APROFUNDANDO C++(E OUTROS TÓPICOS)

TÓPICOS DE ABORDAGEM

- Aprofundando C++
  - Ponteiros e referências
  - Funções (nossas e prontas)
  - Structs (tipos compostos)
- Recursão
- Problemas iterativos

PONTEIROS E REFERÊNCIAS

PONTEIROS

Armazenam o endereço de memória de outras variáveis

Modificar o valor do ponteiro pelo operador “*” modifica o valor da variável original

```cpp
// Ponteiros armazemam endereços de memória
int x = 10;
int *ponteiro = &x;
int y = 20;

*ponteiro = 15;
ponteiro = &y; // Agora o ponteiro aponta para y
*ponteiro = 35;

// Valor da variável que o ponteiro aponta
cout << *ponteiro << endl;

// imprime 15 e 35
cout << x << ' ' << y << endl;
```

ARRAYS SÃO PONTEIROS

Quando criamos um array ‘arr’ (pode ser qualquer outro nome), o próprio arr já é um ponteiro.

arr é um ponteiro para a posição 0, arr + 1 é um ponteiro para a posicão 1 e assim por diante...

arr[x] é apenas uma outra forma de escrever *(arr + x)

```
int n = 10;
int arr[n]; // arr é um ponteiro!
*(arr) = 1; // Mesmo que arr[0] = 1

// arr[i] é o mesmo que *(arr + i)
cout << *(arr + 4) << endl;
sort(arr, arr + n);
```

OPERAÇÕES COM PONTEIROS

Podemos fazer algumas operações com ponteiros:

Ponteiro + número → Retorna o ponteiro apontando para `número` posições de memória a frente.

Ponteiro - Ponteiro → Retorna a distância entre os ponteiros na memória

Isso é especialmente útil quando se é usado com funções prontas do c++ que trabalham com isso

```
// Imprime arr[5] e 3
cout << *(arr + 5) << endl;
cout << (arr + 5) - (arr + 2) << endl;
```

REFERÊNCIAS

Referências são meros apelidos de variáveis.

Embora não seja uma boa prática, pode ser bem útil para evitar escrever nomes muito grandes repetidamente.

Toda modificação feita na referência afeta a variável original

```
// Referências são meros apelidos para variáveis
int variavel_com_nome_muito_grande = 50;
int &z = variavel_com_nome_muito_grande;

z = min(z, 10);
```

ITERATORS

São os ponteiros das estruturas de dados. Funcionam de forma parecida com os ponteiros.

```cpp
// Iterators adicionam uma camada de segurança aos ponteiros
// Para o vector, podemos fazer contas com os iterators em O(1);

vector<int> v(n);
for(int &x : v) x = 5;

// Conta quantos números 5 tem no vector
int cnt = upper_bound(v.begin(), v.end(), 5) -
   lower_bound(v.begin(), v.end(), 5);

cout << *(v.begin() + 5); // acesso ao sexto elemento
```

FUNÇÕES

Podemos criar nossas próprias funções além de usar as prontas.

Nosso código fica mais claro, além de reutilizável (isso ajuda muito!!)

```
// Funções servem para reutilizar código e/ou deixam
// o código mais claro.
int minhaFuncao(int argumento1, int argumento2) {
    return argumento1*argumento1 + argumento2;
}

int f(int x) {
    return x*x + 2*x + 3;
}

void procedimento(int arg1, string arg2) {
    // Qualquer código pode ficar nas funções
    int var = 10;
    for(int i=0; i<10; i++) {
        cout << i << ' ';
    }
    cout << endl;
}
```

PASSAGEM POR VALOR E REFERÊNCIA

Quando passamos um parâmetro para a função, por padrão, uma cópia dele é criada. Portanto, mudanças da cópia dentro da função não alteram o valor original.

