Computer Architecture

Programming in C

Functions

Agenda

- Define a function
- Passing parameters
  - Call-by-value
  - Call-by-reference
- Local variables and visible rules
- Multiple returns
- Exercises

How to define a function?

```c
#include <stdio.h>

int sum(int m, int n){      //function with two parameters
    int i, res=0;
    for(i=m; i<=n; i++){
        res = res + i;
    }
    return res;             //returns m+m+1+ ... +n
}

int main(){                 //main function
    int s;
    s = sum(10, 50);        //first call of the function
    printf("the sum of numbers from 10 to 50 is equal to %d", s);
    s = sum(1, 100);       //second call of the function
    printf("the sum of numbers from 1 to 100 is equal to %d", s);
    return 0;
}
```

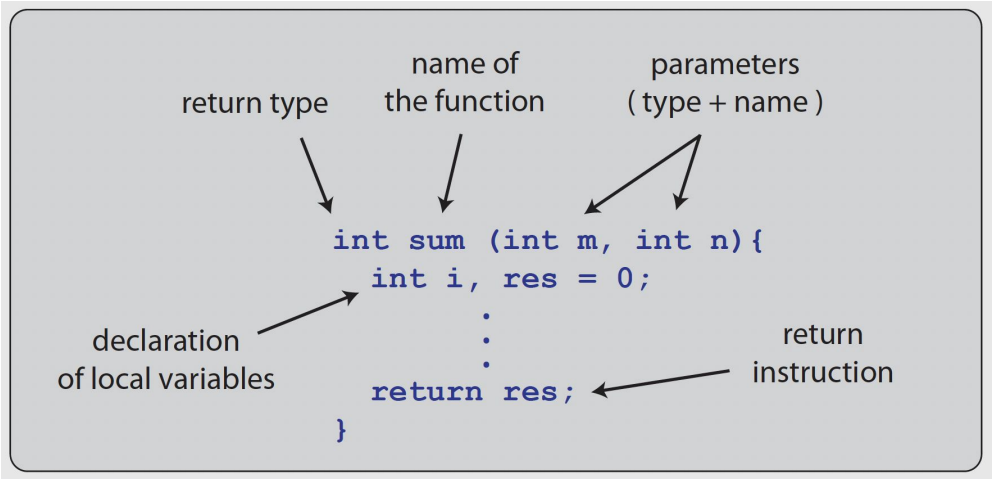
A C program is constructed as a sequence of functions.

- It must have a **principal function,** also called, entry point: `main()`;

In runtime, functions can be:

- **Caller:** the calling function, e.g., the `main()`
- **Callee:** the function being called, e.g., the `sum()`

Anatomy of a function



In the last program,

- The function `sum` is called from the function `main` with integer parameters `m=10` and `n=50` the first call, with integer parameters `m=1` and `n=100` the second call.
- The `return` instruction enables the callee to return its result to the caller (i.e., the instruction which called it.)

Call-by-value and call-by-reference

Two parameter-passing methods used in C program:

- The call-by-value discipline
  - A function receives a **copy** of the arguments' value, not the argument itself
  - So, the changes made to the parameters within the function do **not affect** the original argument outside the function
- The call-by-reference discipline:
  - A function receives a reference (e.g., array, or pointer) to the variable.
  - Modifications inside the function directly affect the original variable.

Call-by-value discipline

```
3   int swap (int x, int y) {
4       int temp;
5
6       temp = x;
7       x = y;
8       y = temp;
9   }
10
11  int main() {
12      int a=0, b=1;
13
14      printf("a=%d and b=%d", a, b);
15      swap(a, b);
16      printf("a=%d and b=%d", a, b);
17      return 0;
18  }
```

Question:

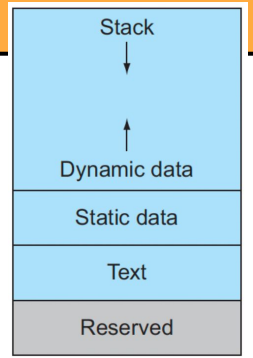
- Will the values of `a` and `b` be swapped, and why?

