Repetição, arranjos e conjuntos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Vimos anteriormente que devemos definir uma estrutura para representar uma informação quando ela consiste de dois ou mais itens que juntos descrevem uma entidade.

- No problema da conversão de segundos para horas, minutos e segundos, definimos a estrutura `Tempo`.
- No problema do ambiente gráfico, definimos as estruturas `Janela` e `Clique`.
- No problema da loteria, definimos a estrutura `SeisNumeros`.

O `Tempo` era composto de três “itens”, que foram representados pelos campos horas, minutos e segundos.

Já para `SeisNumeros` cada item não tinha uma interpretação particular, então não usamos nomes significativos, tivemos que “inventar” os nomes de `a`, ..., `f`.

Como faríamos se ao invés de 6 itens tivéssemos 20? E 1.000? E 1.000.000? Ou ainda, uma quantidade indefinida? E como escrever o código para processar esse tipo de dado?

Vamos ver como fazer essas coisas!

Quando precisamos representar uma coleção com um número fixo de valores da mesma natureza (todos os itens são notas, nomes, pontos, janelas, etc), utilizamos arranjos de tamanho fixo.

Os arranjos em C++ são definidos na biblioteca `array`. Quando declaramos uma variável do tipo arranjo, precisamos especificar o tipo e a quantidade de elementos. A inicialização é feita de forma semelhante a valores do tipo estrutura.

```
#include <array>
using namespace std;

examples
{
    array<int, 4> valores = {10, 4, 9, -1};
    array<string, 3> nomes = {"joao", "jose", "maria"};
```

Para ler ou modificar um valor específico do arranjo, especificamos o seu índice entre colchetes

```
    check_expect(valores[0], 10);
    check_expect(nomes[2], "maria");

    check_expect(valores[1], 4);
    valores[1] = 25;
    check_expect(valores[1], 25);
}
```

Observe que o primeiro elemento de um arranjo tem índice 0.

O valor do índice deve estar no intervalo válido para o arranjo, caso contrário, o programa pode não funcionar corretamente. Por padrão, os índices dos arranjos não são verificados antes de serem acessados.

Para habilitar a verificação do índice no acesso aos elementos devemos usar o método `at`.

```
array<int, 4> valores = {10, 4, 9, -1};
check_expec(valores.at(1), 4);
valores.at(1) = 12;
check_expec(valores.at(1), 12);
```

No caso de

```
valores.at(4);
```

o seguinte erro é gerado

```
terminate called after throwing an instance of 'std::out_of_range'
  what():  array::at: __n (which is 4) >= _Nm (which is 4)
Abortado
```

Tanto as estruturas quanto os arranjos são utilizados para representar informações com dois ou mais itens. Então, como escolher qual utilizar?

- Usamos estruturas quando cada item da informação tem uma interpretação particular (na estrutura `Tempo`, temos os componentes `horas`, `minutos` e `segundos`)
- Usamos arranjos quando os itens da informação são da mesma natureza (todos são nomes, notas, etc)

No exemplo da loteria, os itens da aposta e dos resultados têm a mesma natureza, são todos números, então devemos utilizar arranjos ao invés de estruturas. Vamos alterar o código!

```
// Produz true se n é um dos números
// em sorteados, false caso contrário.
bool sorteado(int n, SeisNumeros sorteados)
{
    bool sorteado = false;
    if (n == sorteados.a) {
        sorteado = true;
    }
    if (n == sorteados.b) {
        sorteado = true;
    }
    if (n == sorteados.c) {
        sorteado = true;
    }
    if (n == sorteados.d) {
        sorteado = true;
    }
    if (n == sorteados.e) {
        sorteado = true;
    }
    if (n == sorteados.f) {
        sorteado = true;
    }
    return sorteado;
}
```

```
// Produz true se n é um dos números
// em sorteados, false caso contrário.
bool sorteado(int n, array<int, 6> sorteados)
{
    bool sorteado = false;
    if (n == sorteados[0]) {
        sorteado = true;
    }
    if (n == sorteados[1]) {
        sorteado = true;
    }
    if (n == sorteados[2]) {
        sorteado = true;
    }
    if (n == sorteados[3]) {
        sorteado = true;
    }
    if (n == sorteados[4]) {
        sorteado = true;
    }
    if (n == sorteados[5]) {
        sorteado = true;
    }
    return sorteado;
}
```

```
// Calcula quantos números de aposta estão em sorteados.
int numero_acertos(SeisNumeros aposta, SeisNumeros sorteados)
{
    int acertos = 0;
    if (sorteado(aposta.a, sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta.b, sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta.c, sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta.d, sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta.e, sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta.f, sorteados)) {
        acertos = acertos + 1;
    }
    return acertos;
}
```

```
// Calcula quantos números de aposta estão em sorteados.
int numero_acertos(array<int, 6> aposta, array<int, 6> sorteados)
{
    int acertos = 0;
    if (sorteado(aposta[0], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[1], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[2], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[3], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[4], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[5], sorteados)) {
        acertos = acertos + 1;
    }
    return acertos;
}
```

E então, o código melhorou? Ainda não! Ele continua repetitivo!

Agora vamos trocar a repetição física do código por uma repetição lógica, usando uma nova estrutura de controle. Isso é possível porque os elementos de um arranjo têm a mesma natureza.

Para cada

Em C++, uma das construções de repetição é o “para cada”, que tem a seguinte forma geral

```
for (Tipo nome : arranjo) {
    instruções;
}
```

O “para cada” funciona da seguinte maneira:

- O primeiro valor de `arranjo` é atribuído para `nome` e as `instruções` são executadas;
- O segundo valor de `arranjo` é atribuído para `nome` e as `instruções` são executadas;
- ...
- E assim por diante até que todos os valores de `arranjo` tenham sidos atribuídos para `nome`.

Ou seja, o “para cada” executa as mesmas instruções atribuindo cada valor de `arranjo` para `nome`, por isso ele chama “para cada”!

