Lab03

Task 1: Array, and Functions

如果我们要从键盘读取5个整数并保存在数组中，然后将这些数字打印出来。以下程序可以作为参考:

- `int arr[5]` 可以创建一个长度为5，类型为 int 的数组
- `scanf("%d", &arr[0])` 可以用将键盘输入读取到 arr[0]

参考程序:

```c
#include <stdio.h>

#define ARRAY_LEN 5

int main(int argc, char** argv) {

    int arr[ARRAY_LEN];

    printf("Enter the elements into the array: ");
    for (int i = 0; i < ARRAY_LEN; i++) {
        scanf("%d", &arr[i]);
    }

    printf("\n The elements of the array are: ");
    for (int i = 0; i < ARRAY_LEN; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Task 1.1

修改上述程序，将数组的读取和显示包装成可反复调用的子程序

> 备注：注意array在函数中的传递。

```c
void read_array(int* arr, int len) {
    int ret;
    printf("Enter the elements into the array: ");

    // TODO

    return;
```

```c
}

void display_array(int* arr, int len) {
    printf("\nThe elements of the array are: ");

    // TODO

    printf("\n");
    return;
}

int main(int argc, char** argv) {

    int arr[ARRAY_LEN];
    read_array(arr, ARRAY_LEN);
    display_array(arr, ARRAY_LEN);

    return 0;
}
```

Task 1.2

编写子程序，输出数组中元素的和

```c
int sum_array(int* arr, int len) {
    int sum = 0;
    /** 遍历求和 **/
    // TODO

    return sum;
}
```

Task 1.3

编写子程序，统计数组中偶数的个数

```c
int count_even(int* arr, int len) {
    /** 计算偶数的个数 **/
    /** 将cnt初始化为0 **/
    int cnt = 0;

    // TODO

    return cnt;
}
```

Task 1.4

编写子程序，找到数组中最大和最小的元素，并打印他们在数组中的位置

```c
void array_max(int* arr, int len) {
    /** 求最大 **/
    /** res 初始化为arr[0] **/
    int res = arr[0];

    /** 如果找到更大的，将res修改成找到的值 **/

    // TODO

    /** 输出一串提示词和找到的最大值 **/
    printf("The maximum element of the array is: %d at index ", res);
    /** 将所有等于res的值的下标输出 **/
    for (int i = 0; i < len; i++) {
        if (arr[i] == res) {
            printf("%d, ", i);
        }
    }
    printf("\n");
}

void array_min(int* arr, int len) {
    /** 求最小，同上 **/
    int res = arr[0];

    // TODO

    printf("The minimum element of the array is: %d at index ", res);
    for (int i = 0; i < len; i++) {
        if (arr[i] == res) {
            printf("%d, ", i);
        }
    }
```

```
    printf("\n");
}
```

Task 1.5

编写子程序，将保存的数组逆序，并打印逆序后的数组内容

- 提示:
  - 注意要修改数组在内存中的内容，而非简单的用逆序将数组打印。
  - 可以使用额外的变量帮助交换。

```c
void reverse_array(int* arr, int len) {
    /** 逆序 **/
    /** 将tmp用于交换元素 **/
    int tmp = 0;
    /** 记录前方下标 **/
    int head_idx = 0;
    /** 记录后方下标 **/
    int tail_idx = len - 1;

    while (head_idx < tail_idx) {
        /** 利用tmp交换变量 **/

        // TODO

        /** 分别改变下标的记录值 **/
        ++head_idx;
        --tail_idx;
    }
    printf("The array is reversed.\n");
}
```

Task 2: Functions and Scopes

C 中变量按其作用域分，可分为局部变量和全局变量，具体的解释为：

- **局部变量（内部变量）（Local Variable）**：在定义它的函数/模块内有效，但是函数返回后失效；
- **全局变量（外部变量）**：在所有源文件内均有效。
  - 在同源文件的函数中使用全局变量，需要将全局变量提前声明；
  - 同时在不包含全局变量定义的不同源文件需要用extern关键字再次声明。（了解）

优先次序：

- **局部变量优先原则**。如果在该函数内定义了一个与之前定义的全局变量同名的变量，那么该同名局部变量就会在函数内部屏蔽全局变量的影响。即当局部变量与全局变量同名时，在局部变量的作用范围之内，全局变量不

起作用。

默认值：

- 全局变量是有默认值的（引用类型的变量默认值都是null，基本类型的变量默认值则不一样，int型变量默认值是0）
- 局部变量没有默认值。

赋值操作：

- 全局变量可以在函数外定义并初始化，但不能进行赋值操作。

2.1 局部作用域-嵌套块

> **Test1**

以下代码：