Description of the call-by-value procedure

```c
#include <stdio.h>

int sum(int m, int n){      //function with two parameters
    int i, res=0;
    for(i=m; i<=n; i++){
        res = res + i;
    }
    return res;             //returns m+m+1+ ... +n
}

int main(){                 //main function
    int s;
    s = sum(10, 50);        //first call of the function
    printf("the sum of numbers from 10 to 50 is equal to %d", s);
    s = sum(1, 100);       //second call of the function
    printf("the sum of numbers from 1 to 100 is equal to %d", s);
    return 0;
}
```

When the instruction `s=sum(10,50);` is reached, the program should be able to determine the value of the integer `s`.

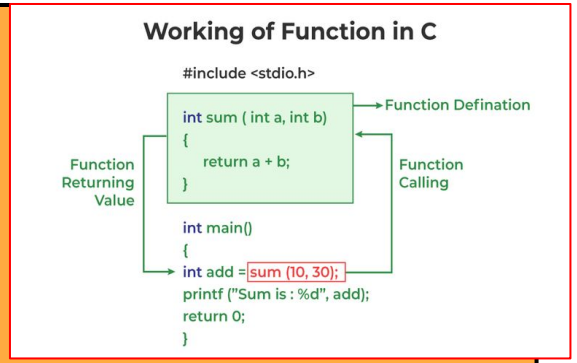
What follows is not an exact description of the evaluation of the expression `s = sum(10, 50);` but operations in a “stack frame” which represents in memory the function and its argument data.



Evaluation of `s = sum(10, 50);`

Phase 1. preparation of the function call

1. **Evaluation** of the expressions **arguments** of the call
   a. in that case: the two constant expressions of value 10 and 50
2. **Allocation** and initialization of the parameters
   a. allocation of two new variables `m` and `n`
   b. Initialization of `m` and `n` with the values calculated in 1.



Phase 2. execution of the function call

3. Execution of the body of the callee
   a. Allocation of local variables `i` and `res`
   b. The execution proceeds until it encounters the instruction `return`

Evaluation of `s = sum(10, 50);`

Phase 3. The execution encounters `return (expr);`

4. computation of the returned expression
   a. The value of the expression `expr` is computed
   b. This value is called the “returned value”
5. **free** the variable allocated for the execution
   a. Free the variables used as parameters `(m, n)`
   b. Free the variables locally declared `(i, res)`
6. **start again** the execution of **the caller** function
   a. The value determined for the expression `sum(10, 50)` is equal to the returned value of step 4

Local variables

The local variables of a function are:

- its parameters
- the variables declared in the body of the function

Lifespan of a local variable:

- it is not allocated yet before the function call
- it is not allocated any more after the function call

So, the variables `m, n, i, res` of the function `sum` exist in memory only during the call of the function.

```c
#include <stdio.h>

int sum(int m, int n){      //function with two parameters
    int i, res=0;
    for(i=m; i<=n; i++){
        res = res + i;
    }
    return res;             //returns m+m+1+ ... +n
}

int main(){                 //main function
    int s;
    s = sum(10, 50);        //first call of the function
    printf("the sum of numbers from 10 to 50 is equal to %d", s);
    s = sum(1, 100);       //second call of the function
    printf("the sum of numbers from 1 to 100 is equal to %d", s);
    return 0;
}
```

Visibility rules

- During the time they are allocated, the **local variables** of a function are **only visible in the body of this function.**
- So, from the function `main`, it is impossible to access the local variables of `sum` ⇒ in fact, the parameters and locally declared variables of `sum` are not even allocated as long as the function `sum` is called.
- Conversely, from the function `sum`, it is **impossible** to access the local variables of `main`. ⇒ the local variables of `main` remain allocated during the function call and execution of `sum` **but** they are **not** accessible.