Quando passamos o valor por referência, qualquer mudança do argumento afeta seu falor original.

```cpp
// vector será passado por cópia!!
// A chamada dessa função agora é O(n)
void funcaoVector(vector<int> v) {
    cout << v.size();
    v[1] = 10;
}

// vector agora é passado por referência
// CUIDADO: Qualquer mudança em v agora afeta o vector original
void funcaoVectorReferencia(vector<int> &v) {
    cout << v.size();
    v[1] = 10; // Afeta o vector original
}
```

FUNÇÕES ÚTEIS

SORT

Ordena os elementos de um intervalo. O intervalo é definido por ponteiros/ iterators. Por padrão, os elementos são ordenados na ordem crescente (o menor vem primeiro)

Complexidade: O(n log n)

```
// sort(inicio, fim)
// ordena os elementos no intervalo.
// por padrão, faz isso na ordem crescente (o menor vem primeiro)

vector<int> u(5) = {4, 3, 10, 5, 1};
sort(v.begin(), v.end());
// u agora é {4, 3, 10, 5, 1}
```

LOWER E UPPER BOUND

lower bound → Retorna o primeiro elemento do intervalo maior **ou igual** a x

upper bound → Retorna o primeiro elemento do intervalo estritamente maior que x

ambas as funções retornam iterators.

Complexidade: O(log n)

```cpp
// lower_bound(inicio, fim, x)
// Retorna o primeiro elemento entre o inicio e o fim
// maior ou igual a x.
// Retorna um interator para este elemento
auto lb = lower_bound(inteiros.begin(), inteiros.end(), 10);

// upper_bound(inicio, fim, x)
// Retorna um iterator para o primeiro elemento maior que x.
auto ub = upper_bound(inteiros.begin(), inteiros.end(), 10);

// 10 13
cout << *lb << ' ' << *ub << endl;
```

NEXT PERMUTATION

Usamos para gerar todas as permutações de uma lista.

O next permutation modifica o vetor para gerar o proximo vetor na ordem lexicográfica.

Ele retorna true quando ainda tem permutações a gerar e false caso contrário

```cpp
// next_permutation gera a próxima permutação de uma lista
// na ordem lexicográfica
bool acabou = next_permutation(inteiros.begin(), inteiros.end());
// Retorna true se a próxima permutação é de fato a próxima na ordem
// lexicográfica
// (usamos isso para ver se já passamos por todas as permutações)


// Iterando por todas as permutações de um vector
vector<int> p = {4, 3, 1, 10, 3};

// Começamos da menor permutação (que é quando ela está ordenada)
sort(p.begin(), p.end());

do {
    for(int x : p) {
        cout << x << ' ';
    }
    cout << endl;
} while(next_permutation(p.begin(), p.end()));
```

STRUCTS

CRIANDO STRUCTS

Structs nos permitem criar nossos próprios tipos de variável, compondo os tipos que já existem.

Todas elas possuem (mesmo que você não especifique) construtores, que são funções que são executadas toda vez que uma variável daquele tipo é criada.

```
struct Pessoa {
    string nome; int idade;

    Pessoa() {}
    Pessoa(string _nome, int _idade) {
        nome = _nome;
        idade = _idade;
    }
}
```

USANDO STRUCTS

Podemos usar struct como usamos outras variáveis (há resalvas)

Para acessar as variáveis e funções dentro de uma struct usamos o ``

```
struct MeuTipo {
    int atributo1;
    string atributo2;

    MeuTipo() {} // Contrutor vazio, não é obrigado a chamá-lo, mas é legal declarar

    // Construtor: Função que será executada quando o objeto for criado
    MeuTipo(int argumento) {
        atributo1 = argumento * 10;
        atributo2 = string(argumento, '0');
    }

    int metodo(int x) {
        return 0;
    }

    void imprima() {
        cout << atributo1 << ' ' << atributo2 << endl;
    }
};

int main() {
    // Aqui ele chama o construtor vazio. Se ele não existe, já era...
    MeuTipo carinhaDoMeuTipo;

    // Chamou o contrutor passando 10 como argumento
    MeuTipo outroCarinha(10);

    cout << carinhaDoMeuTipo.atributo1 << endl;
    cout << carinhaDoMeuTipo.atributo2 << endl;

    carinhaDoMeuTipo.atributo1 = 10;
    carinhaDoMeuTipo.atributo2 = "123";

    carinhaDoMeuTipo.imprima();
}
```

ORDENANDO STRUCTS

Podemos especificar como nossas estruturas serão ordenadas pelo sort.

