**1 《C语言基础语法》**

**1 输入输出**

printf(“string\_1””string\_2”);

prinf(cptr);字符串指针

printf(“Hello World %d”, a); 也可以

scanf(“%a”, &a); 输入的是地址, 否则没法修改其内容 (本质是getchar()构造的函数)

**2 数据类型**

数据类型表示该类型的数据占用多少字节 (它们的存储方式), C语言里最小的数据占用1 byte.

数据扩展或截断时 (char ⬄ int): <data\_type>表示扩展时1或0, 和读取时最高位是符号还是数值

(Tip: 可以用char得到一个byte的内容, 数据的截断和扩展字位)

(数据是补码形式存储)

小端存储: 数据的低位存储在低地址, 高位存储在高地址

大端存储: 数据的高位存储在低地址, 低位存储在高地址

数据的存储方式：

static修饰的数据(或函数)只能在该文件内访问(私有化), static数据只初始化一次 (记忆上次运算结果)

extern修饰声明的数据(或函数), 该数据在其他文件定义从而减少创建次数 (一般在.h里extern声明)

左移右移操作符只能对于正负整数, 不能对浮点数, 因此对于负整数而言左右移动的是他的补码

**按位操作符只是针对整数**

**3 运算符**

① 数值运算符: +, - \* /, %, +=, \*=, ++ …….(注意优先级)

② 逻辑运算符: &&, ||, !

③ 比较运算符: ==, !=, <=, >, ……

④ 位运算符: ~, &, |, ^, <<, >>

⑤ 三目运算符: A? B : C (当A成立时, 则B, 否则C)

转义字符: \字符表示其他含义。如\n, \0 ......

*(Note that: 上述A, B, C, …都是表达式)*

**4 基础语法**

(1) 顺序语句

逗号语句: (A, B, C, …, N) (运算从左向右, 最终是以N的结果)

运算按照上下顺序执行

switch(整型变量){

case A: 表达式; break;

……;

default: 表达式;

}

(2) 分支语句

if(判断表达式){……}

else if(判断表达式){……}

else{……}

(3) 循环语句

while(循环条件){……}

当循环条件!=0时才进入循环

for(初始条件; 判断条件; 变量改变) {……}

*Tip: 上述条件可以没有, 但要有 ;*

建议: 初始条件的变量在外面初始化, 如 int i = 0; for(i; i<10; i++){……}

do{……} while(循环条件);

当循环条件!=0时才进入循环

break结束所在的循环体

continue跳转出当前所在的循环, 进入所在的下一次循环

(4) go to语句

goto Label; ……; Label: 直接从goto跳转到Label然后执行后面的程序

Label: ……; goto Label: 直接从goto跳转到Label然后执行后面的程序

**2《C语言函数》**

**1 函数的结构**

① 创建函数: <data\_type> fun\_name(argument形参数列表){……; return result}

② 调用函数: fun\_name(实参数值); (得到的是该函数的返回值)

函数调用时, 在调用的空间里创建形参变量然后将赋值变量的值拷贝给这些形参, 因此函数里使用的是这些被赋了值的形参, 当函数使用完这些形参及时销毁, (函数里的参数都是局部变量存在栈区, 返回值是单独的寄存器, Ref: 函数栈帧)

当函数的创建在调用的后面, 则需要再调用之前声明该函数 <data\_type> fun\_name(arguments);

递归函数是指函数的返回值是该函数自己, 但实参值发生改变。

**3《C语言数组》**

**1 数组结构**

① 创建数组: <data\_type> arr\_name[num] = {初始化};

初始化一个值, 则全部值都一样 arr\_name[num] = {0或‘\0‘}, 数据类型: **<data\_type> [ ]**

如果<data\_type> arr\_name[ ] = {a, b, c}; 初始化后系统认为该数组有3个元素

② 数值的使用: arr\_name[i] = \*(arr\_name+i) = i[arr\_name];  **(数组名是首元素地址 = &首元素)**

对应的是数组的第i+1元素 (元素标号从0开始)

数组里的元素也可以是数组, 从而构成**多重数组**, 元素的使用arr\_name[i][j] = \*(\*(arr\_name+i)+j);

多重数组定义**只可以忽略第一个元素数量,** 如: arr[ ]**[2]** = {{e11, e12}, {e21, e22}, ……};

多重数组的数组名是首元素地址也就是第一个子数组的地址 (&sub\_arr1)

**2 字符串与数组关系**

① 定义字符串: const char\* str = “abc”; 等价char\* str = “abc”; 用const是为了规范

② 数组定义字符串: char str\_arr[ ] = “abc”; 等价char str\_arr[ ]={‘a’, ‘b’, ‘c’, ‘\0’};

