Curso de C++ 2019.1

Bem vindo! Utilize a barra de navegação para navegar pelo material das aulas.

Licenças

Aula 1 - Compilação e Sintaxe Básica

Boas vindas ao curso de C++ de 2019.1!

Bem vindos à primeira aula de C++! Nela abordaremos:

- Sintaxe básica de fluxo de programa (`if`, `while`, `for` e `range-for`);
- Alguns dos tipos primitivos mais comuns, como `int` e `double`;
- O processo de geração de um executável (preprocessamento, compilação, montagem e linkagem);
- Definição básica de funções.

Além disso, iremos conhecer os tipos `std::string`, para texto, e `std::vector`, para coleções.

Um programa simples em C++

Vamos iniciar com um programa bastante simples em C++, utilizando números inteiros e um pouquinho de texto, e imprimindo algumas informações na tela:

```
#include <iostream>  // std::cout
#include <string>  // std::string

int main()
{
    int year = 2019;
    int age = 26;
    std::string name = "Tarcísio";
    std::cout << "Hello, " << name
              << ". I see you were born in "
              << year - age << "!\n";
}
```

Iniciamos nosso código com a seguinte diretiva:

```
#include <iostream>  // std::cout
#include <string>  // std::string
```

Em C++, a diretiva `#include` é a forma de incluir em nosso código as declarações necessárias para utilizar código externo. No caso, estamos incluindo o *header* `iostream`, que é a parte da biblioteca padrão da linguagem que permite lidar com *input* e *output*. Com ela, poderemos escrever dados na tela durante a execução do programa.

Na sequência, temos a introdução de uma **função**, chamada `main`:

```
int main()
{
```

Veremos essa função em todos os executáveis que criarmos. Ela demarca o ponto de início de execução do código, do ponto de vista do programador. Veremos posteriormente como alterá-la para lidar com argumentos quando necessário, mas no momento `()` significa que nenhum parâmetro é passado para o programa.

Dentro da função `main`, encontramos nossas primeiras *variáveis*, que servem para representar os dados com os quais o programa trabalhará. Variáveis em C++ possuem um tipo e um valor. O tipo de uma variável nunca muda, mas seu valor pode mudar durante a execução do programa.

```
    int year = 2019;
    int age = 26;
    std::string name = "Tarcísio";
```

A variável `year` é uma variável do tipo `int`, que é utilizado para representar números inteiros com sinal. Desta forma, um ano anterior ao ano 0 poderia ser representado como um número negativo, por exemplo.

A variável `age`, por sua vez, representa uma idade, que não pode ser negativa. Mesmo assim, utilizamos `int`, para garantir que possamos fazer aritmética com `year` sem problemas.

Enquanto `year` e `age` são de tipos ditos *primitivos*, ou seja, tipos suportados diretamente pela linguagem, `name` é de um tipo *composto*, ou seja, implementado em C++, utilizando tipos primitivos em sua implementação. No caso, `name` é do tipo `std::string`, que utilizaremos sempre que quisermos representar texto.

A implementação de `std::string` fica na biblioteca padrão de C++, que é um conjuto de código pronto que sempre acompanha o compilador, implementando uma série de funcionalidades comuns para dar suporte à criação de novos programas e bibliotecas. É para usá-la que incluímos `string` via `#include` no início do código.

A variável `name`, por sua vez, é de um tipo não-primitivo, ou seja, um tipo definido em C++, em algum lugar da biblioteca padrão. Esse é o tipo `std::string`, que utilizaremos para

representar texto.

Finalmente, temos as linhas a seguir:

```
    std::string name = "Tarcísio";
    std::cout << "Hello, " << name
              << ". I see you were born in "
              << year - age << "!\n";
}
```

A sequência de usos do operador `<<` utiliza o objeto `std::cout` para escrever na tela. `std::cout` representa a *saída padrão* do programa, que em geral é o terminal onde ele é executado.

Com isso, conseguimos fazer um programa simples em C++ que já faz uma breve interação com o usuário e um pouco de aritmética. Na sequência, veremos como faremos para vê-lo em ação.

O processo de compilação

Agora que vimos um programa simples, precisamos ser capazes de gerar um executável para vê-lo em ação. Iremos então entender como esse processo ocorre, passo a passo.

Compilando um programa simples

Para compilarmos o nosso programa `age.cpp`, iremos utilizar o seguinte comando:

```
g++ -o age age.cpp
```

Este comando irá gerar um executável chamado `age` a partir do nosso código. Ao executá-lo, veremos a seguinte saída:

```
$ g++ -o age age.cpp
$ ./age
Hello, Tarcísio. I see you were born in 1993!
```

Os estágios de compilação

Um programa em C++ passa por pelo menos 5 estágios durante sua compilação, detalhados a seguir.

Preprocessamento

O primeiro estágio é o **preprocessamento**. Esse estágio na realidade vem antes da compilação propriamente dita: o preprocessador é um programa de manipulação de texto que não leva em consideração a linguagem em que o código está escrito. Ele apenas manipula texto de acordo com diretivas que vemos no código iniciando com `#`, como `#include` ou `#ifndef`. Para observar o resultado do preprocessamento, pode-se utilizar o comando `cpp` (C Preprocessor), ou `g++ -E`.

Estes comandos jogarão o resultado para a saída padrão. Para gerar um arquivo com a saída, usamos a opção `-o`. A extensão mais comum para arquivos já preprocessados é `.i` (porém, não é muito comum que estes arquivos sejam vistos pelo programador).

```
cpp age.cpp -o age.i
```

```
g++ -E age.cpp -o age.i
```

Se você abrir o arquivo resultante, verá que ao final de uma grande quantidade de linhas, há o conteúdo do nosso código, exceto pelas diretivas `#include`, que foram substituídas pelo conteúdo dos cabeçalhos da biblioteca padrão:

```
// (...) Grande trecho de código da biblioteca padrão
# 2 "src/lesson_1/age.cpp" 2



# 4 "src/lesson_1/age.cpp"
int main()
{
    int year = 2019;
    int age = 26;
    std::string name = "Tarcísio";
    std::cout << "Hello, " << name
              << ". I see you were born in "
              << year - age << "!\n";
}
```

Compilação

Na sequência ocorre o passo que chamamos **compilação** propriamente dita: o código em C++ é compilado para *assembly*, ou **linguagem de montagem**. Podemos observar este estágio utilizando a opção `-S` com o `g++`. Novamente a saída é dada na saída padrão. Para gerar um arquivo, utilizamos a opção `-o`.

```
g++ -S age.cpp -o age.S
```

ou, partindo de onde paramos antes

```
g++ -S age.i -o age.S
```

A linguagem de montagem depende do tipo de processador para o qual estamos compilando (geralmente x86_64) e tem mais ou menos a seguinte forma:

```
(...)
.LC0:
    .string     "Tarc\303\255sio"
.LC1:
    .string     "Hello, "
.LC2:
    .string     ". I see you were born in "
.LC3:
    .string     "!\n"
    .text
    .globl      main
    .type       main, @function
main:
.LFB1493:
    .cfi_startproc
    .cfi_personality 0x9b,DW.ref.__gxx_personality_v0
    .cfi_lsda 0x1b,.LLSDA1493
    pushq       %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq        %rsp, %rbp
    .cfi_def_cfa_register 6
    pushq       %rbx
    subq        $72, %rsp
    .cfi_offset 3, -24
    movq        %fs:40, %rax
    movq        %rax, -24(%rbp)
    xorl        %eax, %eax
    movl        $2019, -72(%rbp)
    movl        $26, -68(%rbp)
    leaq        -73(%rbp), %rax
    movq        %rax, %rdi
    call        _ZNSaIcEC1Ev@PLT
    leaq        -73(%rbp), %rdx
    leaq        -64(%rbp), %rax
    leaq        .LC0(%rip), %rsi
    movq        %rax, %rdi
.LEHB0:
    call
_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1EPKcRKS3_@PLT
.LEHE0:
    leaq        -73(%rbp), %rax
    movq        %rax, %rdi
    call        _ZNSaIcED1Ev@PLT
    leaq        .LC1(%rip), %rsi
    leaq        _ZSt4cout(%rip), %rdi
(...)
```

Montagem

Neste passo o *assembly* é **montado**, gerando código de máquina binário. Isto gera o código que chamamos *object code* ou **código objeto**. Para visualizarmos este passo temos a opção `-c`.

```
g++ -c age.cpp -o age.o
```

ou partindo do ponto anterior

```
g++ -c age.S -o age.o
```

Perceba que se abrirmos o código objeto num editor de texto comum vemos diversos caracteres estranhos: o formato agora é binário, e não textual. Podemos ver o conteúdo do arquivo de forma mais legível com o comando `objdump -CD` (sugestão: jogue a saída para o comando `less` para poder navegar, pois a saída é grande).

```
objdump -CD main.o | less
```