```c
#include <stdio.h>
int main()
{
    int a = 5;
    {
        // a+10
        a += 10;
        printf("a=%d", a);
    }
    return 0;
}
```

输出为

```
a=15
```

- 在 C 语言中，用{}分隔代码块。左花括号和右花括号分别表示块的开始和结束。
- main() 函数有一个整数变量 a，它在代码块外部被初始化为 5。
- 代码块内部有一行代码将 10 添加到变量 a。
- 我们看到，{}代码块能访问a，并进行相关操作

> **Test2**

以下代码：

```c
#include <stdio.h>
int main()
{
    int a = 5;
    {
        int inner_a = 10;
    } 到此处，inner_a已经消亡
    printf("inner_a =%d", inner_a); //! line 8
    return 0;
}
```

- 在这个程序中，`int main()`函数中，在代码块外部有一个整数变量`a`。
- 另一个变量`inner_a`在代码块内部初始化。
- 我们试图在代码块外部访问和打印代码块内部的`inner_a`的值。

如果尝试编译上面的代码，会注意到它没有成功编译，收到以下错误消息：

```
8 error: 'inner_a' undeclared (first use in this function)
```

这是因为变量`inner_a`是在代码块内部声明的，其作用域仅限于代码块内部。换句话说，它对于代码块内部是本地的，不能从代码块外部访问。

基于上述观察，发现变量局部作用域的通用原则：

```c
{

    /*OUTER BLOCK*/

  {

    //此块开始之前的代码块外部的内容可以在这里被访问

    /*INNER BLOCK*/

  }

  //此处无法访问代码块内部的内容

}
```

2.2 局部作用域-不同的块

在前面的示例中，我们了解了如何无法从代码块外部访问嵌套代码块内部的变量。

现在，我们来了解在不同块中声明的变量的局部作用域。

> **Test3**

```c
#include <stdio.h>
int main()
{
    int a = 5;若想用到该参数，需要传参到sub_func()
    printf("a=%d", a);
    sub_func();
    return 0;
}
void sub_func()
{
    printf("a=%d", a);
}
```

在上面的例子中，

- 在 `main()` 函数中声明整型变量 `a` 。
- 在 `main()` 函数内部，打印出 `a` 的值。
- 另一个函数 `sub_func()` 尝试访问和打印 `a` 的值。
- 由于程序从 `main()` 函数开始执行，因此在 `main()` 函数内部调用了 `sub_func()`。

现在编译并运行上述程序，我们收到以下错误消息：

```
11   error: 'a' undeclared (first use in this function)
```

我们注意到，在第 11 行，函数 `sub_func()` 尝试访问在 `main()` 函数中声明和初始化的 `a` 变量。

因此，变量 `a` 的作用域被限制在 `main()` 函数中，并且被称为 `main()` 函数的局部变量。

我们可以将这种局部作用域的概念一般性地表示如下：

```c
{
    /*BLOCK 1*/
    // 此处无法访问 BLOCK 2 的内容
}

{
    /*BLOCK 2*/
    // 此处无法访问 BLOCK 1 的内容
}
```

2.3 全局作用域

> Test4

```
#include <stdio.h>
int global_a = 5;  //! declaration of global variable

int main()
{
    printf("global_a = %d\n", global_a);
    return 0;
}
```

此时程序运行通过，程序中`global_a`为全局变量，在函数外进行定义并初始化。