Para cada

```
// Produz true se n é um dos números
// em sorteados, false caso contrário.
bool sorteado(int n, array<int, 6> sorteados)
{
    bool sorteado = false;
    if (n == sorteados[0]) {
        sorteado = true;
    }
    if (n == sorteados[1]) {
        sorteado = true;
    }
    if (n == sorteados[2]) {
        sorteado = true;
    }
    if (n == sorteados[3]) {
        sorteado = true;
    }
    if (n == sorteados[4]) {
        sorteado = true;
    }
    if (n == sorteados[5]) {
        sorteado = true;
    }
    return sorteado;
}
```

Nesse código, queremos executar as mesmas instruções, uma vez para cada valor de `sorteados`, então, podemos utilizar o “para cada”.

```
// Produz true se n é um dos números em sorteados,
// false caso contrário.
bool sorteado(int n, array<int, 6> sorteados)
{
    bool sorteado = false;
    for (int s : sorteados) {
        if (n == s) {
            sorteado = true;
        }
    }
    return sorteado;
}
```

Para cada

```
// Calcula quantos números de aposta estão em sorteados.
int numero_acertos(array<int, 6> aposta, array<int, 6> sorteados)
{
    int acertos = 0;
    if (sorteado(aposta[0], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[1], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[2], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[3], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[4], sorteados)) {
        acertos = acertos + 1;
    }
    if (sorteado(aposta[5], sorteados)) {
        acertos = acertos + 1;
    }
    return acertos;
}
```

Nesse código, queremos executar as mesmas instruções, uma vez para cada valor de `aposta`, então, podemos utilizar o “para cada”.

```
// Calcula quantos números de aposta estão em sorteados.
int numero_acertos(array<int, 6> aposta, array<int, 6> sorteados)
{
    int acertos = 0;
    for (int num : aposta) {
        if (sorteado(num, sorteados)) {
            acertos = acertos + 1;
        }
    }
    return acertos;
}
```

No exemplo da loteria, vimos como uma repetição física de código pode ser substituída por uma repetição lógica.

Em geral, não precisamos ter uma repetição física de código para depois trocarmos por uma repetição lógica, podemos projetar a função usando uma repetição lógica diretamente.

Vamos ver como fazer isso!

Como projetar funções que processam arranjos usando o “para cada”

Quando precisamos processar um arranjo, geralmente queremos calcular valores de forma incremental, analisando um por um os elementos do arranjo.

Esses valores podem ser o resultado final da função ou podem ser usados em outras instruções para determinar o resultado final da função.

Então, para escrever o código que processa os elementos de um arranjo com o “para cada” precisamos responder três perguntas

1) Quais variáveis (valores) queremos calcular?
2) Como as variáveis são inicializadas?
3) Como as variáveis são atualizadas?

Para responder cada pergunta, usamos os exemplos e perguntas auxiliares.

a) Quais variáveis (valores) queremos calcular?

   Podemos calcular o resultado final da função apenas analisando os elementos do arranjo, sem precisar fazer nada depois?

   Se sim, então esse é o valor que queremos calcular.

   Se não, temos que pensar em valores que podem nos ajudar com a resposta da função e esses são os valores que queremos calcular.

   Em ambos os casos, criamos variáveis para armazenar esses valores.

b) Como as variáveis são inicializadas?

   Se o arranjo não tiver nenhum elemento, qual é o valor esperado para as variáveis?

   Use esses valores para inicializar as variáveis.

c) Como as variáveis são atualizadas?

Pegue alguns exemplos e suponha que o “para cada” já tenha processado de forma correta todos os elementos do arranjo, exceto o último, e determine os valores que as variáveis devem ter.

Em seguida, considere que o último elemento está sendo processado e determine quais operações são necessárias para modificar os valores das variáveis para que elas fiquem com o valor final esperado.

Generalize e escreva o código para fazer essas operações.

Soma

Projete uma função que some todos os valores de um arranjo de 7 números.

Soma

```
// Soma os valores de numeros.
int soma(array<int, 7> numeros)
{
    return 0;
}

examples
{
    check_expect(soma({1, 0, 1, 0, 5, 0, -7}), 0);
    check_expect(soma({1, 0, 1, 0, 5, 0, 10}), 17);
    check_expect(soma({0, -4, 1, 0, -5, -10, 0}), -18);
}
```

1) Quais variáveis (valores) queremos calcular? A soma.
2) Como as variáveis são inicializadas? A soma é inicializada com 0.
3) Como as variáveis são atualizadas? A soma é atualizada somando o elemento atual.

```
// Soma os valores de numeros.
int soma(array<int, 7> numeros)
{
    int soma = 0;
    for (int num : numeros) {
        soma = soma + num;
    }
    return soma;
}
```

Máximo

Projete uma função que encontre o valor máximo de um arranjo com 5 números.

Máximo

```
// Encontra o valor máximo de numeros.
int maximo(array<int, 5> numeros)
{
    return 0;
}

examples
{
    check_expect(maximo({5, 4, 1, 0, 7}), 7);
    check_expect(maximo({1, -5, 8, 1, 2}), 8);
    check_expect(maximo({-4, -6, -1, -1, -3}), -1);
}
```

1) Quais variáveis (valores) queremos calcular? O máximo.
2) Como as variáveis são inicializadas? O máximo é inicializado com o primeiro elemento.
3) Como as variáveis são atualizadas? Se o elemento atual é maior que o máximo, ele passa a ser o máximo.

```
// Encontra o valor máximo de numeros.
int maximo(array<int, 5> numeros)
{
    int max = numeros[0];
    for (int n : numeros) {
        if (n > max) {
            max = n;
        }
    }
    return max;
}
```

Positivos ou negativos

Projete uma função que verifique se um arranjo com 5 números inteiros existem mais positivos ou mais negativos.

