```c
//
// Created by hengxin on 10/19/22.
//

#include <stdio.h>
#include <stdlib.h>

#define LEN_L 4
#define LEN_R 5

int *Merge(const int L[], int llen, const int R[], int rlen);

int main() {
  int L[LEN_L] = {2, 4, 6, 8};
  int R[LEN_R] = {1, 3, 5, 7, 9};

  int *merge = Merge(L, LEN_L, R, LEN_R);

  for (int i = 0; i < LEN_L + LEN_R; i++) {
    printf("%d ", merge[i]);
  }

  return 0;
}

int *Merge(const int L[], int llen, const int R[], int rlen) {
  int *merge = malloc((llen + rlen) * sizeof *merge);

  int l = 0;
  int r = 0;
  int m = 0;

  while (l < llen && r < rlen) {
    if (L[l] <= R[r]) {
      merge[m++] = L[l];
      l++;
    } else { // L[l] > R[r]
      merge[m++] = R[r];
      r++;
    }
  }

  while (l < llen) {
    merge[m++] = L[l];
    l++;
  }

  while (r < rlen) {
    merge[m++] = R[r];
    r++;
  }

  return merge;
```

```
54 }
```

```c
/**
 * file: pointer.c
 *
 * Created by hengxin on 11/28/21.
 */

#include <stdio.h>
#include <stdlib.h>

int main() {
  /********** On radius **********/
  int radius = 100;

  printf("radius = %d\n", radius);

  // every variable has an address
  // &: address-of operator ("□□□"□□□)
  printf("The address of radius is %p\n", &radius);
  // we have already used the address of a variable before
  // scanf("%d", &radius);

  // radius as a left value; refer to its address (the storage space)
  radius = 200;
  // radius as a right value; refer to its value
  double circumference = 2 * 3.14 * radius;
  printf("radius = %d; circumference = %f\n", radius, circumference);
  /********** On radius **********/

  /********** On ptr_radius1 **********/
  // ptr_radius1 is a variable of type "pointer to int"
  int *ptr_radius1 = &radius;
  // ptr_radius1 is a variable: has its value
  printf("ptr_radius1 = %p\n", ptr_radius1);
  // ptr_radius1 is a variable: has its address
  printf("The address of ptr_radius1 is %p\n", &ptr_radius1);
  /********** On ptr_radius1 **********/

  /********** On *ptr_radius1 **********/
  // IMPORTANT:
  // *ptr_radius1: behaves just like radius
  // type: int; value: the value of radius; address: the address of
radius
  // *: indirection/dereference operator ("□□□□"/"□□□"□□□)
  printf("radius = %d\n", *ptr_radius1);
  // *ptr_radius1 as a right value
  circumference = 2 * 3.14 * (*ptr_radius1);
  // take the address of *ptr_radius1
  // &*ptr_radius1 is the same as ptr_radius1
  printf("The address of *ptr_radius1 is %p\n", &*ptr_radius1);
  // *ptr_radius1 as a left value
  *ptr_radius1 = 100;
  printf("radius = %d\n", *ptr_radius1);
  /********** On *ptr_radius1 **********/
```

```c

    /********** On ptr_radius1 again **********/
    // ptr_radius1 as a left value
    int radius2 = 200;
    int *ptr_radius2 = &radius2;

    ptr_radius1 = ptr_radius2;
    printf("radius = %d\n", *ptr_radius1);

    // ptr_radius1 as a right value
    ptr_radius2 = ptr_radius1;
    printf("radius = %d\n", *ptr_radius2);
    /********** On ptr_radius1 again **********/

    /********** On array names **********/
    int numbers[5] = {0};
    // vs. numbers[2] = {2};
    // numbers++;
    // numbers = &radius;
    int *ptr_array = numbers;
    ptr_array++;
    /********** On array names **********/

    /********** On malloc/free **********/
    // undefined behavior
    // free(numbers);
    /********** On malloc/free **********/

    /********** On const **********/
    // const int * and int const *
    // You cannot modify the value pointed to by ptr_radius3
    // through the pointer (without casting the constness away).