Sua especificação deve seguir algumas regras, caso contrário, o sort pode dar erro.

Leia: https://www.boost.org/sgi/stl/StrictWeakOrdering.html

```
struct MinhaStruct {
    int attr1;
    int attr2;
    int attr3;

    // Podemos implementar nossos operadores para nossos tipos
    bool operator<(const MinhaStruct &rhs) const {
        if(attr2 == rhs.attr2)
            return attr3 < rhs.attr3;

        return attr1 < rhs.attr1;
    }

    // Agora podemos fazer a < b para dois caras do tipo MinhaStruct
};
```

INTRODUÇÃO À RECURSÃO

CONCEITO

Recursão ocorre quando uma função é definida a partir de si mesma.

São exemplos de funções que podem ser definidas recursivamente: fatorial, sequência de fibonacci, potência inteira, etc

Para uma função recursiva ser bem definida, ela precisa ter um caso base (onde ela não chama a si própria)

$$n = n \cdot (n-1)!$$

$$0! = 1$$

$$fib(n) = fib(n-1) + fib(n-2)$$

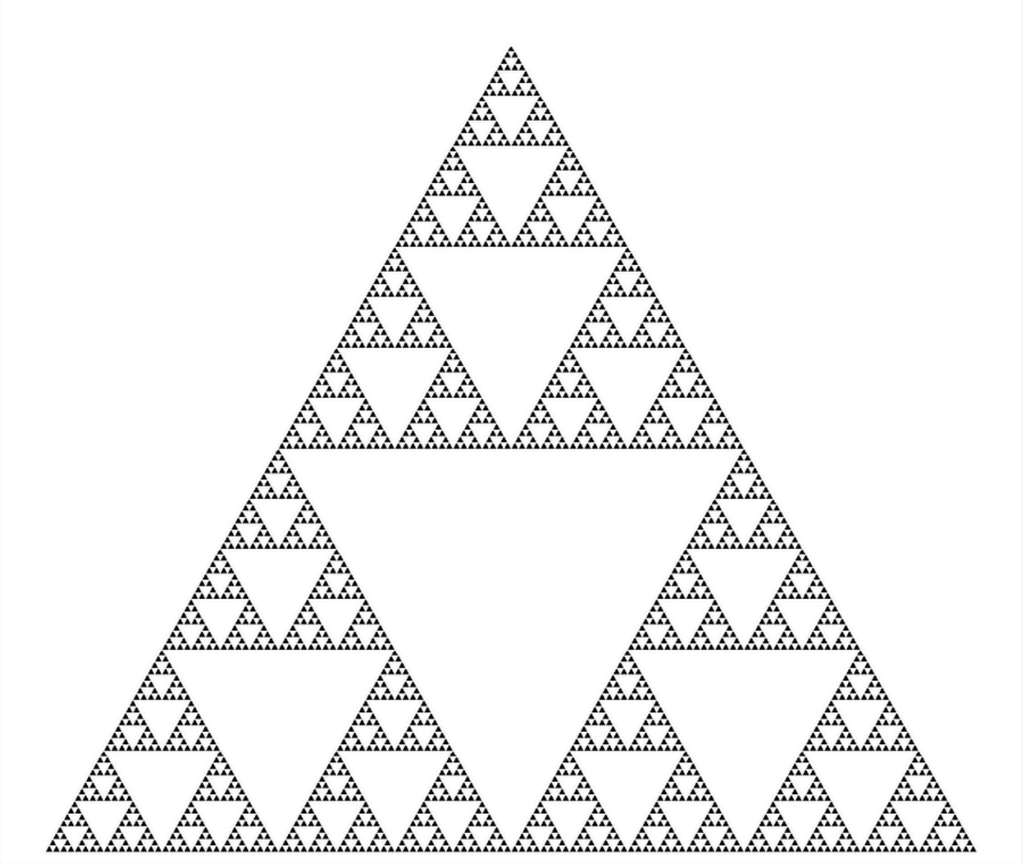
$$fib(1) = 1$$

$$fib(2) = 2$$

MAS SERVE PARA QUÊ?

As vezes é mais fácil pensar em um problema dividindo ele em partes menores de si mesmo.

Cada uma das partes menores pode ser dividida em mais partes menores e assim por diante, até chegarmos nas bases, que devem ser fáceis de calcular.



EXEMPLO (MOEDA)