Note that: 指针定义的字符串无法修改内容因为是const, 数组定义的可以修改内容

也可以char arr[]**[3]**={“ab”, “cd”}; 等价char arr[]**[3]**={{‘a’, ‘b’, ‘\0’},{‘c’, ‘d’,’\0’}};

关键字sizeof() 计算变量的字节(byte)大小 (数组作为函数参数时, 实际是数组名(首元素地址))

函数 strlen() 计算数组里元素的个数, 遇到 ’\0’ 才停止

"string\_1" == "string\_2"比较的是地址

应该: strcmp (string\_1, string\_2);

然而对于int arr[] = {1, 2, 3}; strlen(arr)≠3; 因为没有 ‘\0’

在C99之后可以int n = 5; int arr[n] = {……};

**4 《C语言指针》**

**1 指针**

① 指针变量(简称: 指针): <data\_type>\* ptr = &variable; 变量ptr存储的是变量variable的**首字节地址**

常数量 (直接嵌入内存) 和 宏(在于编译时被替换)没有地址

const变量 和 字符串有地址 *~~(char\* ptr\_1 = “abc”; char\* ptr\_2 = “abc”; ⇨ ptr\_1==ptr\_2)~~*

Note that:

* + - sizeof (指针) = 8byte (64位) 或4byte (32位)
    - 不同的数据类型占有的字节大小不同, 又有大小端存储, 因此首地址可能是这些数据的高位或地位字节
    - char只有1 byte从而没有大小端存储的区别
    - data\_type\* ptr = NULL; 该指针变量ptr为空值 (不指向任何地址, 也称为: 悬空指针)

② 指针的类型:

(1) 指针变量±1对应的是<data\_type>步长, 即地址移动sizeof (data\_type)步长

(2) \*ptr (解引用)后访问该首字节后面多少个字节, 即访问sizeof(data\_type)的空间长度

Note that: char是一个最小访问空间长度 (1 byte), 可以用它来强转得到首地址对应的数据

**指针只要类型一样, 和运算步长一样就是同一类指针(如: 数组名==它内部元素类型的指针)**

void\* 没有具体类型的指针, 所以不能解引用操作 (由于没有具体类型就无法知道解引用后访问

的内存大小, 也不能±整数进行步长运算), 但void\* 可以被赋值为任何类型的指针, 在解引用前

要进行类型转换 (转换为某个具体类型)。

int a = 10;

强制转换只是临时改变了ap的使用方法, 但实际的ap还是void\*类型。

void\* ap = &a; //赋值为任意类型的指针

\*(int\*) ap;

const <data\_type>\* ptr = &variable; .无法修改ptr所指向的地址对应的数据, 由于是const

<data\_type>\* const ptr = &variable; 无法修改ptr所指向的地址变量, 由于是const

const <data\_type>\* const ptr = &variable; .无法修改ptr所指向的地址变量和该地址所对应的数据

**2指针数组&数组指针**

① 指针数组:

<data\_type>\* ptr\_arr[num] = {ptr\_1, ptr\_2, ……}; 数组的元素是 <data\_type>\*

类型: <data\_type>\* [ ] ptr\_arr是变量名

* + - 数组名ptr\_arr是首元素地址⇨ ptr\_arr = &ptr\_1; 因此ptr\_arr + 1 = &ptr\_2;

如果是多重数组, ptr\_arr 是 首元素地址(首个子数组的地址, 即 &arr\_1)

* + - 结合数组的使用方法调用里面的元素: \*ptr\_arr[i] = \*\*(ptr\_arr+i) = \*i[ptr\_arr]; (元素是地址)

(作为函数的参数: int arr[] 或 int\* arr / int arr[][3] 或 int(\*arr)[], 都是首元素地址)

② 数组指针:

<data\_type> (\*arr\_ptr)[num] = &arr; arr\_ptr是数组arr[num]的指针变量

类型: <data\_type> (\* ) [ ]

* + - 数组指针arr\_ptr±1的步长是整个数组的字节长度 (sizeof(arr))
    - 解引用里面的元素: (\*arr\_ptr)[i] = \*((\*arr\_ptr) + i) = i[(\*arr\_ptr)]; ((\*arr\_ptr) = arr数组名)

Tip: int(\*(\*arr\_ptr)[10])[5]表示: 数组指针指向包含10个元素的数组arr, 每个元素也是数组指针(指向含有5个元素的数组指针arr\_sub)

**3 指针函数&函数指针**

① 指针函数:

<data\_type>\* ptr\_func(arguments){…..; return <data\_type>\* ptr;} 返回值是个指针

类型: <data\_type>\* (arguments) func为变量名

注意: 如int\* fun(){int x=10; return &x;}此处的返回值是一个野指针, 由于局部变量x在函数结束

后就销毁了, 导致该变量地址不存在

② 函数指针:

<data\_type> (\*func\_ptr)(arguments)= &func 或 func (函数名就是本体的指针)

此处: <data\_type> func(arguments){……}

类型: <data\_type> (\*) (arguments) func\_ptr是变量名

* + - 可以结合指针数组里面存放函数指针
    - 一般不要用void类型的函数指针强转化为其他类型的函数指针

回调函数: 函数funA通过它的函数指针funA\_ptr被另一个地方(可以是另一个函数通过函数指针参数)在某个情况下调用(funA), 则称函数A是这个地方的回调函数 (也称: 被该地方回调的函数)

**5《C语言动态内存》**

**1 内存空间**

|  |  |  |
| --- | --- | --- |
| 高地址 |  | |
|  | 栈区: | 局部变量, 临时变量(形参), 使用完系统自动回收释放空间 |
| 堆区: | malloc, realloc, calloc 开辟的空间, free指令释放, 系统结束后自动释放 |
| 全局区 | 静态区: | 主要存放程序运行期间一直存在的变量 (全局变量, 静态变量) |
| 常量区: | 字符串字面量, const修饰的常量 |
|  | 代码区: | 编译后生成的程序指令0101指令 |
| 低地址 |  | |

**2 动态空间申请函数**

① malloc:

void\* malloc(size\_t size) 开辟大小为size byte的连续内存, return该空间地址的起始位置

有可能申请失败return为空指针

② calloc

void\* calloc(size\_t num, size\_t size)

开辟num个, 每个空间大小为size的, 而且初始化里面的数据 (创建出来的不是数组，只是动态

连续空间的首地址)

③ realloc

void\* realloc(void\* ptr, size\_t size)

ptr是要修改的空间的地址(起始位置), size是要将该空间调整为新的大小

realloc可以调节空间大小 (其实是新开辟了一个新的空间, 将之前的数据拷贝过来了, 有可能在

原空间上开辟, 也可能在新地址上开辟)

如果给realloc一个空指针, 则功能和malloc一样

\*\*\* realloc会自动把之前的空间释放, 如果realloc开辟失败则原来的空间不会被释放

④ free

释放申请的堆空间 free(ptr); 对于int\* a =&b; 栈空间的变量不能用, 由于是栈空间

free释放的堆空间要注意：指针是该空间的起始位置

free不能对同一个堆空间多次释放, 除非是该指针 = NULL, 因为free(NULL)等于什么都没做

free顺序是按照开辟的逆序

常见的动态空间错误

① 一定要检查创建的空间后指针是否为空 if(ptr == NULL )

② 当内存 free(p) 后, 要p = NULL;

③ 注意动态空间 (堆空间上的) 使用后要释放

**3 柔性数组**

① 结构体里最后一个成员可以允许是一个未知大小的数组 (前面必须有其他成员)，称为柔性数组成员

struct S{

int i;

int a[]; 或int a[0];

};

② sizeof(柔性数组的结构体) 数组不在计算内存范围内 sizeof(struct S) = 4 只有 int 的内存

③ 创建变量: struct S\* ptr = (struct S\*)malloc(sizeof(struct S)+40) 给柔性数组开辟 40 bytes

柔性是指: 开辟后的数组可以通过realloc去改变数组的大小

struct S\* ptr\_new = (struct S\*)realloc(ptr, sizeof(struct S)+80);

如果下述方法:

struct S{

int i;

int\* arr;

};

struct S\* ptrs = (struct S\*)malloc(sizeof(struct S)); 目的是放到堆区(与柔性数组一致)

ptrs->arr = (int\*)malloc(40);

int\* arr\_2 = (int\*)realloc(prts->arr, 80);

ptrs->arr = arr\_2;

但是这个方法要两次malloc，不推荐

**6《C语言结构体》**

**1 结构体**

struct Name{

int a;

char b;

float c;

} s1, s2 = {12, 'a', 0.3};

此处如果结构体定义在main函数外部, 则为s1和s2为全局变量, 如果没有Name则是匿名结构

体, 此时只能用s1, s2不能在创建其他的对象

可以: struct Name s1; 也可以初始化

struct Name{struct Name\* str\_ptr; ……;} 结构体可以把自身的指针存放在里面, 由于指针只占用4或8

byte 但不能存放struct Name st; 因为无法确定结构体总体大小

结构体的数据存储方式：

(1) 第一个变量存储在偏移量为0的起始地址处

(2) 其他变量的地址偏移量 = 最小整数倍 of min(编译器默认对齐数与该变量字节大小的)