Definição dos tipos de dados

```
// O sinal de um número.
enum Sinal {
    Positivo,
    Negativo,
    Nenhum,
}
```

Positivos ou negativos

```
// Verifica se existem mais números positivos ou negativos no arranjo numeros.
// Devolve Positivo se existem mais positivos do que negativos.
// Devolve Negativo se existem mais negativos do que positivos.
// Devolve Nenhum se a quantidade de positivos é igual a de negativos.
Sinal mais_positivos_ou_negativos(array<int, 5> numeros) {
    return Nenhum;
}

examples {
    check_expect(mais_positivos_ou_negativos({1, 2, 3, -1, 0}), Positivo);
    check_expect(mais_positivos_ou_negativos({1, 2, 3, -1, -2}), Positivo);
    check_expect(mais_positivos_ou_negativos({-4, 2, 3, -1, -2}), Negativo);
    check_expect(mais_positivos_ou_negativos({-4, 2, 3, 0, -2}), Nenhum);
}
```

1) Quais variáveis (valores) queremos calcular? A quantidade de positivos e negativos.
2) Como as variáveis são inicializadas? Com zero.
3) Como as variáveis são atualizadas? Se o elemento atual é positivo, incrementa o número de positivos, se o elemento atual é negativo, incrementa o número de negativos.

Positivos ou negativos

```
Sinal mais_positivos_ou_negativos(array<int, 5> numeros)
{
    int num_positivos = 0;
    int num_negativos = 0;
    for (int num : numeros) {
        if (num > 0) {
            num_positivos = num_positivos + 1;
        } else if (num < 0) {
            num_negativos = num_negativos + 1;
        }
    }
    Sinal sinal = Nenhum;
    if (num_positivos > num_negativos) {
        sinal = Positivo;
    } else if (num_negativos > num_positivos) {
        sinal = Negativo;
    }
    return sinal;
}
```

O exemplo da loteria requeria arranjos com 6 elementos.

Mas para esses últimos exemplos, o tamanho fixo do arranjo parece uma imposição artificial.

De fato, o mais comum é problemas que precisam de arranjos de tamanho dinâmico.

Em C++ o tipo arranjo de tamanho dinâmico (ou arranjo dinâmico, ou vetor, ou lista, ou ...) é chamado de `vector` e está disponível na biblioteca `vector`.

Vamos ver as operações básicas com arranjos dinâmicos.

Arranjos dinâmicos

A forma de inicializar arranjos dinâmicos é similar a forma de inicializar arranjos de tamanho fixo, com exceção de que não precisamos especificar a quantidade de elementos.

```
#include <vector>
using namespace std;
examples {
    vector<int> valores = {10, 4, 9, -1};
    vector<string> nomes = {"joao", "jose", "maria"};
```

Assim como para `array`, também acessamos e modificamos os elementos de um `vector` com índices e/ou com o método `at`

```
    check_expect(valores[0], 10);
    check_expect(nomes[2], "maria");

    check_expect(valores.at(1), 4);
    valores.at(1) = 25;
    check_expect(valores[1], 25);
}
```

Tamanho e adição no final

Como `vector` tem tamanho dinâmico, podemos consultar o tamanho (quantidade de elementos) com o método `size`

```
vector<int> idades = {2, 7, 1, 9};

check_expect(idades.size(), 4);
```

Para adicionar um novo elemento no final do arranjo, utilizamos o método `push_back`

```
idades.push_back(4);

check_expect(idades.size(), 5);
check_expect(idades, (vector<int> {2, 7, 1, 9, 4}));
```

ATENÇÃO: quando queremos utilizar um `vector`, um `array` ou uma estrutura como **resultado esperado** em um `check_expect` precisamos colocar o resultado todo entre parênteses e nome do tipo antes de `{`

```
check_expect(..., (array<string, 2> {"casa", "agua"}));
check_expect(..., (vector<int> {4, 10}));
check_expect(..., (Janela {10, 40, 100, 200}));
```

Soma

Utilizamos o mesmo processo para escrever a implementação de funções que processam arranjos de tamanho fixo e arranjos de tamanho dinâmico.

```
// Soma os elementos de valores.
int soma(vector<int> valores)
{
    int soma = 0;
    for (int num : valores) {
        soma = soma + num;
    }
    return soma;
}

examples
{
    check_expect(soma({}), 0);
    check_expect(soma({3, 1}), 4);
    check_expect(soma({4, 6, -2, 0, 1}), 9);
}
```

Uma eleição é realizada com apenas dois candidatos. Cada eleitor pode votar ou no primeiro candidato, ou no segundo candidato, ou ainda, votar em branco. O candidato que tiver mais votos ganha a eleição. Se os votos em branco forem mais do que 50% do total de votos, novas eleições devem ser convocadas. Projete uma função que receba como entrada uma lista não vazia de votos e determine qual foi o resultado da eleição. Dica: deseje uma função auxilar que conte votos de um tipo especificado por parâmetro.

Determinar o resultado de uma eleição.

- O voto pode ser em um de dois candidatos ou em branco;
- Se mais que 50% dos votos forem brancos ou se os candidatos tiverem o mesmo número de votos, é necessário uma nova eleição;
- Se não for necessário uma nova eleição, ganha quem tiver mais votos.

```
// Os votos serão representados por vector<Voto>.

// Um voto em uma eleição.
enum Voto {
    Candidato1,
    Candidato2,
    Branco,
};

// O resultado de uma eleição.
enum ResultadoEleicao {
    Venceu1,
    Venceu2,
    NovasEleicoes,
};
```

```
// Apura o resultado da eleicao considerando os votos no arranjo votos.
// - Produz NovasEleicoes se mais do que 50% do total de votos for branco ou se a
// quantidade de votos do Candidato1 for igual ao do Candidato2.
// - Senão, produz Venceu1 se o Candidato1 teve mais votos ou Venceu2 se o
// Candidato2 teve mais votos.
ResultadoEleicao apura_eleicao(vector<Voto> votos)
{
    return NovasEleicoes;
}

examples
{
    check_expect(apura_eleicao({Candidato1, Candidato2, Candidato1, Branco}), Venceu1);
    check_expect(apura_eleicao({Candidato2, Candidato2, Candidato1, Branco}), Venceu2);
    check_expect(apura_eleicao({Candidato2, Candidato1, Candidato1, Candidato2}), NovasEleicoes);
    check_expect(apura_eleicao({Candidato1, Candidato1, Branco, Branco}), Venceu1);
    check_expect(apura_eleicao({Candidato2, Candidato2, Branco, Branco}), Venceu2);
    check_expect(apura_eleicao({Candidato1, Candidato2, Branco, Branco}), NovasEleicoes);
    check_expect(apura_eleicao({Candidato2, Candidato2, Branco, Branco, Branco}), NovasEleicoes);
}
```

