嵌入式笔试测试卷

1. 选择题

* **D**

mkdir [选项] [目录]

-m —mode=模式，建立目录的时候同时设置目录的权限。

-p —parents 若所建立的上层目录目前尚未建立，则会一并建立上层目录。

-v —verbose 每次创建新目录都显示信息。

-h —help 帮助信息。

* **A**

如果有一个这样的表达式：cond1 && cond2 || cond3 这句代码会怎样执行呢？

1、cond1为假，那就不判断cond2了，接着判断cond3。

2、cond1为真，这又要分为两种情况：

​ 2.1 cond2为真，这就不需要判断cond3了。

​ 2.2 cond2为假，那还得判断cond3。

fork调用的一个奇妙之处在于它仅仅被调用一次，却能够返回两次，它可能有三种不同的返回值：

1、在父进程中，fork返回新创建子进程的进程ID。

2、在子进程中，fork返回0。

3、如果出现错误，fork返回一个负值（题干中说明了不用考虑这种情况）。

在fork函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。

在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程。

很明显fork() &&　fork() || fork()创建了4个新进程。

总结：

第一行fork生成1个新进程。

第二行的三个fork生成4+4=8个新进程。

第三行的fork会生成10个新进程(这是因为前面总共有10个进程,调用一次fork生成10个新进程。

所以一共会生成1+8+10=19个新进程。

* **C**

排除法， 看个位。最后的结果446C个位为C，因此，A，B可以排除。

假设为14进制，（A \* 4 ）%14= 12，结果正好为C。因此，答案为14进制。

* **B**

通常情况下，二维数组的每一行分别使用一个字符串进行初始化。 例如：

char c[3] [8]={{"apple"},{"orange"},{"banana"}};

等价于：

char c[3][8]={"apple","orange","banana"};

A:应改为char b[3][2] ={"d","e","f"};

C:应改为char b[2][3][2] ={{"d","e","f"},{"a","b","c"}};

D:应改为 char b[2][2]={"d","e"};

* **B**

32位操作系统中

int：4字节

unsigned long long：8字节

char ：1字节

short int：2字节

注意和64位操作系统的区别：64位系统中，指针变量和long以及unsigned long 都占8个字节，其他的和32位系统一样

* 简单题

**(1)**6

首先，定义了一个整型数组a，包含6个元素。

&a表示数组a的地址，它是一个指向整型数组的指针。

(&a+1)表示数组a的下一个地址，因为&a是指向整型数组的指针，所以(&a+1)指向数组a之后的内存位置。

((int\*)(&a+1)-1)将(&a+1)的地址类型转换为整型指针，并减去1，得到数组a的最后一个元素的地址。

\*((int\*)(&a+1)-1)通过解引用操作符\*访问最后一个元素的值，即6。

最后，printf函数将6作为整数打印出来。

所以，这段程序的输出结果是6。

**(2**)1. 常量指针（Pointer to Constant）：指针指向的数据不能被修改，但指针本身可以指向其他数据。

**const int \*ptr1**;

在这个例子中，`ptr1`是一个指向整数常量的指针。你不能通过`ptr1`来改变它所指向的整数的值，但你可以改变`ptr1`所指向的地址。

2. 指针常量（Constant Pointer）：指针本身的值（也就是它所指向的地址）不能被修改，但指针指向的数据可以被修改。

**int \*const ptr2;**

在这个例子中，`ptr2`是一个常量指针，指向一个整数。你不能改变`ptr2`所指向的地址，但你可以改变`ptr2`所指向的整数的值。

总的来说，常量指针和指针常量的主要区别在于是否可以改变指针本身的值和指针所指向的数据。常量指针不允许通过指针来改变数据，但可以改变指针所指向的地址。指针常量不允许改变指针所指向的地址，但可以改变指针所指向的数据。

**(3)**

#ifndef HEADER\_FILE\_NAME\_H

#define HEADER\_FILE\_NAME\_H

// 头文件内容

#endif

#pragma once

// 头文件内容

直接放置在头文件的开头。这个指令告诉编译器只包含一次该头文件，避免了重复包含的问题。

这样可以防止头文件被重复包含。

**(4)**

要设计宏定义STR(x)，将USART\_RATE转换成字符串并打印出来，可以使用宏的字符串化操作符#。

#define USART\_RATE 115200

#define STR(x) #x

printf("usart rate = %s\n", STR(USART\_RATE));

