对于 实验 01 C-String 中 30 个问题的解答

Aurthor: JamesNULLiu
Time: 2022/9/10

问题 1 :

指针 dest 和 source 分别存放复制目标字符串和复制源字符串的第一个元素的地址. 由后置 ++ 运算符自增, 每次对自增前的地址分别进行 dereference 操作和 assignment 操作; 直到将 source 所指字符串从左至右第一个 '\0' 字符赋值给 dest 所指字符串相应位置, 赋值操作本身的布尔值为 false, 循环终止. 此时 source 和 dest 分别存放 '\0' 所在位置的下一个元素地址.

问题 2 :

由于自增, dest 在循环结束后不再存放字符串首元素地址, 因此需要返回存放了首元素地址的指针 temp.

问题 3 :

循环结束时, str1 存放了其所指字符串从左至右第一个值为 '\0' 的元素地址.

问题 4 :

"The (plain) char type uses one of these representations. Which of the other two character representations is equivalent to char depends on the compiler." C++ Primer 5th Edition, Stanly B. Lippman, Josee Lajoie, Barbara E. Moo.

也就是说, 根据编译器的不同, 对字符类型的存储方式既可能使用 unsigned char 也可能使用 signed char. 假设在某个编译环境下编译器使用 signed char 存储字符, 而某个字符 ch 的值恰巧为 $(1000000)_b$, 那么从数值角度说 ch 的值是小于 '\0' 的. 可以用以下代码检验 :

```
int main()
{
    signed char ch = 0b10000000;
    signed char ze = '\0';
    cout << (ch > ze);
    cin.get();
}
```

输出结果为: 0.

因此本例中利用指针 p 和 q 显示地将 char 类型转化为 unsigned char 类型, 避免了上文所说的计算错误.

另外, 值得一提的是, 如果仅仅是 ASCII 码表内字符, 不会出现这个问题. 因为 ASCII 码一共只包含 127 个字符, 也就是数值最大也仅为 $(01111111)_b$, 无论以 unsigned char 类型储存还是 signed char 类型储存, 最高位都不可能为 1.

问题 5:

循环终止情况:

a. p 当前储存的内存地址中存放了 '\0', i.e., str1 的长度小于 str2 的长度.
b. q 当前储存的内存地址中存放了 '\0', i.e., str2 的长度小于 str1 的长度.
c. p 当前存储的内存地址中的值与 q 当前存储的内存地址中的值不相等, i.e., p 和 q 分别指向了 str1 和 str2 从左至右第一个不相等的字符的位置.

问题 6:

'\0' 的值为 0, 起到了标识 C 风格字符串结束位置的作用, 相当于 C++ 风格字符串的迭代器 std::string::end(). 本例中还起到了数值大小比较的作用.

问题 7:

显然 C/C++ 不支持诸如 a<x<c 这类的语法.

问题 8:

减去 '0' 是为了得到数字字符所对应的真实数值, 例如 '1' – '0' = 1. 得到该数值后便可以更方便地实现字符串转数字算法.

问题 9:

当已有两个字符串 str1 和 str2 并调用 GetHeapString(str1, str2) 时, 需要扩容的对象是 str1. 传入引用使得在函数内部可以直接对 str1 进行操作. 如果不传入引用, 最后操作的对象实际上是一个在 parameter list 中创建的, 复制了 str1 所储存的地址的局部指针变量,

问题 10:

因为 delete 操作不是赋值 NULL 操作, 释放完的内存是不可读取的, 会导致外部传入的指针引用成为不可读取的状态; 不能对该快内存进行任何操作和判断. 赋值为 NULL 规范化了指针的状态, 在数值上等于 0.

问题 11:

传递二级指针的引用是因为本函数的实现逻辑在于将 source 所指字符串组的字符一一赋值到 dest 所指字符串组. 传递引用结果上直接改变了外部传入的 dest 实参.

问题 12:

在 SortString.cpp 的 Line 9 中, dest[0] 这个指针申请了 n * NUM 个字符大小的空间, 其中 n 是 source 中有多少个字符串; NUM 是每个字符串最大容量. 所以本质上 dest[0] 把所有需要的空间都申请完了. 本循环的目的在于将 n * NUM 的空间分割成 n 个 NUM 个字符大小的空间, 分别由指针 dest[0, 1, ···, n-1] 指向这些分割的空间的首元素地址.

这样做虽然繁琐, 但是模拟了二维数组按行线性存储规律. 如果分别 new 空间, 依照堆的存储机制, 不会 new 出内存线性排布的字符串.

问题 13:

由于不确定第二参数 source 中的字符串是否符合 C 风格字符串规则 (末尾存在 '\0' 标识字符串结束), 因此不能使用库函数 strcpy(). 这里使用库函数 strncpy() 直接限定数量复制最大容量 NUM 个字符.

问题 14:

作用是在末尾添加 '\0' 用以标识字符串结束.

问题 15:

作用是直接操作外部指针而非创建一个指针副本.

问题 16:

作用是找到字符串数组的各个字符串的首元素地址最小的那个字符串.

这么做的原因在问题 25 有说明.

问题 17:

此处释放的是由二级指针指向的指针数组, 而不是指针所指的字符串.