Implementação

Vamos seguir a dica do enunciado e desejar a função `conta_votos`, que conta um determinado tipo de voto.

```
ResultadoEleicao apura_eleicao(vector<Voto> votos) {
    int num_votos1 = conta_votos(votos, Candidato1);
    int num_votos2 = conta_votos(votos, Candidato2);
    int num_brancos = conta_votos(votos, Branco);

    ResultadoEleicao resultado;
    if (num_brancos > votos.size() / 2) {
        resultado = NovasEleicoes;
    } else if (num_votos1 > num_votos2) {
        resultado = Venceu1;
    } else if (num_votos2 > num_votos1) {
        resultado = Venceu2;
    } else {
        resultado = NovasEleicoes;
    }
    return resultado;
}
```

Adicionamos `conta_votos` na lista de pendências e fazemos o projeto da função.

```
// Conta quantos votos no arranjo votos é igual ao alvo.
int conta_votos(vector<Voto> votos, Voto alvo)
{
    return 0;
}

examples
{
    vector<Voto> votos = {Candidato2, Candidato1, Branco, Candidato1, Candidato2, Candidato1};
    check_expect(conta_votos(votos, Candidato1), 3);
    check_expect(conta_votos(votos, Candidato2), 2);
    check_expect(conta_votos(votos, Branco), 1);
}
```

1) Quais variáveis (valores) queremos calcular? A quantidade de votos iguais ao alvo.
2) Como as variáveis são inicializadas? A quantidade é inicializada com 0.
3) Como as variáveis são atualizadas? A quantidade é atualizada em 1 quanto o elemento atual for igual ao alvo.

Verificação: Ok.

Revisão: Ok.

```
int conta_votos(vector<Voto> votos, Voto alvo)
{
    int num = 0;
    for (Voto voto : votos) {
        if (voto == alvo) {
            num = num + 1;
        }
    }
    return num;
}
```

Índice do máximo

Projete uma função que encontre o índice (posição) da primeira ocorrência do valor máximo de um arranjo dinâmico não vazio.

```
// Encontra o índice da primeira ocorrência do valor máximo de valores.
// Requer que valores não seja vazio.
int indice_maximo(vector<int> valores)
{
    return 0;
}

examples
{
    check_expect(indice_maximo({5}), 0);
    check_expect(indice_maximo({5, 5}), 0);
    check_expect(indice_maximo({5, 7, 5}), 1);
    check_expect(indice_maximo({5, 7, 5, 8}), 3);
}
```

1) Quais variáveis (valores) queremos calcular? O índice da primeira ocorrência do máximo (`imax`) e o índice o elemento atual (`i`).
2) Como as variáveis são inicializadas? Os dois valores são inicializados com 0.
3) Como as variáveis são atualizadas? `imax` é atualizado para `i` se o elemento atual for maior que o elemento na posição `imax`. `i` é incrementado com 1.

```
int indice_maximo(vector<int> valores)
{
    int i = 0;
    int imax = 0;
    for (int valor : valores) {
        if (valor > valores[imax]) {
            imax = i;
        }
        i = i + 1;
    }
    return imax;
}
```

Verificação: Ok.

Revisão: Não está claro a relação de `i` com `valor`... podemos mudar `valor` para `valores[i]`.

```
int indice_maximo(vector<int> valores)
{
    int i = 0;
    int imax = 0;
    for (int valor : valores) {
        if (valores[i] > valores[imax]) {
            imax = i;
        }
        i = i + 1;
    }
    return imax;
}
```

E agora? `valor` não é mais utilizado.

A questão é que não queremos mais acessar os elementos do arranjo diretamente, queremos usar um índice para acessar os elementos. Vamos utilizar o laço “para”, que é mais apropriado para essa situação.

Laço “para”

O laço “para” tem a seguinte forma geral

```
for(inicialização; condição; atualização) {
   instruções;
}
```

O funcionamento do “para” é o seguinte

- A inicialização é executada
- Em seguida a condição é verificada, se ela for **`true`** as instruções são executadas, senão o laço termina
- Depois a atualização é executada e a condição é verificada novamente, se ele for **`true`**...

Exemplo - soma

```cpp
// Soma os elementos de valores.
int soma(vector<int> valores)
{
    int soma = 0;
    for (int num : valores) {
        soma = soma + num;
    }
    return soma;
}
```

```cpp
// Soma os elementos de valores.
int soma(vector<int> valores)
{
    int soma = 0;
    for (int i = 0; i < valores.size(); i = i + 1) {
        soma = soma + valores[i];
    }
    return soma;
}
```

Qual das duas implementações é mais simples? A que usa o “para cada”! Por quê?

Porque a variável do laço (`num`) é controlada implicitamente, já no caso do “para” a variável do laço (`i`) é controlado explicitamente.

Mas então, quando usamos o “para” ao invés do “para cada”? Quando queremos mais controle sobre o laço!

Abrimos mão da simplicidade pela flexibilidade.

“para cada”: quando queremos analisar os elementos na sequência.

“para”: para os demais casos (queremos os índices, queremos analisar mais que um elemento de uma vez, não precisamos analisar todos os elementos, etc).

Assim como no “para cada”, precisamos responder as três perguntas

1) Quais variáveis (valores) queremos calcular?
2) Como as variáveis são inicializadas?
3) Como as variáveis são atualizadas?

Mais alguma coisa? Sim! Também precisamos definir os itens do laço

1) Quais são as variáveis do laço e como elas são inicializadas?
2) Qual a condição do laço?
3) Como as variáveis do laço são atualizadas?

Em algumas situações as variáveis do laço são as que queremos calcular.

