Encurtamento de Código

Prof. Edson Alves

2018

Faculdade UnB Gama

Sumário

Encurtamento de Código

Encurtamento de Código

- Em programação competitiva, a velocidade é critério de desempate
- Uma maneira de se melhorar o desempenho no quesito velocidade é digitar o mais rápido possível
- Outra maneira é diminuir a quantidade de caracteres que devem ser digitados no código
- Esta redução pode ser feita através do uso dos recursos da própria linguagem ou através do uso de macros
- O uso inteligente de padrões e lógica também pode levar a reduções nos códigos
- Como os códigos de competição não visam reuso, não faz sentido usar nomes longos e descritivos para variáveis ou evitar o uso de variáveis globais

Dica #1: Renomear tipos

Alguns tipos de dados tem nomes longos, que podem ser encurtados através do uso da diretiva **using** do C++:

```cpp
#include <iostream>
#include <vector>

using namespace std;
using ll = long long;
using ii = pair<ll, ll>;
using vi = vector<ll>;

int main()
{
    ll x = 2, y = 3;
    ii z(x, y);
    vi ns { x, y, 10, -1 };

    return 0;
}
```

Dica #2: C++ Moderno - auto

- Alguns recursos do C++ moderno também levam a uma redução do tamanho dos códigos-fontes
- A partir do C++11, a palavra reservada **auto** pode ser usada para deduzir o tipo de uma variável a partir de sua inicialização
- Em alguns casos, a redução de código é notável:

```
1 map<string, int> m;
2 ...
3 map<string, int>::iterator it = m.begin();
4 auto jt = m.begin();
```

- A inicialização pode ser feita com constantes, retornos de funções ou mesmo outras variáveis

```
1 auto x = 1.0;                    // typeid(x).name() == double
2 auto y = x;                      // typeid(y).name() == double
3 auto z = toupper('a');           // typeid(z).name() == int
4 auto w = strstr("ABC", "BC");    // typeid(w).name() == char *
```

Dica #3: C++ Moderno - range for

- O padrão C++11 também trouxe o recurso do *range for*
- Antes deste padrão, a iteração em um conteiner era feito por meio de iteradores:

```
1 vector<int> vs;
2
3 for (vector<int>::iterator it = vs.begin(); it != vs.end(); ++it) {
4     // O valor do elemento e acessado com a sintaxe *it
5 }
```

- O *range for* simplifica a sintaxe de iteração sobre contêiners, elimando o uso explícito de iteradores:

```
1 vector<int> vs;
2
3 for (auto& v : vs) {
4     // O valor do elemento e acessado pela propria variavel v
5 }
```

Dica #4: C++ Moderno - `lambdas`

- O C++11 também permite o uso de funções *lambda*
- *lambdas* são funções anônimas que podem ser declaradas e implementadas localmente
- O uso de *lambdas* simplifica o código, permitindo a customização de funções que permitem a customização de seu comportamento através do uso de ponteiros para funções

```
1 vector<int> ns { 2, -1, 5, 8, 3, 10 };
2
3 // Primeiro os pares, depois os impares
4 sort(ns.begin(), ns.end(), [](int x, int y) {
5     int a = x % 2;
6     int b = y % 2;
7     return a == b ? x < y : (a == 0);
8 });
```

Dica #5: STL

- A STL (*Standard Template Library*) do C++ é a biblioteca padrão de *templates*
- Ela oferece a implementação de estruturas de dados e funções parametrizadas
- Estas implementações são flexíveis tanto no que diz respeito ao tipo de dado que elas agem quanto ao seu conjunto de parâmetros
- O uso das estruturas e funções da STL simplificam o código
- Além da redução do tamanho, o uso da STL traz confiabilidade ao código, uma vez que as implementações do compiladores são robustas e bem testadas
- Produzir implementações equivalentes durante uma maratona é inviável

Exemplo de uso da STL

```cpp
#include <iostream>
#include <algorithm>

int main() {
    std::string text { "Exemplo de uso da STL" }, vowels { "aeiou" }, ans;

    std::transform(text.begin(), text.end(), text.begin(),
        [](char c) { return tolower(c); });

    std::copy_if(text.begin(), text.end(), back_inserter(ans),
        [&](char c) { return vowels.find(c) != std::string::npos; });

    std::sort(ans.begin(), ans.end());

    auto n = std::unique(ans.begin(), ans.end()) - ans.begin();
    ans.resize(n);

    std::cout << n << " vogais unicas: " << ans << std::endl;

    return 0;
}
```

Dica #6: Impressão de vetores

- Um padrão comum de respostas de problemas de competição é imprimir um vetor de números, separados por espaços entre eles
- O código abaixo gera um *Presentation Error*, pois imprime um espaço em branco após o último número:

```
1 for (size_t i = 0; i < ns.size(); ++i)
2     cout << ns[i] << " ";
3 cout << endl;
```