Você tem um conjunto de moedas S e um número n. Você consegue somar n reais com suas moedas? (pode repetir moedas)

Exemplo: S = {3, 4, 5, 13, 20} n =17
Resposta: SIM → 17 = 13 + 4

Exemplo: S = {3, 10, 14, 17}; n = 21
Resposta: NÃO

Exemplo: S = {3, 4}; n = 6
Resposta: SIM → 6 = 3 + 3

DA PRA FAZER ITERATIVO?

Até dá, mas deixa eu te mostrar outra maneira de pensar...

Suponha que você tenha uma moeda que vale 5

Então se você consegue somar n-5, você consegue somar n.

Em geral, se você tem uma moeda de valor x você consegue somar n se você consegue somar n - x. Você está definindo seu problema a partir de um problema menor, isso é recursão

RESPOSTA

Para cada uma das suas moedas C, você testa se consegue somar n-C.
se der certo para pelo menos uma de suas moedas, então você consegue somar n

Mas ainda falta as bases! Senão vai ter loop infinito.

você trivialmente consegue somar 0 usando nenhuma moeda
você não consegue somar negativo (quem ia ter uma moeda de valo negativo?)

CÓDIGO

A complexidade é bem ruim, mas funciona (não se preocupa com isso agora, a gente vai estudar como melhorar isso depois)

```cpp
vector<int> moedas = {3, 6, 8, 10, 20};

bool consigoSomarX(int x) {
    if(x == 0) return true;
    if(x < 0) return false;

    for(int c : moedas) {
        if(consigoSomarX(x - c)) {
            return true;
        }
    }
    return false;
}
```