Índice do máximo

```
int indice_maximo(vector<int> valores)
{
    int i = 0;
    int imax = 0;
    for (int valor : valores) {
        if (valores[i] > valores[imax]) {
            imax = i;
        }
        i = i + 1;
    }
    return imax;
}
```

Fazendo uma “tradução” direta obtemos

```
int indice_maximo(vector<int> valores)
{
    int imax = 0;
    for (int i = 0; i < valores.size(); i = i + 1) {
        if (valores[i] > valores[imax]) {
            imax = i;
        }
    }
    return imax;
}
```

Qual é mais adequada? A que usa o “para”.

Podemos melhorar? Sim, `i` pode começar com `1`, dessa forma não comparamos `valores[0]` com ele mesmo na primeira iteração.

Projete uma função que receba como entrada um arranjo dinâmico de números, uma posição *i* e um número *n* e devolva um novo arranjo com *n* adicionado na posição *i* do arranjo de entrada.

Especificação

```
// Cria um novo arranjo inserindo o valor no índice pos de valores.
// Requer que 0 <= pos <= valores.size().
vector<int> insere_posicao(vector<int> valores, int pos, int valor)
{
    return {};
}

examples {
    check_expect(insere_posicao({}, 0, 10), (vector<int> {10}));
    check_expect(insere_posicao({8}, 0, 10), (vector<int> {10, 8}));
    check_expect(insere_posicao({8}, 1, 10), (vector<int> {8, 10}));
    check_expect(insere_posicao({2, 8}, 0, 3), (vector<int> {3, 2, 8}));
    check_expect(insere_posicao({2, 8}, 1, 3), (vector<int> {2, 3, 8}));
    check_expect(insere_posicao({2, 8}, 2, 3), (vector<int> {2, 8, 3}));
}
```

Implementação

```cpp
vector<int> insere_posicao(
     vector<int> valores,
     int pos,
     int valor)
{
    vector<int> resultado = {};
    for (int i = 0; i < valores.size(); i = i + 1) {
        if (i == pos) {
            resultado.push_back(valor);
        }
        resultado.push_back(valores[i]);
    }

    return resultado;
}
```

Verificação

```
Ran 6 tests.
3 of the 6 tests passed.
Failures
  insere_posicao.cpp at line 35:
    left : std::vector<int> {}
    right: std::vector<int> {10}

  insere_posicao.cpp at line 37:
    left : std::vector<int> {8}
    right: std::vector<int> {8, 10}

  insere_posicao.cpp at line 40:
    left : std::vector<int> {2, 8}
    right: std::vector<int> {2, 8, 3}
```

```
vector<int> insere_posicao(vector<int> valores, int pos, int valor) {
    vector<int> resultado = {};
    for (int i = 0; i < valores.size(); i = i + 1) {
        if (i == pos) {
            resultado.push_back(valor);
        }
        resultado.push_back(valores[i]);
    }
    if (pos == valores.size()) {
        resultado.push_back(valor);
    }
    return resultado;
}
```

Verificação: Ok.

Revisão: o código tem um caso especial... O que podemos fazer? Separar em três etapas, inserir os elementos antes de `pos`, inserir o `valor` em `pos`, inserir os elementos de `pos` até o final.

Implementação

```
vector<int> insere_posicao(vector<int> valores, int pos, int valor)
{
    vector<int> resultado = {};

    for (int i = 0; i < pos; i = i + 1) {
        resultado.push_back(valores[i]);
    }

    resultado.push_back(valor);

    for (int i = pos; i < valores.size(); i = i + 1) {
        resultado.push_back(valores[i]);
    }

    return resultado;
}
```

Projete uma função que receba como entrada um arranjo dinâmico de números e uma posição e devolva um novo arranjo sem o elemento da posição especificada.

```
// Cria um novo arranjo removendo o elemento da posição pos de valores.
// Requer que 0 <= pos < valores.size().
vector<int> remove_posicao(vector<int> valores, int pos)
{
    return {};
}

examples {
    check_expect(remove_posicao({8}, 0), (vector<int> {}));
    check_expect(remove_posicao({2, 8}, 0), (vector<int> {8}));
    check_expect(remove_posicao({2, 8}, 1), (vector<int> {2}));
    check_expect(remove_posicao({7, 2, 8}, 0), (vector<int> {2, 8}));
    check_expect(remove_posicao({7, 2, 8}, 1), (vector<int> {7, 8}));
    check_expect(remove_posicao({7, 2, 8}, 2), (vector<int> {7, 2}));
}
```

```
// Cria um novo arranjo removendo o elemento da posição pos de valores.
// Requer que 0 <= pos < valores.size().
vector<int> remove_posicao(vector<int> valores, int pos)
{
    vector<int> resultado = {};

    for (int i = 0; i < pos; i = i + 1) {
        resultado.push_back(valores[i]);
    }

    for (int i = pos + 1; i < valores.size(); i = i + 1) {
        resultado.push_back(valores[i]);
    }

    return resultado;
}
```

Verificação de ordem

Projete uma função que verifique se um arranjo de valores está em ordem não decrescente.

Especificação

```
// Produz true se os elementos de valores estão em ordem não decrescente, false
// caso contrário.
bool ordem_nao_decrescente(vector<int> valores)
{
    return false;
}

examples {
    check_expect(ordem_nao_decrescente({}), true);
    check_expect(ordem_nao_decrescente({3}), true);
    check_expect(ordem_nao_decrescente({1, 3}), true);
    check_expect(ordem_nao_decrescente({3, 1}), false);
    check_expect(ordem_nao_decrescente({1, 3, 3}), true);
    check_expect(ordem_nao_decrescente({3, 3, 3}), true);
    check_expect(ordem_nao_decrescente({3, 2, 3}), false);
    check_expect(ordem_nao_decrescente({3, 4, 2}), false);
    check_expect(ordem_nao_decrescente({1, 2, 3, 4, 5, 6}), true);
    check_expect(ordem_nao_decrescente({1, 2, 3, 2, 5, 6}), false);
}
```

A implementação dessa função parece ser diferente das anteriores.

Antes só precisávamos analisar um único elemento do arranjo a cada iteração, agora temos que analisar dois elementos.

Como podemos proceder nesse caso?