```
global_a = 5
```

> Test5

```
#include <stdio.h>
int global_a = 5;  //! declaration of global variable
global_a = 10; //! 赋值  只能声明和初始化

int main()
{           函数内可以赋值全局变量
    printf("global_a = %d\n", global_a);
    return 0;
}
```

此时程序运行错误。全局变量在函数体外只能进行初始化，不能进行赋值运算。

```
error: redefinition of 'global_a'
```

> Test6

```c
#include <stdio.h>
int global_a;  //! declaration of global variable, without initialization
global_a = 10; //! initialization

int main()
{
    printf("global_a = %d\n", global_a);
    return 0;
}
```

只声明了没有初始化

此时程序运行通过，C语言执行过程中，对代码进行了优化，把程序中的`int global_a; global_a =10;`优化为一句话：`int global_a =10`；这就是**初始化**。

> Test7

```c
#include <stdio.h>
int global_a = 5;  //! declaration of global variable
void sub_func();//! declaration of function 'sub_func'
int main()
{
    printf("global_a can be accessed from main() and its value is global_a = %d\n", global_a);
    //call sub_func
    sub_func();
    return 0;
}

void sub_func()
{
  printf("global_a can be accessed from sub_func() as well and its value is global_a = %d\n", global_a);
}
```

函数头

在上面的例子中，

- 在函数`main()`和`sub_func()`之外声明变量`global_a`。
- 我们尝试访问`main()`函数中的`global_a`，并打印其值。
- 我们在`main()`函数中调用`sub_func()`函数。
- 函数`sub_func()`也尝试访问`global_a`的值，并将其打印出来。

该程序编译没有任何错误，输出如下所示：

```
global_a can be accessed from main() and its value is global_a =5
global_a can be accessed from sub_func() as well and its value is global_a =5
```

在这个例子中，有两个函数——`main()`和`sub_func()`。

但是，变量 `global_a` 不是程序中任何函数的局部变量。这种对任何函数都不是局部的变量被称为具有全局作用域，称为**全局变量**。

> **Test8**

```c
#include <stdio.h>
int global_a = 5;  //! declaration of global variable
void sub_func();//! declaration of function 'sub_func'

int main()
{
    printf("global_a can be accessed from main() and its value is global_a = %d\n", global_a);

    int global_a = 10; 定义了一个函数内的（局部）同名变量，优先级更高
    printf("global_a redefined global_a = %d\n", global_a);

    //call sub_func
    sub_func();
    return 0;
}

void sub_func()
{
    printf("global_a can be accessed from sub_func() as well and its value is global_a = %d\n", global_a);
}               此处是全局变量
```

局部变量优先原则:

```
global_a can be accessed from main() and its value is global_a = 5
global_a redefined global_a = 10
global_a can be accessed from sub_func() as well and its value is global_a = 5
```

但是新定义的`global_a = 10`并没有影响到接下来调用的`sub_func()`。

全局变量作用域的原理可以总结如下：

```
//所有全局变量都在这里声明
function1()
{
    // 所有全局变量都可以在 function1 中访问
}
function2()
{
    // 所有全局变量都可以在 function2 中访问
}
```

Task3: Palindrome Number

> “回文”是指正读反读都能读通的句子，它是古今中外都有的一种修辞方式和文字游戏，如“我为人人，人人为我”等。回文串指的是正的读和反的读是一样的字符串，例如 "aba"，"ccbbcc"。
>
> 参考

Task3: 寻找最长回文字符串

从键盘读入一个字符串，寻找并输出最长的回文子串。

Task 3.1

字符数组：如果输入a b c d，会把空格也读进去
数字数组：1 2 3 4，**空格可以表示分割**

结合Task1，进行字符数组的读取和显示，包装成可反复调用的子程序

备注：

- 编写子程序，从键盘读入指定长度为n的字符串（不包括\n）
- 编写子程序，打印指定长度为n的字符串（不包括\n）

Task 3.2

暴力解法思路：

- 列举所有的子串，判断是否为回文串，保存最长的回文串。

编写判断字符串为回文字符串的子函数。

