Solution of Discussion02

DrinkLessMilkTea

2025 年 3 月 10 日

1 Pre-Check

1.1

True, C 语言确实是默认传值调用的语言, 如果想要实现引用调用, 则需要将参数的地址传入 (或者指向参数的指针)

1.2

指针是存储另一个变量地址的变量, 通过指针可以访问到对应地址所保存的变量, , 指针指向的是某个变量的地址, 数组类似于指向首元素的指针. 区别在于, 数组名不是变量, 不能进行变量相关的操作, 如赋值, 算术运算等.

1.3

如果对某个非指针变量进行解引用, 解释器会将变量的值当做地址访问, 企图在地址上取出变量, 如果变量的值正好是一个合法的地址, 则可以将对应地址的数据取出, 否则就会发生非法内存访问的段错误.

如果对某个非指针变量进行 free 操作, 同理, 解释器会首先确定变量值对应的地址是否合法, 如果合法, 则会将对应的地址释放, 否则如果释放不由 malloc 或 realloc 分配的内存就会产生非法释放错误.

1.4

当程序需要使用一块动态变化的内存空间的时候需要使用 heap, 也就是内存大小无法在编译是就确定, 或者需要使用较大的内存空间 stack 无法

存储的时候, 当数据需要在多个函数之间共享的时候也可以用 heap, heap 会向上增长, stack 会向下增长.

2 C

2.1

pp 的值是 p 的地址, 也就是 0xF9320904; *pp 的结果是 pp 指向的变量的值, 也就是 p 的值, 即 0xF93209AC; **pp 可以表示为 *(*pp) = *(p), 也就是 p 指向的变量的值, 即 0x2A.

2.2

2.2.1

这个函数接收的是一个数组和需要计算的元素个数 n, 返回数组前 n 个元素的累加和

2.2.2

这个函数接收的是一个数组和需要计算的元素个数 n, 返回数组后 n 个元素中 0 的个数的相反数

2.2.3

函数执行时, 每一步后 x 和 y 的值如下:

$$x = x \oplus y, y = y \tag{1}$$

$$x = x \oplus y, y = x \oplus y \oplus y = x \tag{2}$$

$$x = x \oplus y \oplus x = y, y = x \tag{3}$$

但是由于是传值调用, 所以原先的 x 和 y 都没有改变

2.2.4

同或可以表示为对异或的结果取反, 即 `~(x ^ y)`

3 Programming with Pointers

3.1

3.1.1

```
int swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  return;
}
```

3.1.2

```
int mystrlen(char* str) {
  if (str == NULL ) return 0;
  int len = 0;
  char* cur = str;
  while(*cur != '\0') {
    len ++;
    cur ++;
  }
  return len;
}
```

3.2

3.2.1

sizeof 会返回参数所占的字节数, 而不是元素数, 所以正确表示数组元素数应该是 `sizeof(summands) / sizeof(int)`

3.2.2

对于字符串来说, 字符串终止符和字符数组的长度无关, 有长度为 n 的字符数组不代表字符串有 n 个字符, 所以应该修改函数如下:

```
void increment(char* string) {
  for (int i=0;string[i] != '\0';i++) {
    string[i] ++;
  }
}
```

3.2.3

函数的逻辑是将字符串 src 复制到字符串 dst 上, 函数有以下几个潜在的问题:

首先, 字符串 src 的长度和 dst 的长度均未做限制, 如果 src 的长度超过 dst , 则会发生访问未知内存的错误; 其次, 对 src 和 dst 都没有判空检查, 如果其中之一为 NULL, 则对 NULL 解引用也会发生错误; 最后, 复制完成后 dst 和 src 都会指向字符串的结尾, 不会回到开头. 修改后的版本如下

```
void copy(char* src, char* dst) {
  if (src == NULL || dst == NULL) {return;}
  int len = min(strlen(src), strlen(dst));
  for (int i=0;i<len;i++) {
    dst[i] = src[i];
  }
  dst[len] = '\0';
}
```

3.2.4

函数的整体实现没有错误, 但是在定义被替换字符串指针 replaceptr 的时候没有正确指定类型, 正确的定义语句应该为

```
char *srcptr, *replaceptr
```

4 Memory Management

4.1

(a) 静态变量存储在 static 区, 在整个程序运行期间都会存在

(b) 局部变量存储在 stack 区, 跟随定义它的函数一起, 在函数返回时失效

(c) 全局变量存储在 static 区, 程序运行期间有效

(d) 局部常量会存储在 stack 区, 全局常量存储在 static 区, 一些数字常量有时候也会直接嵌入在 code 区

(e) 代码段的机器指令会存储在 code 区

(f) malloc 分配的内存会在 heap 区

(g) 字符串字面量会存储在 static 区, 字符数组会存储在 stack 区, 有时候根据编译器的选择也会直接嵌入在 code 区

4.2

(a) `int *arr = (int*)malloc(sizeof(int) * k)`

(b) `char *str = (char*)malloc(p + 1)`

(c)

```
int **mat = (int**)malloc(sizeof(int*) * n);
for(int i=0;i<n;i++) {
  mat[i] = (int*)malloc(sizeof(int) * m);
}
for(int i=0;i<n;i++) {
  for(int j=0;j<m;j++) {
    mat[i][j] = 0;
  }
}
```

4.3

buffer 字符串定义了但是没有初始化, 如果输入的字符串长度小于 10, 这在进行复制的时候会发生未知的错误, 复制到了未知的字符, 同时终止符也会提前被复制

如果用户输入的长度大于 63 个字符, 则会导致其他区域的内存被覆盖 (gets 函数的弊端)

4.4

```
void prepend(struct ll_node** lst, int value) {
  struct ll_node* newNode = (struct ll_node*)malloc(sizeof(struct ll_node));
  newNode->first = value;
  newNode->rest = *lst;
  *lst = newNode;
}
```

4.5

```
void free_ll(struct ll_node** lst) {
  struct ll_node *before = *lst->rest, back = *lst;
  while(before != NULL) {
    free(back);
    back = before;
    before = before->rest;
  }
  free(back);
}
```