A flag `-C` é especial para C++ e faz com que nomes sejam mostrados como no código, com namespaces corretos e afins. Na saída do `objdump` conseguimos ver o código binário como hexadecimal à esquerda, e o *assembly* correspondente à direita.

```
(...)
Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   53                      push   %rbx
   5:   48 83 ec 48             sub    $0x48,%rsp
   9:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  10:   00 00
  12:   48 89 45 e8             mov    %rax,-0x18(%rbp)
  16:   31 c0                   xor    %eax,%eax
  18:   c7 45 b8 e3 07 00 00    movl   $0x7e3,-0x48(%rbp)
  1f:   c7 45 bc 1a 00 00 00    movl   $0x1a,-0x44(%rbp)
  26:   48 8d 45 b7             lea    -0x49(%rbp),%rax
  2a:   48 89 c7                mov    %rax,%rdi
  2d:   e8 00 00 00 00          callq  32 <main+0x32>
  32:   48 8d 55 b7             lea    -0x49(%rbp),%rdx
  36:   48 8d 45 c0             lea    -0x40(%rbp),%rax
  3a:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 41 <main+0x41>
  41:   48 89 c7                mov    %rax,%rdi
  44:   e8 00 00 00 00          callq  49 <main+0x49>
  49:   48 8d 45 b7             lea    -0x49(%rbp),%rax
  4d:   48 89 c7                mov    %rax,%rdi
  50:   e8 00 00 00 00          callq  55 <main+0x55>
  55:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 5c <main+0x5c>
  5c:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 63 <main+0x63>
  63:   e8 00 00 00 00          callq  68 <main+0x68>
  68:   48 89 c2                mov    %rax,%rdx
  6b:   48 8d 45 c0             lea    -0x40(%rbp),%rax
  6f:   48 89 c6                mov    %rax,%rsi
  72:   48 89 d7                mov    %rdx,%rdi
  75:   e8 00 00 00 00          callq  7a <main+0x7a>
  7a:   48 8d 35 00 00 00 00    lea    0x0(%rip),%rsi        # 81 <main+0x81>
  81:   48 89 c7                mov    %rax,%rdi
(...)
```

Linkagem

O processo final de geração do executável é a **linkagem** (que admitidamente não é um verbo real em português). Neste passo, os códigos objeto de todas as unidades de tradução envolvidas no programa são unidos em um só. Como no momento estamos trabalhando com programas de apenas um arquivo, revisitaremos a linkagem adiante quando trabalharmos com código dividido em mais de um arquivo.

Para finalizarmos a geração de nosso executável, basta rodar:

```
g++ -o age age.o
```

Será gerado um executável de nome `age`, que é o mesmo programa que geramos com o comando `g++ -o age age.cpp`. Neste exemplo bastante simples, isso tudo parece uma complicação desnecessária, e realmente é. De fato, no dia-a-dia, exceto o comando `g++ -c`

para gerar código objeto, os outros não costumam ser utilizados com frequência. Em programas maiores, porém, com processos de compilação mais complexos, com diversos arquivos, e principalmente, em que nem todos os .cpp estão no mesmo diretório, a geração de código objeto intermediária é importante e bastante comum.

Diagrama

Por fim, para uma visualização mais "gráfica", temos aqui um diagrama. Este diagrama já presume um programa dividido em diversos arquivos.

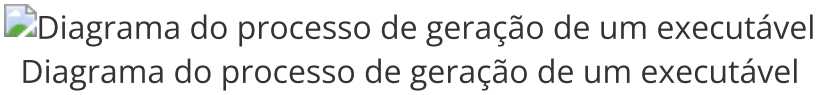

Diagrama do processo de geração de um executável

A construção `if/else if/else`

Aqui temos um código baseado em uma condição. Temos abaixo uma nota, representada como um `double`. Baseado no valor dessa nota, o código imprime uma mensagem relacionada.

```
#include <iostream>

int main()
{
    auto grade = 5.0;

    if (grade < 5.75) {
        std::cout << "Sorry, not this time.\n";
    } else if (grade < 8.0) {
        std::cout << "It's ok.\n";
    } else {
        std::cout << "Congratulations on that!\n";
    }
}
```

O conteúdo dos parênteses logo após a palavra `if` é chamado *condição*. A condição sempre deve ser algo do qual possa ser extraído um *valor verdade*, também conhecido como *valor booleano*, ou seja, `true` (verdadeiro), ou `false` (falso).

```
    if (grade < 5.75) {
```

Nesta linha, `grade < 5.75` irá ser visto como `true`, pois no nosso exemplo `grade == 5.0`, que é menor do que `5.75`.

> **Nota**: O tipo que contém os valores `true` e `false` em C++ chama-se `bool`. Outros tipos, como `int`, podem ser *convertidos implicitamente* para `bool` em alguns casos.

No exemplo de `int`, `0` torna-se `false`, enquanto outros valores se tornam `true`.

A construção `if` executa o código contido no bloco (entre as chaves `{}`) quando a condição tem valor `true`. Portanto, sabemos que nesse código o que executará é a seguinte linha:

```
        std::cout << "Sorry, not this time.\n";
```

Nota: Experimente alterar o valor, recompilar e reexecutar para ver o resultado!.

O bloco dado por `else if` funciona de maneira semelhante. A diferença é que ele funciona de maneira *mutuamente exclusiva*. Ou seja, **mesmo que a condição do `else if` seja verdadeira**, se o bloco do `if` executou, o bloco do `else if` não irá executar.

Por fim, o bloco da parte dada por `else` executa se nenhum outro dos blocos executar. Para que ele execute, é necessário portanto que `grade` seja alterado para um valor maior do que `8.0`.

A seguir, veremos uma construção semelhante ao `if`, mas para repetir operações: a construção `while`.

`auto`

Você deve ter percebido que a variável `grade` foi declarada utilizando `auto`. Este, porém, não é o tipo dela: ela é do tipo `double`, que é utilizado para representar números de ponto-flutuante, (números decimais com número de casas arbitrário).

O especificador `auto` permite **inferir o tipo de uma variável no ponto de sua inicialização**. Com ele, não é necessário repetir a informação do tipo de uma variável: ela é dada pelo tipo do valor utilizado para inicializá-la.

É importante notar que, mesmo declarada com **auto**, uma variável não tem como mudar de tipo durante a execução do programa, ou seja, `grade` é um double e pode assumir apenas valores `double` enquanto existir.

Utilizar `auto` frequentemente nos traz duas principais vantagens:

1. Não temos como declarar variáveis sem inicializá-las, pois ao declarar com `auto` é necessário inicializar para que o tipo possa ser deduzido.
2. Não há repetição de informação no que escrevemos. Ao escrevermos `double grade = 4.0;`, a informação de que a variável é um `double` existe tanto no nome `double` como no valor `5.0`, que já é um `double` para a linguagem. Logo, não é

necessário se repetir. já que o compilador é capaz de inferir que, para conter `5.0`, grade necessita ser `double`.

O laço `while`

É muito comum, ao programar, ser necessário repetir alguma operação diversas vezes, até alcançar alguma condição. Para isso utilizaremos *laços*. Veremos 3 tipos de laços. O primeiro, e mais simples, é o laço `while`. O código nesse laço irá rodar enquanto uma dada condição for verdadeira. Veja o código a seguir:

```
#include <iostream>  // std::cout
#include <cmath>  // sqrt

double square_root(double x)
{
    auto old = 0.0;
    auto guess = 1.0;

    while (guess != old) {
        old = guess;
        guess = (guess + x/guess)/2;
    }
}

int main()
{
    std::cout << square_root(2) << "\n";
    std::cout << sqrt(2) << "\n";

    return 0;
}
```

Vamos analisar o `while` parte a parte:

```
    while (guess != old) {
```

A condição do `while` é dada de forma semelhante ao `if`. Ela deve ser um valor booleano (`true` ou `false`). O corpo do laço executará repetidas vezes, até que, **ao final** de uma das execuções, a condição deixe de ser verdadeira.

O código acima calcula, de forma numérica, a raiz quadrada de um número qualquer `x`. Isso é feito utilizando o método de Newton.

O método de Newton se baseia em um chute inicial. Esse chute pode ser maior, menor ou igual ao valor da raiz quadrada.