Multiple returns

- A function may have multiple returns. This typically happens when with instructions like if-else.
- In that case, the **first encountered return** interrupts the execution.

```
4   int max(int m, int n){
5       int res;
6
7       if(m>n) {
8           res = m;
9       }
10      else {
11          res = n;
12      }
13      return res;
14  }
```

```
16  int max(int m, int n){
17      if (m>n){
18          return m;
19      }
20      else {
21          return n;
22      }
23  }
```

```
25  int max(int m, int n){
26      if (m>n) return m;
27      else return n;
28  }
```

Three versions of the `max` function. Actually, you can also use the ternary operator, which is the 4th way to do it.

Will the outputs be the same?

```
3   void print_succ(int n) {
4       n = n + 1;
5       printf("%d", n);
6   }
7
8   int main() {
9       int i=1;
10      print_succ(i);
11      printf("%d", i);
12      return 0;
13  }
```

```
9   void print_succ(int i) {
10      i = i + 1;
11      printf("%d", i);
12  }
13
14  int main() {
15      int i=1;
16      print_succ(i);
17      printf("%d", i);
18      return 0;
19  }
```

Note: `void` is a keyword that indicates a function doesn't have a return value.

Functions with arrays as parameters

```
3   void print_content(int a[], int size){
4       int i;
5       for(i=0; i<size; i++){
6           printf("%d", a[i]);
7       }
8   }
9
10  int main() {
11      int arr[] = {0, 42, 3, 10};
12
13      print_content(arr, 4);
14      return 0;
15  }
```

- `a` in `print_content` is passed by reference.
- `a` and `arr` represent the same memory address
- `print_content` will print the elements in `arr`.

Function with arrays as parameters

```
3   int length(char s[]){
4       int i=0;
5       while (s[i]!='\0') {
6           i++;
7       }
8       return i;
9   }
10
11  int main() {
12      char string[] = "abcdef";
13      printf("%s is length of  %d\n", string, length(string));
14  }
```

- It will print “abcdef is length of 6”.
- The `while` loop in `length` can also be realized by a `for` loop.

Modifying an array by calling a function

```
4   void print(int a[], int size){
5       int i;
6       for (i=0; i<size; i++) printf("%d ", a[i]);
7       printf("\n");
8   }
9
10  void erase(int a[], int size){
11      int i;
12      for (i=0; i<size; i++) a[i]=0;
13      printf("\n");
14  }
15
16  int main(){
17      int array[] = {3, 38, 23, 17};
18
19      print(array, 4);
20      erase(array, 4);
21      print(array, 4);
22      return 0;
23  }
```

- When the function `erase` is called, it sets the value of the array elements to be `0`, where the array is transmitted by the caller. (call-by-reference)

Exercise 9

```
21  int isitcorrect(char myword[], char myletters[]){
22      //complete the function
23  }
24
25  int main(){
26      char myword[100] = "bdacb";
27      char myletters[100] = "abbccd";
28
29      printf("%d\n", isitcorrect(myword, myletters));
30      return 0;
31  }
```

Write a function `isitcorrect`, which,

- returns 1 when the string `myword` contains exactly the letters appearing in the string `myletters`
- return 0 otherwise.

Analysis

- Every letter should appear the same number of times in the two strings: `myword` and `myletters`
- We typically have the choice between:
  - A simple but inefficient algorithm
  - A more efficient algorithm but less simple algorithm

Hint:

- recall the exercise 2 in arrays_and_strings (for the simple but inefficient way)
- each letter is an integer (for the less simple but efficient way)

Solution 1

Modify the code of exercise 2; define a function `count_occurance`

exercise2.h

```c
#include<stdio.h>

int cout_occurance(char str[], char c) {

    int i, counter = 0;

    for (i=0; str[i]!='\0'; i++){  //explore the elements of s
        if(str[i]==c){             //count the number of c's
            counter++;
        }
    }
    return counter;
}
```

```c
#include <stdio.h>
#include "exercise2.h"


int isitcorrect(char myword[], char myletters[]){

    int i, count_1, count_2;
    for(i=0; myletters[i]!='\0'; i++){
        count_1 = cout_occurance(myword, myletters[i]);
        count_2 = cout_occurance(myletters, myletters[i]);
        if(count_1!=count_2){
            return 0;
        }
    }
    return 1;
}


int main(){
    char myword[100] = "bdacb";
    char myletters[100] = "abbcd";

    printf("%d\n", isitcorrect(myword, myletters));
    return 0;
}
```

include the `count_occurance` into the program.