Vamos resolver o problema para arranjos com 5 elementos usando repetição física de código e depois vamos tentar transformar essa repetição física em uma repetição lógica.

Implementação

```
bool ordem_nao_decrescente(array<int, 5> valores)
{
    bool em_ordem = true;
    if (valores[0] > valores[1]) {
        em_ordem = false;
    }
    if (valores[1] > valores[2]) {
        em_ordem = false;
    }
    if (valores[2] > valores[3]) {
        em_ordem = false;
    }
    if (valores[3] > valores[4]) {
        em_ordem = false;
    }
    return em_ordem;
}
```

1) Quais são as variáveis do laço e como elas são inicializadas? `int i = 1`
2) Qual a condição do laço? `i < valores.size()`
3) Como as variáveis do laço são atualizadas? `i = i + 1`

```
// Produz true se os elementos de valores estão em ordem não decrescente, false
// caso contrário.
bool ordem_nao_decrescente(vector<int> valores)
{
    bool em_ordem = true;
    for (int i = 1; i < valores.size(); i = i + 1) {
        if (valores[i - 1] > valores[i]) {
            em_ordem = false;
        }
    }
    return em_ordem;
}
```

Verificação: Ok.

Revisão: mesmo encontrando valores “fora de ordem” a repetição continua e analisa todos os elementos de `valores`... A repetição só precisa continuar enquanto `em_ordem` for `true`.

Implementação

```
// Produz true se os elementos de valores estão em ordem não decrescente, false
// caso contrário.
bool ordem_nao_decrescente(vector<int> valores)
{
    bool em_ordem = true;
    for (int i = 1; i < valores.size() && em_ordem; i = i + 1) {
        if (valores[i - 1] > valores[i]) {
            em_ordem = false;
        }
    }
    return em_ordem;
}
```

Verificação de palíndromo

Projete uma função que verifique se um arranjo de valores é palíndromo.

Especificação

```
// Produz true se valores é palíndromo, isto é, os elementos de valores vistos
// da esquerda para direita e da direita para esquerda são os mesmos. Produz
// false se valores não é palíndromo.
bool palindromo(vector<int> valores)
{
    return false;
}

examples {
    check_expect(palindromo({}), true);
    check_expect(palindromo({1}), true);
    check_expect(palindromo({2, 1}), false);
    check_expect(palindromo({1, 2}), false);
    check_expect(palindromo({2, 2}), true);
    check_expect(palindromo({2, 1, 2}), true);
    check_expect(palindromo({4, 4, 2}), false);
    check_expect(palindromo({2, 1, 1, 2}), true);
    check_expect(palindromo({2, 0, 1, 2}), false);
    check_expect(palindromo({2, 0, 1, 0, 2}), true);
}
```

Assim como para a função `ordem_nao_decrescente`, vamos resolver o problema para 5 elementos usando repetição física de código.

```
bool palindromo(array<int, 5> valores)
{
    bool palindromo = true;
    if (valores[0] != valores[4]) {
        palindromo = false;
    }
    if (valores[1] != valores[3]) {
        palindromo = false;
    }
    return palindromo;
}
```

1) Quais são as variáveis do laço e como elas são inicializadas?

   `int i = 0, j = valores.size() - 1`

2) Qual a condição do laço?

   `i < j`

3) Como as variáveis do laço são atualizadas?

   `i = i + 1, j = j - 1`

```cpp
// Produz true se valores é palíndromo, isto é, os elementos de valores vistos
// da esquerda para direita e da direita para esquerda são os mesmos. Produz
// false se valores não é palíndromo.
bool palindromo(vector<int> valores)
{
    bool palindromo = true;
    for (int i = 0, j = valores.size() - 1; i < j && palindromo; i = i + 1, j = j - 1) {
        if (valores[i] != valores[j]) {
            palindromo = false;
        }
    }
    return palindromo;
}
```

Exercício

A escola do seu irmão mais novo está fazendo uma coletânea de ditos populares. Cada aluno da escola escolheu um dito popular e a escola agregou todos eles em um arquivo texto (um dito por linha). Agora a escola precisa eliminar os ditos repetidos e classificá-los em ordem, mas ela não sabe como fazer isso. Você pode ajudar?

Classificar uma coleção de ditos em ordem e eliminar ditos repetidos.

- Os ditos devem ser obtidos de um arquivo texto onde cada linha tem um dito.

Devemos fazer um programa completo e não apenas uma função!

Por onde começamos?

Assumimos que temos os dados de entrada e prosseguimos com o projeto de uma função!
Depois fazemos a parte de entrada e saída dos dados.

Mas antes, vamos ver um tipo que irá nos ajudar a resolver o problema: conjuntos.

Um conjunto em C++ podem ser representado pelo tipo `set`, definido na biblioteca `set`.

Um conjunto é semelhante a um arranjo dinâmico, mas não contém elementos repetidos, os elementos são mantidos em ordem e não são indexados.

Conjuntos

```
set<int> s = {3, 1, 4};

check_expect(s, (set<int> {1, 4, 3}));

s.insert(5);
s.insert(4);
s.insert(1);

check_expect(s.size(), 4);
check_expect(s, (set<int> {1, 5, 4, 3}));

s.erase(10);
s.erase(1);
s.erase(3);

check_expect(s, (set<int> {4, 5}));

check_expect(s.count(4), 1);
check_expect(s.count(3), 0);
```

Inserir um elemento existente ou remover um elemento inexistente não altera o conjunto.

Podemos verificar se um conjunto contém um elemento utilizando o método `count`.

Para analisar todos os elementos de um conjunto, usamos o “para cada”

```
set<int> valores = {7, 10, 2};

int soma = 0;

for (int valor : valores) {
    soma = soma + valor;
}

check_expect(soma, 19);
```

É isso! Agora vamos voltar para o problema dos ditos.

Os ditos de entrada serão representados por um arranjo de strings.

Vamos usar um conjunto na implementação...