Após fazer um chute inicial (no nosso caso, arbitrariamente fixado em `1.0`), é feita a média do valor chutado com `x` dividido pelo chute, nas linhas a seguir:

```
        old = guess;
        guess = (guess + x/guess)/2;
```

Vamos acompanhar as possibilidades:

Se o chute for muito abaixo da resposta real, `x/guess` será muito acima da resposta real (pois a raiz quadrada é o número que, ao dividir `x`, resulta nele mesmo), e a média entre os dois se aproximará da resposta. Por exemplo:

- Seja `x = 36`.
- Seja `guess = 3` (abaixo do valor correto, `6`).
- `x/guess == 12`, ou seja, acima do valor correto.
- A média entre `guess` e `x/guess` será `7.5`, aproximando-se da resposta correta.

O exato oposto também pode acontecer: se `x` for maior que a resposta correta, `x/guess` será menor que a resposta e a média se aproximará do valor procurado.

- Seja `x = 36`.
- Seja `guess = 15` (acima do valor correto, `6`).
- `x/guess == 2.4`, ou seja, abaixo do valor correto.
- A média entre `guess` e `x/guess` será `8.7`, também aproximando-se da resposta correta.

O resultado será usado como um novo chute, e o chute anterior é registrado na variável `old`.

Perceba a condição do while: ele se quebra quando o chute anterior for igual ao novo chute. Isso ocorrerá quando a resposta for a correta, pois `guess` será a raiz quadrada de `x`, logo `x/guess` será igual a `guess`, e também à média.

No geral, o laço while é uma forma ideal de representarmos a repetição de operações quando não sabemos o número necessário de repetições, e sim uma condição booleana de parada. Quando nós sabemos quantos passos dar, é mais interessante utilizarmos o laço `for`. Que veremos a seguir.

Antes disso, porém, vamos prestar atenção na seguinte parte do código:

```
double square_root(double x)
{
   // (...)
}
```

Com este código estamos definindo uma *função*. Funções são a unidade mais básica de **reuso de código**. O código de uma função pode ser *chamado* em outros pontos do programa.

```
//     tipo de retorno     nome do primeiro argumento
//    /                   /
double square_root(double x)
//          \       \
//           nome   tipo do primeiro argumento
```

Aqui temos a declaração da função. Nela definimos o seguinte:

1. O *tipo de retorno* da função. Toda função necessita retornar algo de um tipo específico e predefinido.
2. O nome da função, para que ela possa ser utilizada posteriormente.
3. Os argumentos da função. Todo argumento possui um tipo e um nome, e fica disponível no corpo da função, como qualquer outra variável.

Funções são utilizadas em outros pontos do código através de chamadas, como podemos ver na seguinte linha:

```
    std::cout << square_root(2) << "\n";
```

> Nota: Não é coincidência que a sintaxe de chamada de função é muito semelhante à notação de aplicação de funções na matemática. O conceito de funções em programação guarda diversas semelhanças ao conceito de funções matemáticas.

Visto isso, vamos então ao laço `for`.

O laço `for`

Outro caso muito comum em que precisamos executar um código repetidamente é quando precisamos fazer isso um número específico de vezes, ou uma vez para cada elemento em uma coleção de valores. Para estes casos, os laços do tipo `for` são a forma mais direta de escrever o código.

Em C++, temos dois tipos de laço `for`: o `for` normal (clássico), e o `range-for`, introduzido em C++11. Veremos os dois na sequência.

O `for` clássico

```
#include <iostream>
#include <vector>  // std::vector

double average(std::vector<double> grades)
{
    auto sum = 0.0;
    auto size = grades.size();

    for (auto i = 0u; i < size; ++i) {
        sum += grades[i];
    }

    return sum/size;
}


int main()
{
    auto grades = std::vector<double>{9.0, 8.0, 4.0};

    std::cout << average(grades) << '\n';
}
```

O primeiro laço `for` é a versão mais "clássica". Ele é visto em diversas outras linguagens, como C ou Java. Dentro dos parênteses, temos as seguintes partes:

```
//            1         2        3
//            |         |        |
    for (auto i = 0u; i < size; ++i) {
```

1. **Inicialização**: Este campo executa apenas uma vez, antes do laço iniciar.
2. **Condição**: O laço executará enquanto esta condição for verdadeira, como acontece no `while`.
3. **Passo**: Este código executará ao final do corpo do laço, todas as vezes que o corpo também executar.

Nota: Utilizamos `0u` nesse for porque `grades.size()` retorna `std::size_t`, que é um tipo `unsigned`. Em geral, evitaremos aritmética com tipos `unsigned`, mas abriremos uma exceção quando formos obrigados a utilizar `std::size_t`.

Qualquer uma dessas partes pode ser vazia. Se a condição for vazia, o laço roda infinitamente. Com a inicialização e o passo vazio, podemos escrever um laço `while` na forma de `for`, assim como o inverso é possível.

O laço for também garante que a variável inicializada na contagem existe apenas durante o escopo do `for`, e seu nome pode ser reutilizado posteriormnete no código.

Com isso, conseguimos ir da posição `0` até a posição `size-1` (sequências em C++ tem 0 como seu primeiro índice), e somar todas as notas, para posteriormente dividi-las pelo número de notas e obter a média.

O `range-for`

```
#include <iostream>
#include <vector>  // std::vector

double average(std::vector<double> grades)
{
    auto sum = 0.0;

    for (auto i: grades) {
        sum += i;
    }

    return sum/grades.size();
}


int main()
{
    auto grades = std::vector<double>{9.0, 8.0, 4.0};

    std::cout << average(grades) << '\n';
}
```

O segundo laço é o *range-for*. Este `for` precisa de um objeto sobre o qual ele possa iterar - no nosso caso, um `std::vector`.

```
//          2         1
//          |         |
    for (auto i: grades) {
```

O `range-for` tem uma estrutura mais rígida. Sempre haverá:

1. Um objeto a ser iterado sobre.
2. Uma variável que irá representar, em cada iteração do laço, o **elemento atual** do objeto.

Utilizando o *range-for* dessa forma conseguimos chegar ao mosmo resultado que com o `for` clássico.

O mecanismo que permite o funcionamento do `range-for` será introduzido posteriormente, quando estudarmos *iterators*.

Exercícios

1. Escreva um laço que passe **por todos** os números de 1 a 100, mas imprima apenas os pares.
    - Dica: o operador `%` pode ser utilizado para obter o resto da divisão.
2. Faça um laço baseado no da questão anterior. Neste laço, para cada número, imprima:
    - `fizz` se o número for divisível por 3
    - `buzz` se o número for divisível por 5
    - `fizzbuzz` se o número for divisível por ambos
    - O próprio número, em todos os outros casos.
    - Lembre de **não** imprimir o número nos três primeiros casos.
3. Utilizando algum dos tipos de laço vistos nesta aula, **escreva funções** que imprimam os valores da sequência de Fibonacci, das seguintes formas:
    - Dado um `n`, até o `n`-ésimo elemento da sequência.
    - Dado um `x`, até o último elemento da sequência que seja menor ou igual a `x`

    Dica: Considere que o `n` do primeiro elemento é 0, e que fib(0) = 0 e fib(1) = 1.

    Utilize o seguinte `main`:

    ```
    // implemente sua função aqui

    int main()
    {
        fib_up_to(10);
        fib_less_than(1024);
    }
    ```
4. Utilizando algum dos tipos de laço vistos em aula, **escreva uma função** que, dada uma std::string qualquer, **retorne** `palindrome` se ela for um palíndromo e `not a palindrome` se ela não for um palíndromo.

    Palíndromo: Uma string que é igual a si mesma se lida de trás para frente, como `arara`.

    - Para acessar cada caractere da string, pode se usar `s[i]` para uma `string s` e uma posição `i`, ou pode-se usar o *range-based* `for` na string.
    - Lembre-se que as posições na `string` iniciam-se em 0.

    Utilize o seguinte `main`:

```cpp
#include <iostream>
#include <string>

// implemente sua função aqui

int main()
{
    using namespace std::string_literals;

    std::cout << "arara is "s << check_palindrome("arara"s) << '\n';
    std::cout << "banana is "s << check_palindrome("banana"s) << '\n';
}
```

Aula 2 - `std::vector` e Parâmetros de Função

Nesta segunda aula, veremos um dos tipos mais utilizados da biblioteca padrão, o tipo `std::vector`. Utilizamos `vector` quando queremos agrupar diversos elementos de um mesmo tipo. O tipo `vector` é muito utilizado pois sequências de valores de tamanho variável são extremamente comuns em programação.