- The head file has the suffix .h
- If the file name is wrapped by <>, the compiler will search the file in system directories
- If the file name is wrapped by “”, the compiler will search the file in the current directory first; if not found, the system directories.

Solution 2 (more efficient than Solution 1)

```c
int isitcorrect(char myword[], char myletters[]){
    int i, inv_word[256]={}, inv_letter[256] = {};
    //construct the two inventories
    for(i=0; myword[i]!='\0'; i++){
        inv_word[myword[i]]++;
    }
    for(i=0; myletters[i]!='\0'; i++){
        inv_letter[myletters[i]]++;
    }
    //verfication
    for(i=0; i<256; i++){
        if(inv_word[i]!=inv_letter[i]){
            return 0;
        }
    }
    return 1;
}

int main(){
    char myword[100] = "bdacb";
    char myletters[100] = "abbccd";

    printf("%d\n", isitcorrect(myword, myletters));
    return 0;
}
```

- This version is more efficient than Solution 2. Can you explain why?

Exercise 10

```c
#include<stdio.h>

void bubble_sort(int a[], int size){

}

void print(int a[], int size){
    int i;
    for(i=0; i<size; i++){
        printf("%d ", a[i]);
    }
    printf("\n");
}

int main(){
    int arr[] = {9, 5, 8, 3, 1, 2, 6, 7, 4, 0};

    bubble_sort(arr, 10);
    print(arr, 10);
}
```

Write a function `bubble_sort` that sort an array of integers using the bubble sort algorithm.

We've known how to complete it in Python, now, please translate the Python code to C.

Exercise 11

The most naive strategy to earn profit in a stock market is to buy a share of a company at the time `tb` and sell it in a time `ts` in the future. The price difference between `ts` and `tb` is the profit. For example, one can buy 1 share of Apple Inc at the time t0 when the price is $100, and sell it at the time t3 when the price goes to $200. So, she/he will earn (200-100) = 100 dollars.

Please write a function that takes a list of the prices `[P0, P1, ..., P(n-1)]` of a stock during a duration `t= [t0, t1, ..., t(n-1)]` as the input. The function will compute the price differences `Pj-Pi`, where `j>i`, and print the maximum one.

```
13  int main(){
14      int maxv;
15      //daily price
16      int R[10] = {9, 5, 7, 3, 8, 18, 20, 19, 30, 21};
17
18      //complete the code
19
20
21      printf("max profit is %d\n", maxv);
22      return 0;
23  }
```

Solution 1

```
13  int main(){
14      int maxv;
15      //daily price
16      int R[10] = {9, 5, 7, 3, 8, 18, 20, 19, 30, 21};
17
18      maxv = -1000;
19      for(int i=0; i<10; i++){
20          for (int j=i+1; j<10; j++){
21              if(maxv<R[j]-R[i]){
22                  maxv = R[j]-R[i];
23              }
24          }
25      }
26      printf("max profit is %d\n", maxv);
27      return 0;
28  }
```

- For each price, compute the difference between it and the following prices.
- Then, return the largest one.
- Two for loops are needed.

Solution 2 (much faster than Solution 1)

```
43  int main(){
44      int i, maxv, minv;
45
46      int R[10] = {9, 5, 7, 3, 8, 18, 20};
47
48      maxv = -1000;
49      minv = R[0];
50      for(i=0; i<10; i++) {
51
52          maxv = max(maxv, R[i]-minv);
53          minv = min(minv, R[i]);
54      }
55      printf("max profit is %d\n", maxv);
56      return 0;
57  }
```

- Only one for loop is needed;

Hint:

- keep (and update) the minimum price found in each iteration
- So, in each iteration, you can compare the max_value you found so far with the current_price - minimum_price, and update it.

codeforces.com

- https://codeforces.com/ : it is a platform typically used when preparing for competitive programming contests.
  - It provides a huge number of programming questions; some are challenging;
  - It has an online judge system (OJ), testing your code and giving feedback. You can submit your answers in many different programming languages, including C.
  - It offers the following features: Short (2-hours) contests, called "Codeforces Rounds", held about once a week.
  - Free for registration!

