
C

One of the reasons why the C programming language is so popular is the flexibility of its use for memory management. Programmers have opportunities to control how, when, and where to allocate and deallocate memory. Memory is allocated statically, automatically, or dynamically in the C programming language.

Static variables and global variables are static-duration variables, which are allocated along with the executable code of the program and persist during the execution of the application. When a local variable is declared as static inside a function, it is initialized only once. Whatever values the function puts into its static local variables during one call will be present when the function is called again. The following is a typical interview question about static local variables: What is the output when the function `test` executes? (See Listing 2-1).

Listing 2-1. *C Code with Static Variables*

```
int hasStatic(int n) {
 static int x = 0;
 x += n;
 return x;
}

void test() {
 int sum = 0;
 for(int i = 1; i <= 4; ++i)
     sum += hasStatic(i);

 printf("Result of sum is %d.\n", sum);
}
```

---

[1] The TIOBE company publishes the monthly *TIOBE Programming Community Index*, which is an authoritative indicator of the popularity of programming languages. Please refer to the web site `www.tiobe.com/index.php/content/paperinfo/tpci/index.html` for the latest ranking.

The variable `x` in the function `hasStatic` is declared as `static`. When the function is called the first time with a parameter 1, the static variable `x` is initialized as 0, and then set to 1. When the function is called the second time with a parameter 2, it is not initialized again and its value 1 persists. It becomes 3 after execution. Similarly, its value becomes 6 and 10 after execution with parameters 3 and 4. Therefore, the result is the sum 1+2+6+10=19.

Automatic-duration variables, also called local variables, are allocated on the stack. They are allocated and deallocated automatically when the program execution flow enters and leaves their scopes. In the sample code in Listing 2-2, there is a local array `str` in the function `allocateMemory`. What is the problem with it?

Listing 2-2. *C Code to Allocate Memory*

```
char* allocateMemory() {
 char str[20] = "Hello world.";
 return str;
}

void test() {
 char* pString = allocateMemory();
 printf("pString is %s.\n", pString);
}
```

Because the array `str` is a local variable, it is deallocated automatically when the execution flow leaves the function `allocateMemory`. Therefore, the returned memory to `pString` in the function `test` is actually invalid, and its content is somewhat random.

One limitation of static-duration and automatic-duration variables is that their size of allocation is required to be compile-time constant. This limitation can be avoided by dynamic memory allocation in which memory is more explicitly, as well as more flexibly, managed. In the C programming language, the library function `malloc` is used to allocate a block of memory dynamically on the heap, where size is specified at runtime. This block of memory is accessible via a pointer that `malloc` returns. When the memory is no longer needed, the pointer should be passed to another library function, `free`, which deallocates the memory in order to avoid memory leaks.

The function `allocateMemory` in Listing 2-3 utilizes the function `malloc` to allocate memory dynamically. What is the problem in it?

Listing 2-3. *C Code to Allocate Memory*

```
void allocateMemory(char* pString, int length) {
```

```
    pString = (char*)malloc(length);
}

void test() {
    char* pString = NULL;
    allocateMemory(pString, 20);
    strcpy(pString, "Hello world.");
}
```

The parameter `pString` in the function `allocateMemory` is a pointer, and the string content it points to can be modified. However, the memory address it owns cannot be changed when the function `allocateMemory` returns. Since the input parameter `pString` of `allocateMemory` in the function `test` is `NULL`, it remains `NULL` after the execution of `allocateMemory`, and it causes the program to crash when the `NULL` address is accessed in `test`. In order to fix this problem, the first parameter of `allocateMemory` should be modified as `char** pString`.

Macros are commonly used in the C programming language and also commonly asked about in technical interviews. Since it is tricky and error-prone to use macros, unexpected results are gotten when they are misused. For example, what is the output when the function `test` in Listing 2-4 gets executed?

Listing 2-4. *C Code with Macros*

```
#define SQUARE(x) (x*x)

void test() {
    printf("Square of (3+4) is %d.\n", SQUARE(3+4));
}
```

The result of 3+4 is 7, and the square of 7 is 49. However, 49 is not the result of `SQUARE(3+4)`. Macros are substituted by the preprocessor, so `SQUARE(3+4)` becomes 3+4*3+4, which is actually 19. In order to make the result as expected, the macro should be modified to `#define SQUARE(x) ((x)*(x))`.

Even in the revised version of macro `SQUARE`, its result might look surprising if it is not analyzed carefully in some cases. For instance, what are the results of `x` and `y` after the code in Listing 2-5 is executed?

Listing 2-5. *C Code with Macros*

```
#define SQUARE(x) ((x)*(x))

void test() {
    int x = 5;
    int y = SQUARE(x++);
    printf("Result of x is %d, y is %d.\n", x, y);
}
```

It looks like that the variable `x` increases only once in the code above. However, when the macro is substituted, the variable `y` is assigned as `((x++)*(x++))`. The result of `y` is 25, and `x` becomes 7 because it increases twice.

Many coding interview questions can be solved in the C programming language. A question in the following subsection is an example, and more questions are discussed in later chapters.

Palindrome Numbers

Question 1 Please implement a function that checks whether a positive number is a palindrome or not. For example, 121 is a palindrome, but 123 is not.

Converting a Number into a String

It is easy to check whether a string is a palindrome or not: We can check whether the first character and the last one are identical, and then compare the second character and the second one from the end, and so on. If the converted string is a palindrome, the original should also be a palindrome.

This solution can be implemented with the code in Listing 2-6, which converts a number into a string with the library function `sprintf`.

Listing 2-6. *C Code to Verfiy Palindrome Numbers (Version 1)*

```
/* It returns 1 when number is palindrome, otherwise returns 0. */
#define NUMBER_LENGTH 20
int IsPalindrome_solution1(unsigned int number) {
    char string[NUMBER_LENGTH];
    sprintf(string, "%d", number);

    return IsPalindrome(string);
```

```
}


int IsPalindrome(const char* const string) {
    int palindrome = 1;
    if(string != NULL) {
        int length = strlen(string);
        int half = length >> 1;

        int i;
        for(i = 0; i < half; ++ i) {
            if(string[i] != string[length - 1 - i]) {
                palindrome = 0;
                break;
            }
        }
       }

    return palindrome;
}
```

Usually, this solution is not the one expected by interviewers. One reason is that while it is intuitive, interviewers expect something innovative, and another reason is that it requires auxiliary memory to store the converted string.

Composing a Reversed Number

As we know, it is easy to get digits from right to left via the / and % operators. For example, digits in the number 123 from right to left are 3, 2, and 1. A reversed number 321 is constructed with these three digits. Let's check whether the reversed number is identical to the original one. If it is, the original number is a palindrome.

The implementation is shown in Listing 2-7.

Listing 2-7. *C Code to Verfiy Palindrome Numbers (Version 2)*

```
   /* It returns 1 when number is palindrome, otherwise returns 0. */
int IsPalindrome_solution2(unsigned int number) {
    int reversed = 0;
    int copy = number;

    while(number != 0) {
        reversed = reversed * 10 + number % 10;
        number /= 10;
    }

    return (reversed == copy) ? 1 : 0;
}
```

Source Code:

001_PalindromeNumber.c

Test Cases:

- Palindrome numbers with odd length
- Palindrome numbers with even length
- Non-palindrome numbers