A seguir, veremos mais a fundo como funcionam **funções**, as quais vimos brevemente na primeira aula. Iremos ver como passar **parâmetros** para funções, e a diferença entre **valores** e **referências**. Por fim, veremos como funciona o qualificador `const`, que será muito importante daqui em diante.

O que é um `std::vector`?

```
#include <iostream>
#include <vector>  // std::vector


int main()
{
    auto v = std::vector<int>{};

    v = {1, 2, 3, 4};

    std::cout << "in v:\n";

    for (auto i: v) {
        std::cout << i << '\n';
    }

    auto w = std::vector<int>{4, 3, 2, 1};

    std::cout << "in w:\n";

    for (auto i: w) {
        std::cout << i << '\n';
    }
}
```

`std::vector` representa em C++ um conceito chamado *vetor*. Um vetor é o que chamamos de "coleção". Ele serve como um recipiente para vários elementos do mesmo tipo. O tipo de coleção que um vetor representa é o que chamamos de "sequência", pois a ordem dos elementos é mantida e importa.

`vector` difere um tanto dos tipos primitivos e do tipo `string` por ser *parametrizado*, e portanto sua declaração vai ser um pouco diferente.

Um tipo parametrizado, também conhecido como um **template de classe** ("*class template*") não é realmente um tipo, e sim uma "fábrica de tipos". Ele recebe parâmetros através da sintaxe de parênteses angulados (`<>`). O tipo `vector` recebe até dois parâmetros, mas para a aula de hoje vamos nos preocupar apenas com o primeiro: o tipo dos elementos do vetor.

```
    auto v = std::vector<int>{};
```

Podemos colocar diversos elementos em v com o que chamamos de **lista de inicialização**, ou "*initializer list*". Isso é uma funcionalidade de C++ moderno (C++11 e adiante), então não se surpreenda em ver código mais complexo para fazer esta mesma coisa, especialmente em código antigo:

```
    v = {1, 2, 3, 4};
```

Isso também pode ser feito diretamente ao declarar o vetor:

```
    auto w = std::vector<int>{4, 3, 2, 1};
```

Sinta-se à vontade para alterar a linha acima. Insira elementos, remova elementos, altere elementos. Sempre recompile e veja o resultado. Tente colocar elementos que não sejam do tipo `int`, como `double`s ou mesmo `std::string`s.

Acessando e alterando elementos

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


int main()
{
    auto v = std::vector<int>{1, 2, 3, 4};
    auto w = std::vector<int>{4, 3, 2, 1};

    std::cout << "v[0] = " << v[0] << ", w[2] = " << w[2] << "\n";

    std::cout << "v.size() = " << v.size() << "\n";

    std::cout << "v.front() = " << v.front() << "\n";

    std::cout << "v.back() = " << v.back() << "\n";

    std::vector<int> q = {};

    std::cout << "q.empty() = " << q.empty() << "\n";

    if (q.empty()) {
        std::cout << "q is empty\n";
    } else {
        std::cout << "there's something inside q\n";
    }
}
```

Acessando elementos

Os elementos de um vetor podem ser acessados primariamente pelo operador `[]`:

```
    std::cout << "v[0] = " << v[0] << ", w[2] = " << w[2] << "\n";
```

Perceba que o primeiro elemento é o de índice 0. Isso não deve ser uma novidade muito grande para quem já conhece outras linguagens de programação, mas pode causar um

pouco de confusão de início. Por causa disso, se um vetor tiver n elementos, o último tem o índice n-1. Ou seja, o último elemento de `w` é `w[3]`, pois `w` possui tamanho 4.

Vetores são **objetos**. Em C++, objetos são todos os valores que não são de tipos primitivos. Vetores não são os primeiros objetos vistos por nós: `std::string`s também são objetos.

Uma característica principal de objetos é agruparem diversas informações. Um vetor, por exemplo, é capaz de informar o seu tamanho, ou seja, de quantos objetos ele é composto no momento, com a função-membro `size`.

```
    std::cout << "v.size() = " << v.size() << "\n";
```

Funções-membro, também chamadas de "métodos", são rotinas de código atreladas ao objeto. Veremos mais adiante o conceito de função, função-membro e como criar objetos. Por agora, apenas mantenha esses nomes em um canto da mente :)

Outros métodos interessantes de acesso em `vector` são os seguintes:

`front()`: obtém o primeiro elemento do vetor (o elemento "da frente").

```
    std::cout << "v.front() = " << v.front() << "\n";
```

`back()`: obtém o último elemento do vetor (o elemento "de trás").

```
    std::cout << "v.back() = " << v.back() << "\n";
```

Outras informações podem ser adquiridas com `std::vector`, por exemplo, verificar se ele está vazio (ou seja, com 0 elementos) com `empty`:

```
    std::cout << "q.empty() = " << q.empty() << "\n";
```

Note que `empty` retorna um `bool`, e pode ser utilizado em estruturas condicionais (`if`):

```
    if (q.empty()) {
        std::cout << "q is empty\n";
    } else {
```

Tente inicializar `q` com algum elemento para mudar o comportamento de `empty`.

Alterando elementos

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


int main()
{
    auto v = std::vector<int>{1, 2, 3, 4};
    auto w = std::vector<int>{4, 3, 2, 1};

    std::cout << "Before: v[3] = " << v[3] << "\n";

    v[3] = 42;

    std::cout << "Now: v[3] = " << v[3] << "\n";

    auto n = v.size();

    std::cout << "n = v.size() = " << n << "\n";

    std::cout << "Before: v[n-1] = " << v[n-1] << "\n";

    v.push_back(777);

    n = v.size();  // Atualiza tamanho do vetor

    std::cout << "n = v.size() = " << n << "\n";
    std::cout << "Now: v[n-1] = " << v[n-1] << "\n";

    std::cout << "v[n-2] = " << v[n-2] << "\n";

    v.pop_back();

    n = v.size();

    std::cout << "Popped vector's back\n";
    std::cout << "n = v.size() = " << n << "\n";
    std::cout << "Now: v[n-1] = " << v[n-1] << "\n";

    std::cout << "v[n-2] = " << v[n-2] << "\n";

    auto strings = std::vector<std::string>{ "Hello", "World", "!" };

    std::cout << "strings = {\""
              << strings[0] << "\", \""
              << strings[1] << "\", \""
              << strings[2] << "\"}\n";

    std::cout << "With strings' contents we can say... you had it coming:\n";
    std::cout << strings[0] << ", " << strings[1] << strings[2] << "\n";

    // strings.push_back(3);
}
```

Da mesma forma que é possível acessar um elemento no vetor com o operador `[]`, é possível substituir o elemento naquele ponto:

```
    std::cout << "Before: v[3] = " << v[3] << "\n";

    v[3] = 42;

    std::cout << "Now: v[3] = " << v[3] << "\n";
```

Também é possível adicionar novos elementos ao final do vetor, utilizando `push_back(...)`:

```
    auto n = v.size();

    std::cout << "n = v.size() = " << n << "\n";

    std::cout << "Before: v[n-1] = " << v[n-1] << "\n";

    v.push_back(777);

    n = v.size();  // Atualiza tamanho do vetor

    std::cout << "n = v.size() = " << n << "\n";
    std::cout << "Now: v[n-1] = " << v[n-1] << "\n";

    std::cout << "v[n-2] = " << v[n-2] << "\n";
```

Da mesma forma, é possível remover o último elemento do vetor, através do `pop_back`:

```
    v.pop_back();

    n = v.size();

    std::cout << "Popped vector's back\n";
    std::cout << "n = v.size() = " << n << "\n";
    std::cout << "Now: v[n-1] = " << v[n-1] << "\n";

    std::cout << "v[n-2] = " << v[n-2] << "\n";
```

Por fim, por `std::vector` ser um tipo parametrizado, é possível criar vectors de diferentes tipos, como por exemplo, `std::string`:

```
    auto strings = std::vector<std::string>{ "Hello", "World", "!" };

    std::cout << "strings = {\""
              << strings[0] << "\", \""
              << strings[1] << "\", \""
              << strings[2] << "\"}\n";

    std::cout << "With strings' contents we can say... you had it coming:\n";
    std::cout << strings[0] << ", " << strings[1] << strings[2] << "\n";
```

E isso mantém todas as características de um `vector`, incluindo `size`, `push_back`, `front`, e demais métodos demonstrados anteriormente.

Note que não é possível inserir um `int` em `strings`. Descomente a linha abaixo para ver isso, pois ela não compila.

```
    // strings.push_back(3);
```

A tipagem parametrizável de `vector` ocorre durante a compilação. Desta forma, `std::vector<int>` e `std::vector<std::string>` são tipos completamente diferentes para o compilador, e erros como o acima são facilmente evitados.