Exercise 12

Three brothers: https://codeforces.com/contest/2010/problem/B

B. Three Brothers

time limit per test: 1 second
memory limit per test: 256 megabytes

Three brothers agreed to meet. Let's number the brothers as follows: the oldest brother is number 1, the middle brother is number 2, and the youngest brother is number 3.

When it was time for the meeting, one of the brothers was late. Given the numbers of the two brothers who arrived on time, you need to determine the number of the brother who was late.

Input

The first line of input contains two different integers a and b ($1 \le a, b \le 3, a \ne b$) — the numbers of the brothers who arrived on time. The numbers are given in arbitrary order.

Output

Output a single integer — the number of the brother who was late to the meeting.

Example

input	Copy
3 1	
output	Copy
2	

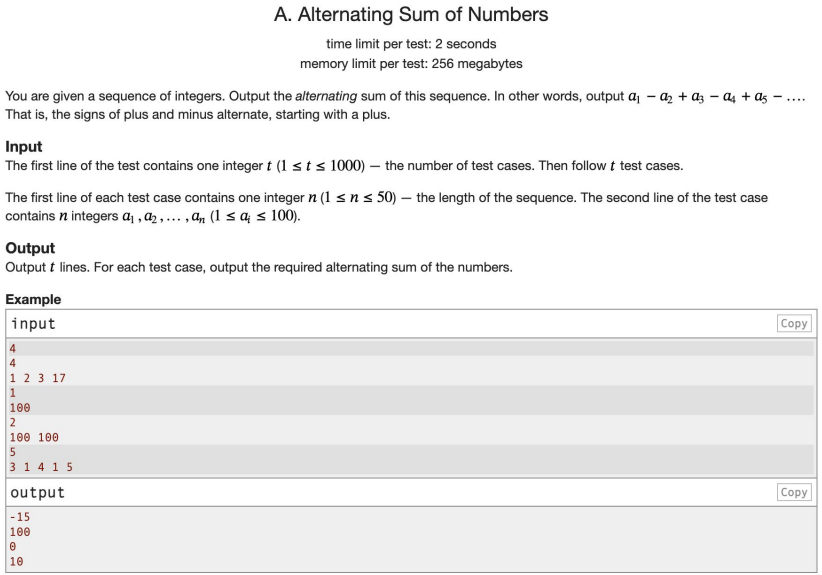
```c
#include<stdio.h>

int main(){
    int a, b, absent;
    scanf("%d %d", &a, &b);
    //complete the code

    printf("%d", absent);
    return 0;
}
```

It can be solved without using if-else!

Exercise 13

Alternating sum of numbers:

https://codeforces.com/problemset/problem/2010/A



A. Alternating Sum of Numbers

time limit per test: 2 seconds

memory limit per test: 256 megabytes

You are given a sequence of integers. Output the *alternating* sum of this sequence. In other words, output $a_1 - a_2 + a_3 - a_4 + a_5 - \ldots$. That is, the signs of plus and minus alternate, starting with a plus.

Input

The first line of the test contains one integer t ($1 \le t \le 1000$) — the number of test cases. Then follow t test cases.

The first line of each test case contains one integer n ($1 \le n \le 50$) — the length of the sequence. The second line of the test case contains n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le 100$).

Output

Output t lines. For each test case, output the required alternating sum of the numbers.

Example

input (Copy)

```
4
4
1 2 3 17
1
100
2
100 100
5
3 1 4 1 5
```

output (Copy)

```
-15
100
0
10
```

- It is a simple problem, but a good practice for `scanf` and `printf`.

Hint:

- Read the description of input and output carefully.
- You can use a while loop to get the input (from the OJ of codeforeces.com)
- Scanf and non-standard input format [Tutorial for beginners] - Codeforces