    const int *ptr_radius3 = &radius;
    // *ptr_radius is read-only
    // *ptr_radius3 = 300;
    // You are allowed to do this, but you should not do it!
    int *ptr_radius4 = ptr_radius3;
    *ptr_radius4 = 400;
    printf("radius = %d\n", radius);

    // int * const
    int *const ptr_radius5 = &radius;
    // ptr_radius5 = ptr_radius3;
    *ptr_radius5 = 500;
    printf("radius = %d\n", radius);

    // const int * const
    const int *const ptr_radius6 = &radius;
    // ptr_radius6 = ptr_radius3;
    // *ptr_radius6 = 600;
    /********** On const **********/
}
```

```
 1 # 8-pointer
 2
 3 ## `radius.c`
 4
 5 ## Swap Numbers (`selection-sort.c`)
 6
 7 ## Pointers and Arrays (`selection-sort.c`)
 8
 9 - `(int *arr)`
10 - `numbers[i]`
11 - `&numbers[i]`
12 - `const` in `Print`
13
14 ## Dynamic Memory Management (`selection-sort.c`)
15
16 - `malloc.h` vs. `stdlib.h`
17 - `malloc`
18   - size = 0: implementation-defined
19 - `free`
20   - memory leak (heap)
21   - undefined behaviors
22     - double `free`
23     - `free` non-`malloc`
24     - dereference `free`d memory
25
26 ## `merge.c`
27
28 ## `radius.c`
29
30 - `scanf.c`
```

```c
// Created by hfwei on 2022/10/13.
//

#include <stdio.h>
#include <stdlib.h>

void WrongSwap(int left, int right);
void Swap(int *left, int *right);
void Print(const int *arr, int len);
void SelectionSort(int *arr, int len);

int main() {
  int len = 0;
  printf("Please input the length of the array to sort.\n");
  scanf("%d", &len);

  // void *: the type for a generic pointer (replacing char *)
  // include stdlib.h (not malloc.h which is deprecated) for malloc
  int *numbers = malloc(len * sizeof(*numbers));
  // null pointer: not the same with any non-null pointers
  // #define NULL ((void *) 0); but do not rely on it
  if (numbers == NULL) {
    printf("Error! Memory Not Allocated!\n");
    return 0;
  }

  printf("Please input %d integers.\n", len);
  for (int i = 0; i < len; i++) {
    scanf("%d", numbers + i);
  }

  Print(numbers, len);
  // numbers: the address of the first element of the `numbers` array
  // pass by value: the copy of the address of the first element of the `numbers` array
  SelectionSort(numbers, len);
  Print(numbers, len);

  // avoid memory leak (why memory leak?)
  free(numbers);
  // undefined behavior
  // free(numbers);
  // numbers[5] = 5;
}

void Print(const int *arr, int len) {
  printf("\n");
  for (int i = 0; i < len; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}

```

```
53 // arr: the (copy of the) address of the first element of the `
   numbers` array
54 void SelectionSort(int *arr, int len) {
55   for (int i = 0; i < len; i++) {
56     // find the minimum of numbers[i .. len - 1]
57     // arr[i] is a syntactic sugar for *(arr + i)
58     int min = *(arr + i);
59     int min_index = i;
60     for (int j = i + 1; j < len; j++) {
61       if (*(arr + j) < min) {
62         min = *(arr + j);
63         min_index = j;
64       }
65     }
66
67     // swap arr[i] and arr[min_index]
68     // WrongSwap(arr[i], arr[min_index]);
69     // &(*(arr + i))
70     // Swap(&arr[i], &arr[min_index]);
71     // &arr[i] is the same as (arr + i)
72     Swap(arr + i, arr + min_index);
73   }
74 }
75
76 void Swap(int *left, int *right) {
77   int temp = *left;
78   *left = *right;
79   *right = temp;
80 }
81
82 // Wrong WrongSwap: does not work
83 void WrongSwap(int left, int right) {
84   int tmp = left;
85   left = right;
86   right = tmp;
87 }
```