> **Nota**: Há também operações como remover elementos do meio de um vetor, porém estas lidam com conceitos que serão vistos mais adiante (especificamente, iterators) e portanto só serão vistas quando tais conceitos estiverem devidamente explicados

---

Cuidado especial - `std::vector<bool>`

Um aviso: o tipo `std::vector<bool>` não deve ser utilizado. Entenderemos melhor o que ocorre com ele que o torna uma má ideia, mas por enquanto é suficiente saber que, diferentemente de qualquer outro `std::vector`, `std::vector<bool>`, em uma tentativa de otimização de espaço, não se comporta exatamente como esperado em algumas operações.

Por sorte, `vector<bool>` não é realmente algo de uso muito corriqueiro. É bastante raro aparecer necessidade, e existem soluções melhores, como `std::bitset`.

As leituras complementares 3 e 4 trazem alguns detalhes sobre este assunto. É interessante lê-las novamente quando já tivermos visto alguns assuntos mais complexos, como templates.

Parâmetros de função: valores e referências

Valores

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


namespace fibo {

std::vector<int> to_n(int n)
{
    auto fibs = std::vector<int>{};

    auto f1 = 0;
    auto f2 = 1;

    while (n != 0) {
        fibs.push_back(f1);
        auto aux = f1;
        f1 = f2;
        f2 += aux;
        --n;
    }

    return fibs;
}

}


int main()
{
    auto n = 10;
    auto fibs = fibo::to_n(n);

    std::cout << n << '\n';
}
```

Primeiramente, precisamos entender o que são parâmetros passados por **valor**. Veja a função seguir:

```
std::vector<int> to_n(int n)
{
    auto fibs = std::vector<int>{};

    auto f1 = 0;
    auto f2 = 1;

    while (n != 0) {
        fibs.push_back(f1);
        auto aux = f1;
        f1 = f2;
        f2 += aux;
        --n;
    }

    return fibs;
}
```

Na função acima, o único parâmetro é `n`. Ele é passado por **valor**, o que significa que quando a função é chamada, ele é inicializado com base no que está entre parênteses, mas é **uma variável totalmente diferente**. Portanto, alterar `n` dentro da função não afeta de forma alguma o valor original:

```
    auto n = 10;
    auto fibs = fibo::to_n(n);

    std::cout << n << '\n';
```

O código acima imprimirá `10`, mesmo que `to_n` altere o parâmetro `n` em seu código. Então, um parâmetro passado por valor dentro da função é uma entidade *independente* do parâmetro no ponto da chamada, e passar um parâmetro por valor, em condições normais, não o altera. Veremos algumas exceções à regra eventualmente, mas, quando isso ocorrer, será claramente denotado.

Referências

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


namespace vectors {

void zero_all(std::vector<int>& ints)
{
    for (auto& i: ints) {
        i = 0;
    }
}

}



int main()
{
    auto v = std::vector<int>{1, 2, 3, 4};

    vectors::zero_all(v);

    for (auto i: v) {
        std::cout << i << '\n';
    }
}
```

Vejamos agora um exemplo do conceito oposto, um parâmetro recebido por **referência**:

```
void zero_all(std::vector<int>& ints)
{
    for (auto& i: ints) {
        i = 0;
    }
}
```

O parâmetro `ints` é uma referência, o que efetivamente significa que ele **se refere** ao objeto original. Não há a criação de uma nova entidade. O parâmetro **é** o mesmo valor do ponto de chamada, e alterações feitas a partir dele refletirão no valor original.

Veja também a linha

```
    for (auto& i: ints) {
```

Aqui, ao declararmos a variável que será usada no `for`, também usamos uma referência. O mesmo conceito se aplica. se não utilizássemos uma referência aqui, os valores originais em `ints` não seriam alterados.

Logo, ao executar:

```
    auto v = std::vector<int>{1, 2, 3, 4};

    vectors::zero_all(v);

    for (auto i: v) {
        std::cout << i << '\n';
    }
```

Veremos:

```
0
0
0
0
```

Ou seja, o objeto passado no ponto de chamada foi modificado, o que é verificado logo na sequência.

Agora experimente com o código acima. Sugestões:

- Primeiro, imprima o conteúdo de `ints` dentro de `zero_all`, após o `for` que os modifica.
- Com a alteração anterior, experimente trocar os usos de referências por valores, tanto no parâmetro quanto no `for`.

Parâmetros de função: const

O mecanismo de passagem por valor

```
#include <iostream>
#include <numeric>  // std::accumulate
#include <vector>


namespace value {

double average_squares(std::vector<double> values)
{
    for (auto& i: values) {
        i *= i;
    }

    return std::accumulate(begin(values), end(values), 0.0)/values.size();
}

}


int main()
{
    auto values = std::vector<double>(1000000);

    std::iota(begin(values), end(values), 1.0);

    std::cout << value::average_squares(values) << '\n';
}
```

Conforme vimos na seção passada, usamos **referências** se *pretendemos alterar o valor enviado como parâmetro* e **valores** quando não.

Vamos então entender como um parâmetro por valor funciona nesse caso. Vejamos um exemplo que aproveita o parâmetro passado por valor para computar:

```
double average_squares(std::vector<double> values)
{
    for (auto& i: values) {
        i *= i;
    }

    return std::accumulate(begin(values), end(values), 0.0)/values.size();
}
```

Como vimos antes, essa o parâmetro passado a essa função fica intacto após a chamada. Para que isso seja possível, o que está ocorrendo é que o parâmetro é **inicializado por cópia**.

Salvo em exceções que veremos adiante, tomar parâmetros por valor implica em uma cópia, que pode ser uma operação cara em um tipo que seja grande, ou possua uma semântica complexa.

Sabendo disso, podemos ficar tranquilos em tomar por valor parâmetros de tipos simples e pequenos, como `int` ou `double`. Mas ao lidar com parâmetros que podem ter uma cópia cara, se tudo que precisamos é fazer **leitura** do parâmetro, gostaríamos então de evitar essa cópia.

O que sabemos até agora

```
#include <iostream>
#include <string>


namespace ref {

bool is_palindrome(std::string& s)
{
    auto i = s.size() - 1;

    for (auto j = 0u; j < s.size()/2; ++j) {
        if (s[j] != s[i]) {
            return false;
        }
        --i;
    }

    return true;
}

}


int main()
{
    using namespace std::string_literals;

    const auto word = "arara"s;

    std::cout << word << " is a palindrome: " << ref::is_palindrome(word) <<
'\n';
}
```

Sabemos que receber o parâmetro por valor, no caso em que vimos, implica em uma cópia para não alterar o original. Ora, na seção anterior vimos que podemos passar parâmetros por referência. Referências efetivamente representam um acesso direto ao objeto original. Portanto, ao usá-las, evitamos uma cópia.

Em C++, porém, existe o conceito de **constantes**. Em contraste com as **variáveis** que vimos até agora, constantes são valores nomeados que **nunca mudam**. Sabendo que esses valores nunca mudam, é possível tanto para o programador quanto para o compilador tomar decisões que não seriam possíveis sem essa certeza.

No nosso exemplo acima, temos uma constante, definida com o qualificador `const`:

```
    const auto word = "arara"s;
```

Infelizmente, em nossa função, baseado apenas na **assinatura** da função, não há nenhuma maneira de garantir que o parâmetro não será modificado. E como veremos quando fizermos programas com mais de um arquivo, o compilador nem sempre tem acesso ao **corpo** da função. Portanto, ao tentar compilar, recebemos um erro:

```
src/lesson_2/palindrome.cpp: In function ‘int main()’:
src/lesson_2/palindrome.cpp:48:69: error: binding reference of type
‘std::__cxx11::string&’ {aka ‘std::__cxx11::basic_string<char>&’} to ‘const
std::__cxx11::basic_string<char>’ discards qualifiers
     std::cout << word << " is a palindrome: " << ref::is_palindrome(word) <<
'\n';
                                                                     ^~~~
src/lesson_2/palindrome.cpp:7:6: note:   initializing argument 1 of ‘bool
ref::is_palindrome(std::__cxx11::string&)’
 bool is_palindrome(std::string& s)
      ^~~~~~~~~~~~~
```

> **Assinatura** é o nome que se dá ao nome de uma função em conjunto com os tipos aceitos como seus parâmetros. **Corpo** é como chamamos o código que define o que a função faz (o que está dentro das chaves `{ ... }`).

Sabendo disso, portanto, precisamos de uma forma de garantir ao compilador que nossa função não altera seu parâmetro.