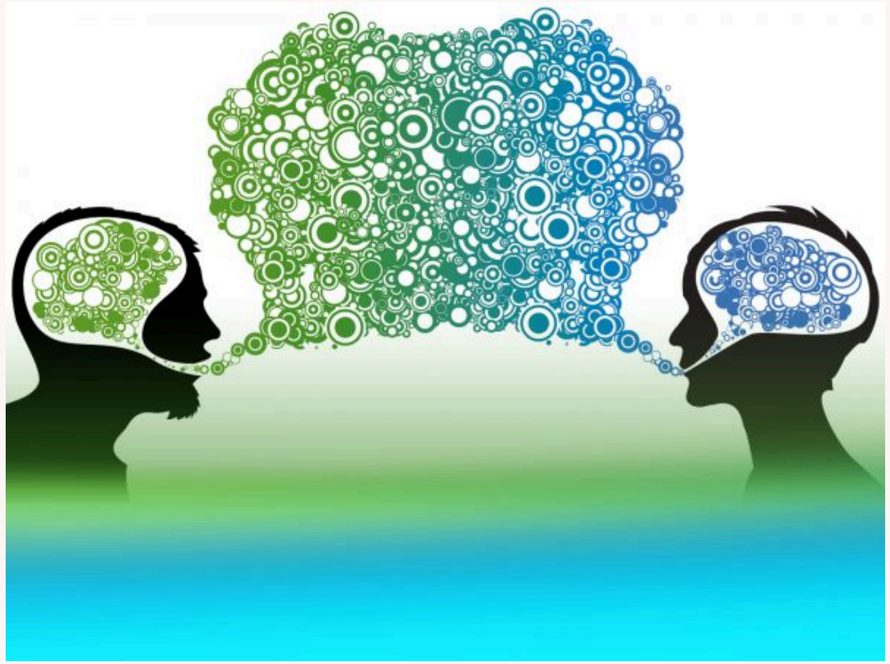
PROBLEMAS INTERATIVOS

QUESTÕES CONVENCIONAIS

Os problemas que fizemos até agora seguem este fluxo:

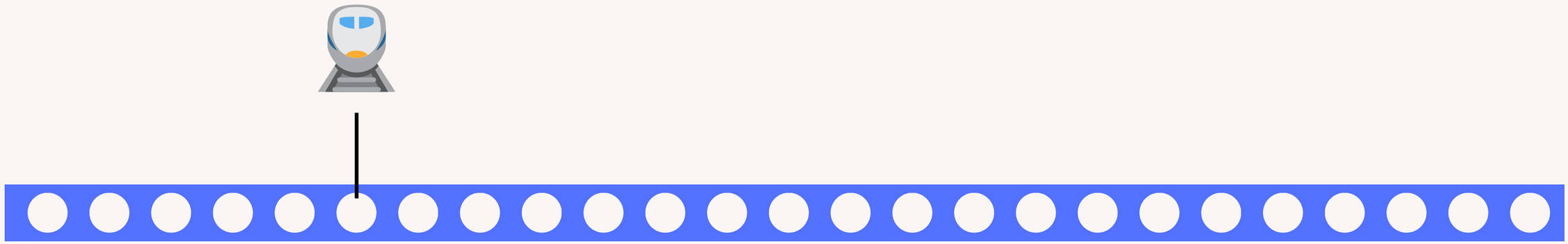
Seu programa recebe uma entrada e deve imprimir uma saída, apenas isso.