问题 18:

函数内部定义的局部变量在未初始化的情况下是随机值, 而后续调用的函数都依赖对 NULL 的判断而终止运行, 因此必然会出错.

问题 19:

会变化. 函数 GetHeapString() 每次 new 出来的地址不一样; 而函数 FreeHeapString() 每次会将 str 赋值为 NULL.

问题 20/21:

首先, strA, strB, strC, strD, strE 都是函数中定义的指针变量, 它们自己所在的位置都是内存栈. 需要区分的是它们储存的地址在内存中的什么区域.

strA 的类型为 char[15][20], 也就是二维数组类型. 数组类型数组名的地址是数组首元素的地址; 而以数组形式创建的字符串存放的位置是内存栈. 因此 strA 所储存的地址是一个处于内存栈的地址.

strB 的类型为 char *[15], 是一维指针数组类型, 每个元素是个指针. 以指针形式创建的字符串, 指针所储存的地址是一块文字常量区的地址. 例如: const char *p = “aaa”, p 这个指针存放的是文字常量区的地址; 而 p 本身是在内存栈中创建的 --- 也就是说 p 自己所在的地址是内存栈的地址. 又因为 strB 的一维指针数组, 任何数组的数组名都是数组首元素的地址, 所以 strB 所储存的实际上是 strB[0] 这个指针自己的地址. 这是一块在内存栈中的地址.

strB0 与 strB 同理, 它的值是一个在内存栈中的指针自己的地址. 值得一提的地方在于, strB[0] 这个指针所储存的文字常量区中的地址和 strB0[0] 所储存的地址是一致的, 这是因为它们都指向文字常量区中相同的一串文字.

strC, strD, strE 经过后续的 new 操作, 结果上都指向内存堆中的地址.

结构图这里就省略了, 上文对特殊的结构已有详尽的分析.

问题 22:

strA 的类型是 char[15][20]. 参照我夏季实训所写的论文 *cpp 数组类型对象探究及冒泡排序选择排序快速排序算法的分析优化* 可以知道, char[N] 类型与 char * 类型同级可转换; char[N][M] 类型与 char (*)[M] 类型同级可转换. 不存在从 char[N][M] 类型到 char ** 类型的转换.

其次, 就算可转换, 由于 BubbleB() 的实现依靠地址的交换, strA 存放的地址可能在交换后不再在首元素; 此时 strA 无法指代整个字符串组.

问题 23:

根据问题 20/21 的分析, 我们知道 strB 中每个指针元素都指向文字常量区中的文字. strcpy() 函数无法将文字常量区中的字符串作为第一个参数, 因此会报错.

作为 Plan B, 这里可以改用 std::swap() 操作文字常量区中的内存.

问题 24:

strC 的创建方式在问题 12 中已做分析. strC 本身是个二级指针, 各个指针元素储存的就是各个字符串的首地址. BubbleB() 做的事情就是交换这些指针储存的地址信息, 执行完成后 strC[0,1,⋯,n-1] 这些指针按照顺序指向排完序的字符串首地址. strC[0] 的内存信息改变与否取决于排完序后其指向的字符串有无改变.

问题 25:

strC 的创建方式在问题 12 中已做分析. strC 由 BubbleB() 排序的过程在问题 24 已经分析. 问题 16 中提到了找到地址最小的那个指针, 是因为虽然排完序后 strC[0,1,⋯,n-1] 这些指针存放的地址发生了变化, 但是 new 出来的整个空间还是按照线性紧密储存的. 想要一劳永逸地用 delete[] , 必须找到这些空间中开头的(最小的地址). 因此要传入 true 以另一种方式释放内存.

问题 26:

BubbleA() 本质上是不改变地址而交换地址中的值. 这里还要注意的是, strC 经过 GetString0() 重新构造后, 任意指针 strC[x] 所储存的地址信息的堆中的地址, 所以才可以用 BubbleA() 中的 strcpy() 函数. 如果单纯赋值字符串常量而指向文字常量区的字符串, 那么是不可以用 strcpy() 函数的. 因此结果上 strC[0] 自己这个指针没变化, 它储存的地址信息也没变化, 但是地址上的内存可能被 strcpy() 改写了.

问题 27:

GetString1() 是 GetString0() 的简化版, 没有申请线性的空间用以存放所有字符串. 由于 BubbleA() 和 BubbleB() 都没有对线性存储方式的依赖, 因此与 strC 一样, 对于 strD, 两种函数都可以使用.

问题 28:

GetString2() 在 new 空间时仅仅 new 了目标字符串长度大小的空间; 超出该空间的内存都不可读. 由于 BubbleA() 利用 strcpy() 复制值, 可能将长度较长的字符串赋值给较短的字符串, 又由于较短字符串的后续空间不可读而程序报错.

问题 29:

可以. 因为在 SortString.h 中对于 FreeString() 的第三个参数给的默认值就是 false. 传不传入 false 在结果上是等价的.

问题 30:

ture 和 flase 的区别就在效率上. 当然可以对 strc 中的一个个指针分别使用 delete 来释放内存. 更快捷的方式就是问题 25 直接 delete[] .