O qualificador `const` em referências

```
#include <iostream>
#include <string>


namespace const_ref {

bool is_palindrome(const std::string& s)
{
    auto i = s.size() - 1;

    for (auto j = 0u; j < s.size()/2; ++j) {
        if (s[j] != s[i]) {
            return false;
        }
        --i;
    }

    return true;
}

}


int main()
{
    using namespace std::string_literals;

    const auto word = "arara"s;

    std::cout << word << " is a palindrome: " << const_ref::is_palindrome(word)
<< '\n';
}
```

Para podermos lidar com o problema que surgiu, então, podemos utilizar **referências para constantes**. Estas referências permitem utilizar o benefício de acessar diretamente o valor original enquanto garantimos não alterar o parâmetro que foi passado à função. Conseguimos agora compilar e executar:

```
arara is a palindrome: 1
```

Evitando a cópia no primeiro exemplo

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


namespace const_ref {

double average_squares(const std::vector<double>& values)
{
    auto total = 0.0;

    for (auto i: values) {
        total += i*i;
    }

    return total/values.size();
}

}

int main()
{
    auto values = std::vector<double>(1000000);

    std::iota(begin(values), end(values), 1.0);

    std::cout << const_ref::average_squares(values) << '\n';
}
```

No primeiro exemplo, porém, não basta adicionar `const` e `&`. Veja o que ocorre se tentarmos:

```
namespace const_ref {

double average_squares(const std::vector<double>& values)
{
    for (auto& i: values) {
        i *= i;
    }

    return std::accumulate(begin(values), end(values), 0.0)/values.size();
}

}
```

```
src/lesson_2/average_squares.cpp:48:8: error: redefinition of ‘double
const_ref::average_squares(const std::vector<double>&)’
 double average_squares(const std::vector<double>& values)
        ^~~~~~~~~~~~~~~
src/lesson_2/average_squares.cpp:22:8: note: ‘double
const_ref::average_squares(const std::vector<double>&)’ previously defined here
 double average_squares(const std::vector<double>& values)
        ^~~~~~~~~~~~~~~
src/lesson_2/average_squares.cpp: In function ‘double
const_ref::average_squares(const std::vector<double>&)’:
src/lesson_2/average_squares.cpp:51:14: error: assignment of read-only reference
‘i’
         i *= i;
              ^
```

Porém, o uso do parâmetro original para computar não é realmente necessário. Podemos acumular os valores um a um em uma variável local e escrever o código da seguinte maneira:

```
namespace const_ref {

double average_squares(const std::vector<double>& values)
{
    auto total = 0.0;

    for (auto i: values) {
        total += i*i;
    }

    return total/values.size();
}

}
```

E está feito: conseguimos garantir que não ocorre uma cópia, ao mesmo tempo em que suportamos computar sobre constantes.

Exercícios

1. Escreva uma função que receba uma string e verifique o balanceamento de três tipos de parênteses: `()`, `[]` e `{}`. Parênteses estão balanceados quando:

    1. Da esquerda pra direita, todo fechamento é precedido por uma abertura: todo `)` tem um `(` anterior correspondente.
    2. Todo parênteses aberto é fechado: `()` está balanceado, `(()` não.
    3. Todo fechamento de parênteses fecha o tipo correspondente à última abertura: `([])` é ok, `([)]` não.

    Utilize um `std::vector` em sua implementação.

    Utilize o seguinte `main`:

```
int main()
{
    using namespace std::string_literals;

    auto cases = std::vector<std::string>{
        "int main(int argv, char** argv) { std::cout << argv[0] << '\n';
}"s,
        "([[]]{[]}{()})"s,
        ""s,
        ")"s,
        "([)()(])({}{)(})"s,
        "[[](){](()"s,
    };

    for (auto c: cases) {
        std::cout << "Case " << c << " is "
                  << (balanced(c) ? "balanced" : "not balanced")
                  << '\n';

    }
}
```

2. Escreva uma função que receba um caractere (`char`) e uma `string` e retorne o número de vezes que o caractere aparece na string. **Lembre-se: copiar uma string pode ser custoso.** A string não deve ser alterada de forma alguma.

   Utilize o seguinte main:

```
int main()
{
    using namespace std::string_literals;

    const auto word = "evidentemente"s;
    const auto letter = 'e';
    std::cout << "number of " << letter << " in " << word << ": "
              << count_occurences(letter, word);
}
```

3. Escreva uma função que troca dois inteiros de lugar.

   Utilize o seguinte main:

```
int main()
{
    auto i = 2, j = 3;
    swap(i, j);
    std::cout << i << "\n"; // deve imprimir 3
    std::cout << j << "\n"; // deve imprimir 2
}
```

4. Escreva uma função que receba um vetor de inteiros e remova duplicatas adjacentes,, ou seja, sequências do mesmo valor.

   Exemplo:

```
{1, 1, 1, 2, 3, 3, 4, 1, 1} -> {1, 2, 3, 4, 1}
```

Retorne o resultado e não altere a entrada.

Utilize o seguinte main:

```
int main()
{
    for (auto i: remove_duplicates({1, 1, 1, 2, 3, 3, 4, 1, 1})) {
        std::cout << i << '\n';
    }
}
```

5. [Desafio] Faça o mesmo exercício que o acima, mas em vez de retornar o resultado, altere o vetor de entrada.

   Utilize o seguinte main:

```
int main()
{
    auto ints = std::vector<int>{1, 1, 1, 2, 3, 3, 4, 1, 1} ;

    remove_duplicates(ints);

    for (auto i: ints) {
        std::cout << i << '\n';
    }
}
```

Apêndice 1: Inicialização de vector à moda antiga

Inicializando um vetor da forma "antiga":

```
#include <vector>


int main()
{
    std::vector<int> old;

    old.push_back(1);
    old.push_back(2);
    old.push_back(3);
    old.push_back(4);
}
```

É isso. Não havia uma forma mais cômoda :(.

Aula 3 - Classes e Múltiplos Arquivos

Chegamos à terceira aula, onde finalmente aprenderemos uma das coisas mais importantes em C++: como definir nossos próprios tipos. Para isso, temos em C++ classes. Vários conceitos envolvidos na definição de classes são semelhantes à outros que já vimos antes, então, se necessário, revise o material anterior, ou consulte-o em caso de dúvida.

Veremos também como podemos dividir nosso código em mais de um arquivo, para uma melhor organização de nossos projetos, e também introduziremos aos nossos programas `namespaces`, que vimos brevemente nas aulas anteriores.

Classes - Visão geral

```
namespace math {

class Vector2D {
public:
    Vector2D(double x, double y);
    Vector2D() = default;
    Vector2D(const Vector2D &) = default;

    double x() const;
    double y() const;

    Vector2D &invert();
    Vector2D &multiply(double);
    Vector2D &sum(const Vector2D &);

private:
    double x_{0}, y_{0};
};

Vector2D inverted(Vector2D);
Vector2D sum(Vector2D, const Vector2D &);
Vector2D multiply(Vector2D, double);

}
```

Nosso primeiro exemplo de classe será um **vetor**. Atenção aqui: não estamos falando do vetor como uma sequência de elementos: aqui estamos falando de um vetor bidimensional, como o conceito usado em álgebra linear ou física.

Escolhemos um vetor 2D como exemplo de classe pois ele é mais complexo que um tipo primitivo, mas mesmo assim possui operações simples para descrevermos. Ele permitirá explorar tudo que pretendemos explicar sobre classes nesta aula.

Para começarmos, veremos como formar a organização lógica do nosso código, utilizando **namespaces**. Um namespace nada mais é que uma coleção de entidades de um programa: funções, tipos, variáveis globais, etc. Já utilizamos diversas vezes o namespace mais conhecido: `std`, que engloba todos os elementos da biblioteca padrão, como `vector`,

`string` e `cout`, e vimos outros em exemplos da aula anterior. Criar um namespace é bastante simples:

```
namespace math {
```

Pronto! Tudo o que for declarado aqui dentro faz parte do namespace `math`, até o `}` final do namespace. Com isso, todo o conteúdo do arquivo a partir daqui, em código de fora do namespace, deve ser utilizado utilizando o prefixo `math::`. Veremos também formas de simplificar isso quando necessário mais adiante.

Agora, vamos à definição de nossa classe. Para criar uma classe utilizamos a keyword `class`. Uma classe introduz um tipo no programa, ou seja, após a definição da classe ela pode ser utilizada como um tipo como os que já vimos antes. Por exemplo, `std::string` é uma classe, apesar de sua definição ser mais complexa que a classe que mostraremos aqui.