Especificação

```
// Seleciona uma ocorrência de cada dito do arranjo ditos e classifica o
// resultado em ordem alfabética.
vector<string> classifica_ditos_unicos_em_ordem(vector<string> ditos) {
    return {};
}

examples
{
    string dito1 = "Esmola demais até santo desconfia";
    string dito2 = "Diga com quem andas que lhe direi quem és";
    string dito3 = "Saco vazio não para em pé";
    check_expect(classifica_ditos_unicos_em_ordem({}),
                 (vector<string> {}));

    check_expect(classifica_ditos_unicos_em_ordem({dito1, dito2, dito1}),
                 (vector<string> {dito2, dito1}));

    check_expect(classifica_ditos_unicos_em_ordem({dito3, dito1, dito2, dito1, dito3, dito2}),
                 (vector<string> {dito2, dito1, dito3}));
}
```

Precisamos analisar todos os elementos da entrada e podemos fazer isso na ordem que eles aparecem, então pode usar o “para cada”.

O que queremos calcular com o “para cada”?

Calcular a resposta diretamente parece complicado pois teríamos que evitar as repetições de ditos e ainda classificar o arranjo de saída.

Podemos computar um resultado intermediário mais facilmente que nos ajude a computar o resultado da função? Sim! Vamos criar um conjunto com os ditos, eles serão únicos e estarão em ordem! E depois?

Criamos o arranjo de resposta com os elementos do conjunto.

Implementação

```
// Seleciona uma ocorrência de cada dito do arranjo ditos e classifica o
// resultado em ordem alfabética.
vector<string> classifica_ditos_unicos_em_ordem(vector<string> ditos)
{
    // Usamos um conjunto para selecionar apenas uma ocorrência de cada dito e
    // também classificá-los em ordem.
    set<string> unicos = {};
    for (string dito : ditos) {
        unicos.insert(dito);
    }

    vector<string> result = {};

    for (string dito : unicos) {
        result.push_back(dito);
    }

    return result;
}
```

Temos a função que calcula o que queremos, e agora, o que precisamos fazer para ter um programa?

Escrever o código que faz a entrada e saída. Vamos desejar por funções auxiliares e escrever a função principal.

```
int main()
{
    // Entrada
    vector<string> ditos = le_ditos();

    // Processamento
    vector<string> ditos_unicos = classifica_ditos_unicos_em_ordem(ditos);

    // Saída
    exibe_ditos(ditos_unicos);
}
```

```cpp
// Produz um arranjo de ditos lendo um dito por linha da entrada padrão.
vector<string> le_ditos()
{
    return {};
}

// Exibe na saída padrão os ditos.
void exibe_ditos(vector<string> ditos)
{
}
```

O que essas funções têm de diferente das funções que temos escrito até agora?

A função `le_ditos` não tem argumentos de entrada e a função `exibe_ditos` não tem resposta (usamos o tipo `void` para representar isso).

Por isso não temos como escrever os exemplos para essas funções!

A implementação da função `exibe_ditos` é direta:

```
// Escreve na saída padrão os ditos.
void exibe_ditos(vector<string> ditos)
{
    for (string dito: ditos) {
        cout << dito << endl;
    }
}
```

Mas e a implementação da função `le_ditos`? Ler um dito parece simples

```
vector<string> le_ditos()
{
    vector<string> ditos = {};
    string dito;

    getline(cin, dito);
    ditos.push_back(dito);

    return ditos;
}
```

Mas como ler uma quantidade indeterminada de ditos? Precisamos de uma nova forma de repetição!

Enquanto

A forma geral do “enquanto” é:

```
while (condição) {
    instruções;
}
```

O funcionamento do “enquanto” é o seguinte

- A condição é verificada;
- Se a condição for **`true`**, as instruções são executadas e processo começa novamente.
- Se a condição for **`false`**, o laço é finalizado.

Ou seja, o “enquanto” executa as mesmas instruções enquanto a condição for verdadeira.

Exemplo

```
vector<int> valores = {1, 2, 4};

int i = 0;
int soma = 0;
while (i < valores.size()) {
    soma = soma + valores[i];
    i = i + 1;
}

check_expect(soma, 7).
```

Quando usar o “enquanto”?

Claro, não precisamos do “enquanto” para fazer a soma dos valores de um arranjo, podemos usar o “para cada”.

Então, quando usamos o “enquanto”?

Em geral, quando precisamos de uma repetição que não dependa de uma coleção de valores. Mas também podemos usar o “enquanto” quando precisamos analisar os elementos de um arranjo e o “para” e o “para cada” não são adequados (deixam o código mais complicado).

Agora podemos voltar para a implementação de `le_ditos`.

A função `getline` produz um valor que pode ser usado com uma condição. Se `getline` conseguir ler uma linha da entrada, o valor produzido corresponde a `true`, caso contrário, o valor corresponde a `false`.

Então, a ideia para implementar a função `le_ditos` é:

- Enquanto conseguiu ler uma linha, adiciona a linha no arranjo.

```
// Produz um arranjo de ditos lendo um dito por linha da entrada padrão.
vector<string> le_ditos()
{
    vector<string> ditos = {};
    string dito;

    while (getline(cin, dito)) {
        ditos.push_back(dito);
    }

    return ditos;
}
```

O programa está pronto! Agora podemos compilar e testar.

Quando o programa é executado ele fica esperando os ditos serem digitados, um por linha. Para sinalizar que não serão digitados mais ditos, pressionamos “crtl + d”.

Compile e teste o programa dessa forma. Qual a sua impressão sobre a “facilidade” de uso do programa?

Parece repetitivo ter que digitar os ditos, afinal, estes ditos já foram coletados e estão salvos em algum arquivo.

Quando temos um arquivo `.txt`, podemos utilizar o seu conteúdo como entrada do nosso programa. Nesse caso, tudo o que está no arquivo aparece para o programa como se tivesse sido digitado pelo usuário.

Se o arquivo com os ditos chama `"ditos.txt"`, podemos utilizar o conteúdo do arquivo como entrada para o programa da seguinte forma

```
./ditos < ditos.txt
```

O símbolo `<` é interpretado pelo shell como redirecionamento da entrada, então, ao invés de esperar o usuário digitar a entrada para o programa, o shell utiliza o conteúdo do arquivo como entrada do programa.