- Além disso, é preciso de uma linha extra, após o **for**, para a quebra de linha
- O código abaixo resolve ambos problemas:

```
1 for (size_t i = 0; i < ns.size(); ++i)
2     cout << ns[i] << (i + 1 == ns.size() ? '\n' : ' ');
```

Dica #7: Eliminação de `if`

- A eliminação de um **`if`**, além de reduzir o tamanho do código, traz como benefício adicional um ganho de performance de execução
- O operator % permite navegar circularmente pelos números $0, 1, \ldots, N-1$ sequencialmente sem a necessidade de um **`if`** para determinar o momento de se reiniciar o contador

```
1 i = (i + 1) % N;          // Sentido crescente
2 j = (j + N - 1) % N;      // Sentido decrescente
```

- É possível alternar entre dois inteiros A e B sem necessidade de um **`if`**: se $C = A + B$, então o valor de i alterna entre A e B se for atualizado com o valor $C - i$:

```
1 int i = A;
2 ...
3 i = C - i;
```

Dica #8: Definição de constantes importantes

- Embora constantes também possam ser definidas como macros, usar os recursos da própria linguagem permitem um maior nível de verificação e validação da parte do compilador
- A palavra reservada **const** pode ser usada para este fim
- Em geral, as constantes devem ser variáveis globais e receber nome com todos os caracteres maiúsculos
- Todas as constantes devem receber um valor inicial, uma vez que não é possível alterar seus valores em qualquer outro ponto do código

```
1 const int MAX { 1000010 };
2 const ll oo { (1 << 62) };
3 const ll MOD { 1000000007 };
4 const double PI = acos(-1.0);
```

Macros

Dica #9: Macro para laços for

- O uso de macros pode encurtar o tamanho de padrões comuns em códigos C/C++
- Um destes padrões é o laço for que faz a travessia de um vetor de N elementos
- Além da redução do tamanho, a macro tem a vantagem de evitar erros no incremento

```cpp
#include <iostream>

#define REP(i, a, b) for (int (i) = (a); (i) <= (b); (i)++)

int main()
{
    int ns[] { 2, 3, 5, 8, 13, 21 };

    REP(i, 1, 4)
        std::cout << ns[i] << std::endl;      // 3 5 8 13

    return 0;
}
```

Dica #10: Macros para debug

- Macros podem ser utilizadas para imprimir informações de *debug* do código
- Escrevendo as informações de saída em `cerr` a solução poderá ser aceita mesmo com os *logs* ativados (desde que a escrita destes *logs* não gere um TLE)
- Os juízes online definem a macro `ONLINE_JUDGE`, que pode ser usada para desabilitar os *logs*
- As macros de *log* devem ser simples de usar e criadas para os tipos mais comuns (variáveis de tipo primitivo, vetores, *arrays*, etc)

Exemplo de uso de macros de log

```
#define LOG(var, sep) (cerr << #var << " = " << (var) << (sep))

#define LOGM(msg) (cerr << (msg))

#define LOGV(vec, sep) {                                        \\
    cerr << #vec << " =";                                       \\
    for (const auto& v : (vec)) cerr << " " << v;               \\
    cerr << (sep); }

#define LOGA(arr, N, sep) {                                     \\
        cerr << #arr << " =";                                   \\
        for (int i = 0; i < N; ++i) cerr << " " << (arr)[i];    \\
        cerr << (sep); }

#define LOGA2(arr, N, M, sep) {                                 \\
        cerr << #arr << " =\n";                                 \\
        for (int i = 0; i < N; ++i) {                           \\
            for (int j = 0; j < M; ++j)                         \\
                cerr << (arr)[i][j] << " ";                     \\
            cerr << endl;                                       \\
        } cerr << (sep); }
```

Dica #11: Inclusão da biblioteca padrão do C++

- A biblioteca padrão do C++ está dividida em várias sub-bibliotecas, cada uma associada a um arquivo *header* diferente
- A inclusão apenas dos *headers* necessários acelera o processo de compilação
- Contudo, a falta de algum *header* necessário gera um erro de compilação, e a identificação e correção do erro toma tempo
- Uma alternativa utilizada por muitos competidores é incluir o *header* `bits/stdc++.h`
- Este arquivo faz parte do GCC, e incluir todos os arquivos associados a biblioteca padrão do C++

Referências

1. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
2. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.
3. **SKIENA**, Steven S.; **REVILLA**, Miguel A. *Programming Challenges*, 2003.
4. CppReference[1].

[1] https://en.cppreference.com/w/