```
class Vector2D {
```

A partir deste ponto, o tipo `math::Vector` já existe para o nosso programa. Um tipo, por sua vez, possui uma **interface**. É importante, porém, que cuidemos com essa palavra: interface é um termo com diversos significados quando falamos de programação. O que queremos dizer aqui é o seguinte: a interface de um tipo define como como outras partes do progarama o utilizam. Como estamos introduzindo no programa um tipo completamente novo, somos responsáveis por definir duas coisas:

- Como ele funciona.
- Como ele é utilizado.

Por termos a responsabilidade de definir estas duas coisas, C++ traz a possibilidade de separar detalhes explicitamente, com o conceito de `public` e `private`. Elementos internos do tipo que estamos definindo (chamados **membros**) podem ser públicos, ou seja, acessíveis para quem utiliza o tipo, ou privados, ou seja, acessíveis somente dentro da definição interna do tipo. A isso damos o nome de **visibilidade**. Iniciamos a seção pública da classe com a seguinte linha:

```
public:
```

Na seção `public` veremos todos os *métodos* da classe `Vector`. Os métodos de uma classe são funções, semelhante às que vimos na aula 3, mas diretamente ligadas aos objetos da classe. Já vimos métodos em objetos da clase `std::string` e `std::vector`, como `size()`, `empty()` e `push_back()`.

As primeira funções públicas que veremos nessa classe são seus **construtores**. Os construtores definem como um valor do tipo da classe é inicializado. Nesta classe temos três construtores:

```
    Vector2D(double x, double y);
    Vector2D() = default;
    Vector2D(const Vector2D &) = default;
```

Os construtores serão explicados na seção seguinte.

Na sequência definiremos métodos que descrevem as operações básicas que podem ser feitas com um `Vector2D`.

```
    double x() const;
    double y() const;

    Vector2D &invert();
    Vector2D &multiply(double);
    Vector2D &sum(const Vector2D &);
```

Veremos métodos na seção seguinte a construtores.

Por fim, temos os **atributos** da classe `Vector2D`. Como os atributos definem o objeto e precisam ser mantidos consistentes pelos métodos, eles não fazem parte da interface pública de uma classe, e sim são **privados**. Nosso `Vector2D` possui duas coordenadas num espaço bidimensional, `x` e `y`, que são `double`s. Para evitar conflito com os métodos, utilizamos os nomes `x_` e `y_`.

```
private:
    double x_{0}, y_{0};
};
```

Com isso, terminamos a **definição** de nossa classe. Sabemos que a classe está **definida** pois apesar de não conhecermos ainda a implementação de seus métodos, sabemos **os tipos dos atributos que a compõem**.

Ainda dentro do namespace `math`, temos algumas funções que operam utilizando `Vector2D`. Essas funções são funções como as que já vimos na aula anterior, mas como podemos notar, elas não tem corpo. Veremos mais adiante nesta aula como implementar as funções em um outro arquivo.

Quando temos apenas o tipo de retorno e a assinatura de uma função, temos uma **declaração**. A funções com corpo, damos o nome de **definição**. Toda definição também é uma declaração, mas o contrrário não é verdade. Esses conceitos irão gradualmente se demonstrar bastante importantes.

Vamos agora à primeira parte de nossas classes: os atributos e construtores.

Classes - Atributos e construtores

Atributos

Com vimos anteriormente, ao final da definição de nossa classe temos os **atributos**, ou **variáveis-membro**, do objeto. Em geral, eles são mantidos na parte privada da classe, e são acessíveis apenas pelos métodos dos próprios objetos. Os atributos do objeto são, na maior parte das vezes, o motivo de criarmos um novo tipo: agrupar informações que compõem um conceito que não é diretamente representado por tipos primitivos ou outros tipos já disponíveis.

```
private:
    double x_{0}, y_{0};
```

No tipo `Vector2D` temos apenas dois membros, `x_` e `y_`. Os nomes possuem um underscore no fim para não conflitarem com os nomes dos métodos que dão acesso (apenas de leitura) para eles. Note também o valor entre `{}` depois de cada atributo. Este valor será utilizado na inicialização pelo construtor padrão (explicado em detalhes a seguir). Qualquer construtor de `Vector` que não tenha outro valor especificado para inicializar `x_` e `y_` utilizará os valores especificados aqui.

Os atributos de nosso `Vector2D` definem seu estado e são tudo o que realmente é necessário para ter um objeto desse tipo. De fato, para o compilador, um `Vector2D` nada mais é do que *dois double que estão sempre em conjunto*.

Construtores

Para um tipo nos ser útil, é necessário que sejamos capazes de criar, ou **instanciar**, valores que pertençam a ele. Para que possamos fazer a inicialização dos valores criados corretamente, temos os **construtores**.

Como vimos antes, temos três construtores:

```
    Vector2D(double x, double y);
    Vector2D() = default;
    Vector2D(const Vector2D &) = default;
```

- Um que recebe dois `double`, que irá inicializar os atributos do `Vector2D`.
- Um que recebe zero atributos, conhecido como **construtor padrão**, cuja implementação ainda é desconhecida.
- Um que recebe outro `Vector2D` por referência para constante, chamado **construtor de cópia**, responsável por inicializar um `Vector2D` igual a outro já existente.

Classes - Métodos

Os **métodos**, ou **funções-membro** de `Vector2D` definirão as operações que podem ser feitas com os objetos do tipo.

Os primeiros métodos que serão definidos servem para podermos acessar os valores internos do vetor. Não queremos deixar os atributos públicos, por diversos motivos. O mais comum deles é que não queremos que eles possam ser alterados por algum código de fora da classe, para que o controle do estado do objeto fique sujeito apenas aos métodos da classe. Para isso, fazemos o que chamamos de *accessors* ou *getters*:

```
    double x() const;
    double y() const;
```

Em C++, é comum *getters* serem nomeados de acordo com o nome do atributo que expõem, porém sem quaisquer desambiguadores que não são muito amigáveis para quem utilizará o objeto (como o `_` colocado nos atributos `x_` e `y_`).

Em ambos os métodos `x()` e `y()`, vemos também um qualificador `const`.

Na sequência, temos operações mais complexas, que envolvem alterar o estado atual do objeto:

```
    Vector2D &invert();
    Vector2D &multiply(double);
    Vector2D &sum(const Vector2D &);
```

Compilação em múltiplos arquivos

vector2d.h

```
#ifndef MATH_VECTOR_H
#define MATH_VECTOR_H

namespace math {

class Vector2D {
public:
    Vector2D(double x, double y);
    Vector2D() = default;
    Vector2D(const Vector2D &) = default;

    double x() const;
    double y() const;

    Vector2D &invert();
    Vector2D &multiply(double);
    Vector2D &sum(const Vector2D &);

private:
    double x_{0}, y_{0};
};

Vector2D inverted(Vector2D);
Vector2D sum(Vector2D, const Vector2D &);
Vector2D multiply(Vector2D, double);

}

#endif
```

`vector2d.cpp`

```cpp
#include "vector2d.h"


namespace math {

Vector2D::Vector2D(double x, double y) :
    x_{x},
    y_{y}
{}

double Vector2D::x() const
{
    return x_;
}

double Vector2D::y() const
{
    return y_;
}

Vector2D& Vector2D::invert()
{
    return *this = Vector2D{-x_, -y_};
}

Vector2D &Vector2D::multiply(double s)
{
    return *this = {s*x_, s*y_};
}

Vector2D &Vector2D::sum(const Vector2D &other)
{
    x_ += other.x_;
    y_ += other.y_;

    return *this;
}

Vector2D inverted(Vector2D v)
{
    v.invert();
    return v;
}

Vector2D sum(Vector2D lhs, const Vector2D &rhs)
{
    lhs.sum(rhs);

    return lhs;
}

Vector2D multiply(Vector2D lhs, double s)
{
    lhs.multiply(s);

    return lhs;
}
```

```
}
```

`main.cpp`

```cpp
#include <iostream>
#include "vector.h"


void print_vector(const math::Vector &v)
{
    std::cout << '{' << v.x() << ", " << v.y() << "}\n";
}


int main()
{
    using math::Vector;

    auto v = Vector{3, 5};

    print_vector(v);

    v.sum({7, 8});

    print_vector(v);

    auto iv = inverted(v);

    print_vector(v);
    print_vector(iv);

    v.invert();

    print_vector(v);

    v.multiply(0.5);

    print_vector(v);

    auto w = sum(v, {7, 8});

    print_vector(v);
    print_vector(w);

    auto u = multiply(v, 0.5);

    print_vector(v);
    print_vector(u);


    const auto &v2 = v;
}
```

Exercícios

Represente, através de uma classe, um aluno em uma disciplina, possuindo:

- Nome;
- Ano de entrada no curso;
- Três notas.

O objeto que representa o aluno deve ser rapaz de reportar, no mínimo:

- Há quantos anos está no curso.
- Sua média na disciplina.

Pondere sobre a existência de construtor padrão na classe e justifique brevemente em um comentário sua decisão de implementá-lo ou não.

Na sequência, faça duas funções (externas à classe) que recebam vetores de alunos, sendo elas:

- Uma que retorne quem passou na disciplina (considere média 6);
- Uma dizendo quais alunos já jubilaram, considerando:
  - Não existe entrada no 2º semestre - ou seja, todos entram no 1º semestre;
  - O período máximo para realizar o curso é de 7 anos.

Este exercício deve ser entregue como um pacote com cinco arquivos;