Da mesma forma que existe redirecionamento de entrada, também existe redirecionamento de saída.

Quando utilizamos redirecionamento de saída, ao invés do resultado do programa ser exibido na tela, ele é salvo em um arquivo.

Para ler a entrada do arquivo `"ditos.txt"` e salvar o resultado no arquivo `"ditos-unicos.txt"`, executamos

```
./ditos < ditos.txt > ditos-unicos.txt
```

O símbolo `>` é interpretado pelo shell como redirecionamento da saída.

Momento de apreciação

Aprecie esse momento. Temos um programa completo que pode ser utilizado por um usuário final para realizar uma tarefa útil!

Exercício

Projete uma função que separe as “partes” de uma string usando um espaço como delimitador.

```
// Produz uma lista das "partes" de s usando um espaço como delimitador.
vector<string> separa(string s)
{
    return {};
}

examples
{
    check_expect(separa(""), (vector<string> {}));
    check_expect(separa("casa"), (vector<string> {"casa"}));
    check_expect(separa("Seu Jorge cantou a musica."),
                 (vector<string> {"Seu", "Jorge", "cantou", "a", "musica."}));
}
```

Como fazer a implementação? Vamos precisar de uma repetição, mas qual tipo? Diferente das repetições que fizemos anteriormente, esta não depende de uma "coleção" de valores, então vamos usar o "enquanto".

E como fazer a repetição? Vamos tentar generalizar a solução de alguns exemplos específicos.

```
string s = "Seu Jorge cantou a musica.";
vector<string> palavras = {};

s.find(" ", 0); // a partir do indice 0 ("S"), encontra o indice do primeiro " ", que é 3
palavras.push_back(s.substr(0, 3 - 0)); // "Seu"

s.find(" ", 3 + 1); // a partir do indice 4 ("J"), encontra o indice do primeiro " ", que é 9
palavras.push_back(s.substr(4, 9 - 4)); // "Jorge"

s.find(" ", 9 + 1); // a partir do indice 10 ("c"), encontra o indice do primeiro " ", que é 16
palavras.push_back(s.substr(10, 16 - 10)); // "cantou"

s.find(" ", 16 + 1); // a partir do indice 17 ("a"), encontra o indice do primeiro " ", que é 18
palavras.push_back(s.substr(17, 18 - 17)); // "a"

s.find(" ", 18 + 1); // a partir do indice 18 ("m"), não tem mais " ", produz -1.
palavras.push_back(s.substr(17, 26 - 19)); // "musica." 26 é o tamanho de s
```

Quais informações (variáveis) são necessárias a cada iteração? O índice de início (`inicio`) e o valor produzido pela busca do espaço (`fim`).

Como as variáveis são inicializadas? `inicio = 0`.

O que precisa ser feito a cada iteração? Calcular `fim`, extrair e adicionar a substring em `palavras`. O que mais? Atualizar o `inicio` para a próxima iteração!

Todas as iterações são iguais? Não, a última iteração é diferente. Como identificar se estamos na última iteração? `find` produz `-1`.

Como o `inicio` é atualizado? No caso geral, `inicio = fim + 1`. E no último caso? `inicio = s.length()`.

Qual a condição para o laço continuar a execução? `inicio < s.length()`

```cpp
// Separa uma string usando um espaço como delimitador.
vector<string> separa(string s)
{
    vector<string> palavras = {};
    int inicio = 0;
    int fim;
    while (inicio < s.length()) {
        fim = s.find(" ", inicio);
        if (fim != -1) {
            palavras.push_back(s.substr(inicio, fim - inicio));
            inicio = fim + 1;
        } else {
            palavras.push_back(s.substr(inicio, s.length() - inicio));
            inicio = s.length();
        }
    }
    return palavras;
}
```

Podemos melhorar? Sim! Como `fim` é usado apenas dentro do laço, mudamos o local da declaração.

Exercício

Modifique a função `separa` de maneira que a string seja separada por um ou mais espaços:

```
check_expect(separa("mais   de  um "), (vector<string> { "mais", "de", "um" }));
```

Exercício

Um número inteiro positivo n é primo se ele tem exatamente dois divisores distintos, 1 e n. Projete uma função que verifique se um número inteiro positivo é primo. Dica: Faça exemplos de código (sem repetição lógica) para verificar se alguns números específicos (5, 8, 11) são primos e tente generalizar o código com repetição lógica usando o "enquanto". Leia com cuidado a definição e faça o código mais simples e direto possível!

Especificação

```
// Produz true se n é um número primo, isto é, se n tem exatamente dois
// divisores distintos, 1 e ele mesmo. Produz false se n não é primo.
//
// Requer que n seja maior que 0.
bool primo(int n)
{
    return false;
}

examples
{
    check_expect(primo(1), false);
    check_expect(primo(2), true);
    check_expect(primo(3), true);
    check_expect(primo(4), false);
    check_expect(primo(5), true);
    check_expect(primo(8), false);
    check_expect(primo(11), true);
}
```

Como fazer a implementação? Generalizando soluções específicas!

Implementação

```
int n = 5;
int num_divisores = 0;

if (n % 1 == 0) {
    num_divisores = num_divisores + 1;
}
if (n % 2 == 0) {
    num_divisores = num_divisores + 1;
}
if (n % 3 == 0) {
    num_divisores = num_divisores + 1;
}
if (n % 4 == 0) {
    num_divisores = num_divisores + 1;
}
if (n % 5 == 0) {
    num_divisores = num_divisores + 1;
}
return num_divisores == 2; // 2 == 2 -> true
```

Implementação

```
bool primo(int n)
{
    int num_divisores = 0;
    int i = 1;
    while (i <= n) {
        if (n % i == 0) {
            num_divisores = num_divisores + 1;
        }
        i += 1;
    }
    return num_divisores == 2;
}
```

Podemos melhorar! Sim! (Discussão em sala)

Implementação

```
bool primo(int n)
{
    bool primo = n > 1;
    for (int i = 2; i <= n / 2 && primo; i = i + 1) {
        if (n % i == 0) {
            primo = false;
        }
    }
    return primo;
}
```