(中间的字节空间都是空的)

(3) 结构体的占用的总空间 = 整数倍 of max(所有变量对齐数)

各变量的对齐数是: min(编译器默认对齐数与该变量字节大小的)

(4) 结构体里嵌套了其他结构体, 则该嵌套的结构体的所占总空间为自身大小, 对齐到偏移量为自身里

最大对齐数的整数倍, 该空间总大小 = 整数倍 of max(所有对齐数包括嵌套的结构体)

(结构体自身的对齐数是自己内部的 max(内部变量的对齐数))

结构体的位段一般不使用, 因为跨平台效果不好。

**2 枚举**

enum Day{

Mon = 1, //枚举常量 Mon = 1, 是初始值, 默认 = 0; 后面的依次递增

Tues,

Wed,

};

Note that: 每个枚举的内容后面都是 **,** 逗号

enum Day d = Wed; //枚举变量d的取值只能是定义的值(Mon, Tues, Wed), 这些值也可以直

接使用对应int值, 如Wed 就是int值3

枚举只有在定义变量时才占用空间, 也可直接用枚举里定义的类型

枚举量是有int值的 默认从0→...., 也可以自己修改例如: Mon = 1;

为什么要用枚举：

（1）增加代码的可读性

（2）和#define定义的相比较enum有类型检查, 更加严谨

（3）防止命名污染 (封装)

（4）便于调试

（5）使用方便，一个可以定义多个变量

**3 联合体**

union Un{

int a;

char b;

};

union Un u = {1, 'a'};

u u.a u.c的地址都是一样的, 所占的空间是最大对齐数的整数倍, 共享一个空间联合体

里的数据类型同一时间每次只能使用里面的一个数据类型 (其他数据会被修改, 因为同一个内存)

**7《C语言文件处理》**

**文件处理**

每个被使用的文件都是在内存里有个文件信息区(是一个结构体 FILE) 因此这个文件的地址就是FILE\*

FILE\* fp 这个指针对应的就是该地址的文件的存储空间

文件信息区的地址 (确定文件所在内存空间地址) 的函数:

FILE\* fopen(const char\* filename, const char\* mode)

注意: 参数const char\* filename (不是本地路径的需要绝对路径 \\ 第一个 \ 是转义符)

使用完fclose(fp); （fp=NULL;）

FILE\* fp = fopen(.....); 这个地址是为了确定这个文件信息区的储存器的空间范围, 从而可以通过读写函

数对其内部内容操作

读写过程 (通过文件的所在位置(地址)对其操作)：

常用函数如：

int fputc(int character, FILE\* fp); int character 是把字符强转为ASIIC码值

int fputs(const char \*str, FILE \*stream);

**8《C语言编译预处理》**

1 程序的编译过程

main.c → 编译过程 (预处理, 编译, 汇编) → 链接过程

|  |  |  |
| --- | --- | --- |
| 编译 | 预处理过程 | 头文件包含, #define定义被替换并删除define, 注释删除, ......文本操作 |
| 编译过程 | 把预处理后的文件编译成汇编语言 |
| 汇编过程 | 把汇编代码转换为2进制代码, 每个原文件都有符号表 |
| 链接 | 链接过程 | 合并段表, 符号表的合并和重定位 (符号表的目的是可以跨文件链接) |

**2 宏 #define**

① 宏函数

#define name(ARGUMENTS) fun(ARGUMENTS)

#undef 取消宏定义的名字

例如: offsetof 计算结构体内数据的偏移量

#define OFFSETOF(struct\_type, m\_name) (size\_t)&(((struct\_type\*)0)->m\_name)

(# 和 ## 符号拼接作为了解)

② 宏判断

#ifdef 判断某个宏是否已定义

#ifndef 判断某个宏是否没有被定义

#define 定义宏（通常配合条件编译使用）

#undef 在此之后不再使用这个宏定义

#if 更复杂的条件判断（可以加表达式）

#else 否则的情况

#elif 多个条件判断时使用

#endif 结束条件编译区块

宏是替换, 与typedef不一样:

typedef OldName NewName;

typedef struct Name{......} newName;

typedef char\* String; 指针新名字String

typedef struct Node{…… } Node, \*NodePtr; 结构体指针NodePtr如同struct Node\*, 而 Node 如同struct Node

typedef int(\*FuncPtr) (int, char, ...); 该结构的函数指针的新名字为FunPtr