```
student.h
student.cpp
functions.h
functions.cpp
main.cpp
```

E deve poder ser compilado com a seguinte linha:

```
g++ -std=c++17 student.cpp functions.cpp main.cpp
```

Utilize o seguinte `main.cpp`

```
#include <iostream>
#include <string>
#include <vector>

#include "student.h"
#include "functions.h"

int main()
{
    using std::string_literals;
    using school::Student;

    // TODO: initialize a vector called `students` here

    for (auto s: school::approved(students)) {
        std::cout << s.name()
                  << " is approved by grade with average "
                  << s.average() << ".\n";
    }

    for (auto s: school::dismissed_by_time(students, 2019)) {
        std::cout << s.name()
                  << " was dismissed for being in the course for too long: "
                  << s.years_enrolled(2019) << " years.\n";
    }
}
```

Aula 4 - CMake e Streams

Nessa aula veremos uma das ferramentas mais populares para organizar projetos C++, o **CMake**.

Organização de um projeto simples com CMake

O que é o CMake

CMake é um **gerador de sistemas de build**. Um **sistema de build** é uma coleção de utilitários que organiza a compilação de um projeto. Alguns exemplos são:

- Make
- Ninja
- MSBuild (parte do Microsoft Visual Studio)

O CMake é uma ferramenta capaz de gerar projetos para diversos sistemas de build diferentes. Com isso, conseguimos não só organizar nosso projeto como suportar diversos ambientes diferentes, compiladores diferentes e sistemas operacionais diferentes.

Um

Um projeto básico com CMake

Seja um programa composto pelos seguintes arquivos:

```
$ tree grading
.
├── functions.cpp
├── functions.h
├── main.cpp
├── student.cpp
└── student.h
```

Você pode baixar estes arquivos aqui: Exemplo.

Exercícios

Aula 5 - Streams

cout e cin

fstreams

stringstreams

Evitando problemas

Exercícios

Utilizando o código desenvolvido no exercício da aula 3, adicione a funcionalidade de ler as informações dos alunos a partir da entrada padrão, ou de um arquivo.

Se você ainda não fez o exercício da aula 3, baixe essa versão.

A entrada deve estar no seguinte formato:

```
<número de entradas>
<nome>
<ano de entrada no curso>
<nota1>
<nota2>
<nota3>
<nome>
<ano de entrada no curso>
<nota1>
<nota2>
<nota3>
```

Exemplo:

```
3
João
2014
4.0
7.5
6.0
Afonso
2018
7.0
9.0
9.0
Maria
2012
6.0
9.0
10.0
```

Utilize **CMake** para compilar o seu programa, com o comando `add_executable`. Não esqueça de começar o arquivo `CMakeLists.txt` com os comandos `cmake_minimum_required` e `project`.

Este exercício deve ser entregue como um pacote com cinco arquivos;

```
CMakeLists.txt
student.h
student.cpp
functions.h
functions.cpp
main.cpp
```

E deve poder ser compilado da seguinte forma:

```
mkdir build && cd build
cmake ..
cmake --build
```

Utilize o seguinte `main.cpp`

```
#include <iostream>
#include <fstream>

#include "student.h"
#include "functions.h"

int _main(std::ostream& in)
{
    auto students = school::read_students(in);

    for (auto s: school::approved(students)) {
        std::cout << s.name()
                  << " is approved by grade with average "
                  << s.average() << ".\n";
    }

    for (auto s: school::dismissed_by_time(students, 2019)) {
        std::cout << s.name()
                  << " was dismissed for being in the course for too long: "
                  << s.years_enrolled(2019) << " years.\n";
    }

    return 0;
}

int main(int argc, char** argv)
{
    if (argc == 1) {
        return _main(std::cin);
    }

    auto file = std::ifstream{argv[1]};

    return _main(file);
}
```

Aula 6 - Herança e Polimorfismo

O mecanismo de herança e métodos virtuais

Gerência manual de memória

O std::unique_ptr

Destrutores

Exercícios

A partir do exercício da aula 5, implemente as seguintes modificações:

- Crie uma classe-base `Course` com um método virtual `average'. Esta será a interface para o cálculo de nota de mais de uma disciplina.
- Crie duas classes que herdam de `Course` implementando `average` de formas diferentes:
  - Uma chamada `DataStructures`, cuja média é a média entre três trabalhos com peso 0.8 e uma prova com peso 0.2.
  - Uma chamada `Calculus`, cuja média é a média entre três provas.
- Em lugar de cada `Student` possuir três notas para calcular sua média, ele deve ter um `std::vector<unique_ptr<Course>>` que guarda todas as suas disciplinas com suas respectivas notas, e um método `averages` que retorna um vetor de `double` com todas as suas médias.
- Altere o formato de entrada para que seja:

```
<número de entradas>
<nome>
<ano de entrada no curso>
<número de disciplinas>
<nome da primeira disciplina>
<nota1>
<nota2>
<...>
<nome da segunda disciplina>
<nota1>
<nota2>
<...>
```

Exemplo:

```
3
João
2014
1
Data Structures
4.0
7.5
6.0
10.0
Afonso
2018
2
Calculus
7.0
9.0
9.0
Data Structures
10.0
10.0
9.0
8.0
Maria
2012
2
Data Structures
9.0
9.0
10.0
10.0
Calculus
6.0
9.0
10.0

- Alterações no `main` dessa vez ficam como parte do exercício.
```

Aula 7 - Templates

Exercícios

Implemente o algoritmo **insertion sort**, descrito em pseudocódigo abaixo:

```
insertion-sort(vector A):
    for i = 1 to A.size-1:
        k = A[i]
        j = i - 1
        while j > -1 and A[j] > k:
            A[j + 1] = A[j]
            j -= 1
        A[j + 1] = k
```

Perceba que o parâmetro `A` no pseudocódigo está tipado como um `vector`. Em C++, este parâmetro precisará ser um `std::vector`. Utilize o que foi visto em aula para fazer de sua implementação um template de função que possa ordenar um `std::vector` de qualquer tipo.

Enumere em um comentário características que estão sendo assumidas sobre o tipo genérico utilizado, como pontos ou discorrendo em um parágrafo. Não é necessária uma descrição formal, apenas anote o que perceber.

Utilize CMake para compilar o seu projeto, com o seguinte `main.cpp`:

```
#include <iostream>
#include <memory>
#include <vector>

#include "insertion_sort.hpp"

namespace {

class DynInt {
public:
    DynInt(int value):
        value_{std::make_unique<int>(value)}
    {}

    auto value() const
    {
        return *value_;
    }

private:
    std::unique_ptr<int> value_;
};

std::ostream& operator<<(std::ostream& out, DynInt const& di)
{
    out << "DynInt{" << di.value() << "}";
    return out;
}

bool operator<(DynInt const& rhs, DynInt const& lhs)
{
    return rhs.value() < lhs.value();
}

}

int main()
{
    using namespace std::string_literals;

    auto print_vector = [](auto const & v)
    {
        std::cout << "{ ";
        for (auto &&i: v) {
            std::cout << i << " ";
        }
        std::cout << "}\n";
    };

    auto vi = std::vector{2, 1, 4, 3};

    algorithm::insertion_sort(vi);

    print_vector(vi);

    auto vs = std::vector{"test"s, "prototype"s, "production"s, "angel"s,
"nerv"s, "gehirn"s};
```

```
    algorithm::insertion_sort(vs);

    print_vector(vs);

    auto vdi = std::vector<DynInt>{};
    vdi.emplace_back(3);
    vdi.emplace_back(2);
    vdi.emplace_back(4);
    vdi.emplace_back(1);

    algorithm::insertion_sort(vdi);

    print_vector(vdi);
}
```