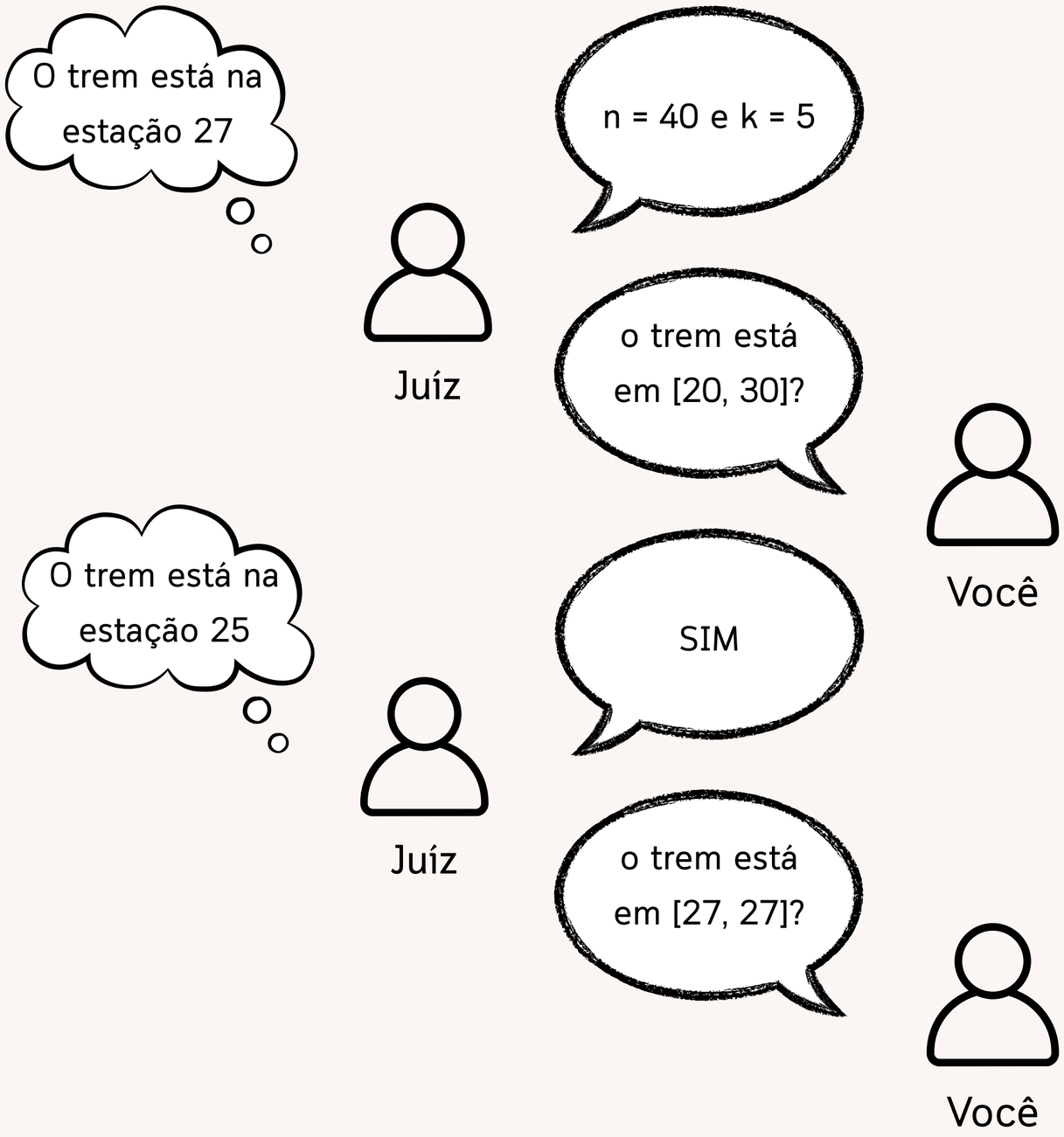
Em um problema interativo, seu programa deve “conversar” com o sistema (juíz) para alcançar um determinado objetivo (diferente em cada problema)



EXEMPLO: ACHE O TREM

Há n (n <= 10^18) estações de trem em uma linha reta e um número k (k<=10).
Há um trem em uma das n estações, mas você não sabe qual.
Você pode perguntar ao sistema: O trem está entre `a` e `b`? [você escolhe o a e o b]
A cada pergunta, o trem anda até k posições (para frente ou para trás)
Seu objetivo é descobrir onde o trem está.
Você pode fazer até 4500 perguntas, se estourar, é WA.




O trem está na estação 27
Juíz
n = 40 e k = 5
o trem está em [20, 30]?
Você
O trem está na estação 25
Juíz
SIM
o trem está em [27, 27]?
Você


O trem está na estação 20
Juíz
NÃO
o trem está em [20, 20]?
Você
Juíz
SIM; ACCEPTED
Você

MAS E O CÓDIGO?

Você pode ler e imprimir a saída normalmente, mas tem um detalhe:
Sempre dê ‘flush’ quando for imprimir algo, para garantir que o juíz vai receber sua pergunta/resposta.

Você pode fazer isso com cout << endl ou cout << flush para não quebrar a linha.

Mais detalhes: https://codeforces.com/blog/entry/45307

```
int main() {

    int n, k;
    cin >> n >> k;

    while(true) {
        cout << 5 << ' ' << 5 << endl;
        string resposta;
        cin >> resposta;

        if(resposta == "Bad") {
            // Você fez uma pergunta inválida ou
            // estourou a quantidade de perguntas
            return 0;
        } else if (resposta == "Yes") {
            // resposta é SIM
            return 0;
        } else if (resposta == "No") {
            // resposta é NÃO
        }

    }
}
```

DÚVIDAS?