数组名----标签

```c
int abc[10];
fun(abc);        //实参 这里abc属于数组，所以直接是地址
void fun(int *p);//形参
void fun(int p[10]);//实际上还是地址传递
```

- **结构体**

结构体变量

```c
struct abc{int a;int b;int c;};
struct abc buf;
fun(buf);           //实参传递1
void fun(struct abc a1)//形参传递1 这里重新在内存中开辟了一个结构体空间用于接收数据

fun(&buf);              //实参传递2
void fun(struct abc *a2) //形参传递2 推荐：节约空间
```

- 字符空间：结束标志不同
  看到空间就要想到两个要素：空间首地址、结束标志

  结束标志：内存里面存放了0x00(1B)，则为字符空间
- 非字符空间：0x00不能当作空间结束标志

1.4.3 返回值

随笔

C语言操作对象：资源/内存（内存类型的资源，LCD缓存、LED灯） 访问方式：地址总线/数据总线

C语言面向内存 C++面向对象

C语言如何描述资源的属性：

资源大小

关键字：用来限制内存（土地）的大小，关键字

> int a -> 说明在这个内存空间中有这个变量名a来描述int大小的内存（土地）；char c -> 利用C变量名代替一个字节

硬件芯片操作的最小单位：bit 0/1
软件操作的最小单位：Byte（8 bit）

内存属性：

1. 内存操作的大小
2. 内存的变化性，可写可读

变量分配从内存地址从高往低分配例子

字符串的重要属性：结尾一定要有个'\0'

二维指针：圆圈指向内存，内存中存放的是地址又指向其他的内存

二维数组：内存读取是整块整块的读

C语言编译器其实就是字符串的解释器

《C语言深度解剖》

《C专家编程》

《程序员面试宝典》

《C和指针》