在这个宏定义中，#操作符将宏参数x转换为一个字符串。当我们使用STR(USART\_RATE)时，它会被替换为"USART\_RATE"，然后在printf函数中用%s来打印这个字符串

**(5)**

首先，我们需要理解\*(buff+1)[1]的含义。buff+1表示buff数组的第二个元素的地址，即&buff[1]。然后，\*(buff+1)表示取出该地址处的值，即buff[1]。由于buff[1]是一个字符串，所以可以看作是一个字符数组。最后，[1]表示取该字符数组的第二个元素。

在这个例子中，buff[1]是字符串"int"，然后[1]表示取出字符数组"int"的第二个字符，即字符'n'。

纠正：如果表达式为(\*(buff+1))[1]那么结果确实为 n

但是在本题中最终的结果为d

上述分析没问题但是 \*(buff+1)[1] 系统编译时等价于\*((buff+1)[1])

即buff[1]处(buff+1)结果就等价于 buff[2]

**(6)**

下面的代码输出结果为">6"。

这是因为在表达式`(a+b>6)`中，变量`a`被声明为无符号整型(`unsigned int`)，而变量`b`被声明为有符号整型(`int`)。当进行运算时，有符号整型会被自动转换为无符号整型，这种转换被称为整型提升。

在这个例子中，变量`b`的值为-20，当它与无符号整型`a`相加时，会发生整型提升。整型提升将`b`的值转换为一个大于等于0的无符号整数。

所以，`(a+b)`的结果为6 + (-20) = -14，但由于整型提升，它会被转换为一个无符号整数。最终的比较表达式为`(unsigned int)(-14) > 6`，这个比较结果为假，即`(a+b>6)`为假。

根据条件运算符（三元运算符）的规则，如果条件为真，则执行冒号前的表达式，否则执行冒号后的表达式。在这个例子中，`(a+b>6)`为假，所以执行`puts("<=6")`，输出结果为"<=6"。

大题一

#include <stdio.h>

#include <string.h>

void removeSameCharacter(char\* s)

{

int hash[256] = {0}; // 创建一个大小为256的哈希表，用于存储ASCII字符

char result[10000 + 1]; // 结果字符串，多出的一个字符用于存储字符串结束标志'\0'

int index = 0; // 结果字符串的索引

for(int i = 0; i < strlen(s); i++)

{

if(hash[s[i]] == 0)

{ // 如果字符没有出现过

hash[s[i]] = 1; // 标记字符已经出现过

result[index++] = s[i]; // 将字符添加到结果字符串中

}

}

result[index] = '\0'; // 添加字符串结束标志

strcpy(s, result); // 将结果字符串复制回原字符串

}

int main()

{

char str[10001] = "abbbcacacdc";

removeSameCharacter(str);

printf("%s\n", str); // 输出结果

return 0;

}

大题二

#include <stdio.h>

int main()

{

unsigned int x = 0x76543210;

char \*c = (char \*)&x;

/\*由于char\*指针只能访问地址中的一个字节，因此\*c将给出x的最低有效字节。

如果系统是小端（即最低有效字节存储在最小的地址），即\*c将等于0x10。

如果系统是大端（即最低有效字节存储在最大的地址），即\*c将等于0x76\*/

/\* 小端系统中，最低有效字节位于最小的地址 \*/

if (\*c == 0x10)

{

printf("小端系统\n");

}

else

{

printf("大端系统\n");

}

return 0;

}

大题三

#include <stdio.h>

/\* 冒泡排序算法 \*/

void bubbleSort(int \*a, int n)

{

for (int i = 0; i < n - 1; i++)

{

for (int j = 0; j < n - i - 1; j++)

{

if (a[j] > a[j + 1])

{

// 交换相邻的两个元素

int temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

}

/\* 查找中位数 \*/

int find\_median(int \*a, int n)

{

bubbleSort(a, n); // 使用冒泡排序对数组进行排序

if (n % 2 == 0)

{

return (a[n / 2 - 1] + a[n / 2]) / 2;

}

else

{

return a[n / 2];

}

}

int main()

{

int data[] = {5, 3, 1, 4, 2};

int n = sizeof(data) / sizeof(data[0]); //n为data数组内数据的个数

int median = find\_median(data, n);

printf("中位数为：%d\n", median);

return 0;

}