Handbook for Programming Contest

Release 0.1 alpha

Xinhong

May 09, 2015

- PDF
- GitHub

Contents:

CHAPTER

ONE

INTRO

The handbook is for quick look on general programming contest like ACM.

The handbook is built with Sphinx. Lanuage demostrated here is C/C++.

1.1 Motivation

I have been using the code template made by Shanghai JiaoTong University for a long time. And I found some problems.

1.1.1 The codes are "bad codes"

When I say the code is bad doesn't mean the code doesn't work or performance is poor. As a code template, I think the goal is to achive

- it works
- avoid mistyping (due to the tense in the contest environment)

While the second is a huge problem. I know sometimes in order to type fast, the variables' name may need to be as short as possible. But readability is really an issue considering the insanely short variables in the template made by Shanghai JiaoTong University.

A nice code style gives you a nice mood during the contest. So in this handbook, the variables' naming will be *intention-revealing*, readability is first concern, I need you to understand the code first, then you will not make typing mistake, finally you will type fast.

1.2 Contribution

This handbook is currently at very beginning, where content is far away from **enough**. If you are interested in making it better, please **DO**.

CHAPTER

TWO

TOPICS

Contents:

2.1 I/O

I/O exists in every program, in most cases it is easy, some cases are annoying.

I recommend you to use C instead C++ especially when the input or output is small. The advantage of performance using C I/O is obvious, while C++ may handle some situation quite friendly.

2.1.1 In

scanf

Reading from stdio, we often use `scanf` (scanf doc).

For example:

```
scanf("%d", &a_int); // read int
scanf("%ld", &a_long); // read long
scanf("%lld", &a_longlong); // read long long
scanf("%llu", &a_unsignedlonglong); // read unsigned long long
scanf("%f", &a_float); // read float
scanf("%lf", &a_double); // read double
```

and of course, character array:

```
char string[LENGTH];
scanf("%s", &string);
```

gets

While if you want to read a whole line string containing spaces, `scanf` may not be the best choice, and `gets` (gets doc) is obviously better. (although some IDE will warn you it is not safe, go to hell!)

Here is an example:

```
char string[LENGTH];
gets(string);
```

WATCH OUT!! In this case, special attentions: if the stdin is:

```
1
I am lucky!
<EOF>
```

and the code is:

```
int i;
scanf("%d", &i);
char string_unlucky[100], string_lucky[100];
gets(string_unlucky);
gets(string_lucky);
printf("string_unlucky is: %s\n", string_unlucky);
printf("  string_lucky is: %s\n", string_lucky);
```

You will get the output:

```
string_unlucky is:
  string_lucky is: I am lucky!
```

try to figure out why.

2.1.2 Out

printf

Basically, we use `printf` (printf doc) for put content to stdout.

Just do it similarly:

```
printf("%d", a_int);
```

2.2 Evaluation

2.2.1 Expression Forms

Normally there are three forms of expression:

- Infix (daily life)
- Prefix
- Postfix

Infix

Infix expression has this form $(a+b)\times c$, we are familiar with this form and we use it every day, while it is not friendly when we want to evaluate it :-(.

Prefix

Prefix expression has this form $\times + abc$, which operator goes first, then operands.

Postfix

Postfix expression has this form $ab + c\times$, which operator goes after operands.

2.2.2 Expression Evaluation

Among the three forms, Infix is the hardest to evaluate, and the other two are relatively easy.

Postfix

1. Create a stack to store operands
2. Scan the expression from left to right, token by token
   - if the token is an operand, push it to stack
   - if the token is an operator, pop the operands and apply operetor to the operands in the stack and push the result back to stack

Here is an example, the operands are single-digit integers, and operators are `+ - * /`.

```
#include <stdio.h>
#include <stack>
#include <string.h>

bool is_operator(char character) {
    if (character == '+' || character == '-'
          || character == '*' || character == '/')
        return true;
    else
        return false;
}

int perform_operation(char the_operator,
      int operand_left, int operand_right) {
    if (the_operator == '+') return operand_left + operand_right;
    else if (the_operator == '-') return operand_left - operand_right;
    else if (the_operator == '*') return operand_left * operand_right;
    else return operand_left / operand_right;
}

int evaluate_postfix(char *expression) {
    int length = strlen(expression);

    std::stack<int> operands;
    for (int i = 0; i < length; ++i) {
        if (is_operator(expression[i])) {
            int operand_right = operands.top(); // right comes first
            operands.pop();
            int operand_left = operands.top();
            operands.pop();
            operands.push(perform_operation(expression[i],
                          operand_left, operand_right));
        } else {
            operands.push(expression[i] - '0');
        }
    }
    return operands.top();
```

```
}

int main() {
    char expression[] = "12+3*4+";
        printf("%s = %d\n", expression, evaluate_postfix(expression));
}
```

and the output is:

```
12+3*4+ = 13
```

Prefix

Similar to postfix.

Infix

Translate into postfix then evaluate it. (smart)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search