```
//! 判断是否是 Palindrome array
int isPalindromic(char* arr, int len) {
    for (int i = 0; i < len / 2; i++) {
        //--TODO--
        if (arr[i] != arr[len - i - 1]) {
            return 0;
        }
    }
```

```
    return 1;
}

void longestPalindrome(char* arr, int len) {
    int longest_start_index = 0; // 最长回文字符串开始的索引
    int longest_length = 0; // 最长回文字符串的长度
    int current_length = 0;  // 当前判断的子串的长度

    for (int i = 0; i < len; i++){
        for (int j = i + 1; j <= len; j++) {
            current_length = j-i+1;
            // TODO
        }
    }

    display_array(arr + longest_start_index * sizeof(char), longest_length);
    return;
}

```

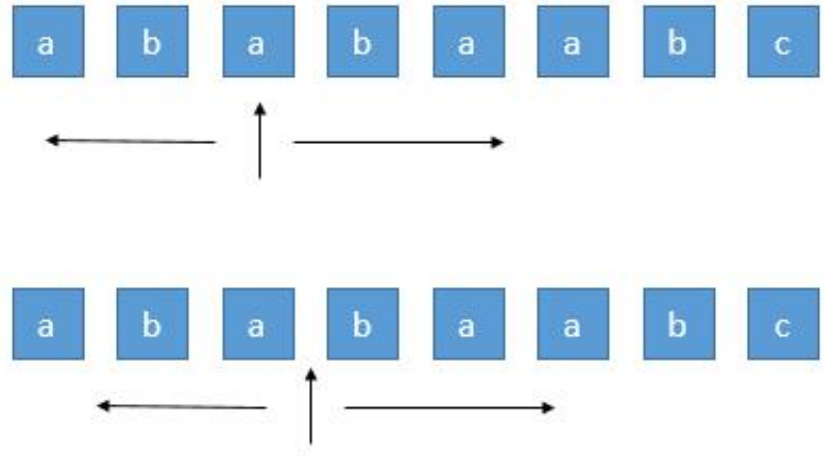
注：时间复杂度：

- 两层 for 循环 $\mathcal{O}(n^2)$
- for 循环里边判断是否为回文，$\mathcal{O}(n)$，
- 所以时间复杂度为 $\mathcal{O}(n^3)$。

Task 3.2

扩展中心法

- 我们知道回文串一定是对称的，所以我们可以每次循环选择一个中心，进行左右扩展，判断左右字符是否相等即可。



由于存在奇数的字符串和偶数的字符串，所以我们需要从一个字符开始扩展，或者从两个字符之间开始扩展，所以总共有 n + n - 1 个中心。

```c
//!  扩展中心法
int expandAroundCenter(char* arr, int len, int left, int right) {
    int L = left, R = right;
    // TODO

    return R - L - 1;
}


int longestPalindrome(char* arr, int len) {

    int longest_start_index = 0, longest_end_index = 0;

    for (int i = 0; i < len; i++) {
        //! 字母本身为中心
        int len1 = expandAroundCenter(arr, len, i, i);

        //!  字母间隙为中心
        int len2 = expandAroundCenter(arr, len, i, i + 1);

        int current_max_length = (len1 >= len2) ? len1 : len2;

        if (current_max_length > longest_end_index - longest_start_index) {
            longest_start_index = i - (current_max_length - 1) / 2;
            longest_end_index = i + current_max_length / 2;
        }
    }
    display_array(arr + longest_start_index * sizeof(char),  longest_end_index - longest_start_index + 1);
    return 0;
}
```

时间复杂度为 $\mathcal{O}(n^2)$。

Homework

探索时间复杂度为线性的方法来寻找回文字符串

例如Manacher's Algorithm。

要求：

1. 提供程序流程框图。
2. 提供源代码，使用CMake。生成的可执行文件命名为main。
3. 提供实验报告，分析说明算法的线性特性。

多做几组，画出函数运行时间关于n**的函数**

4. 感兴趣的同学可以通过更大长度的数组来对比一下三种算法的执行时间（不做要求）。