



C++ - Módulo 08

Contêineres modelados, iteradores, algoritmos

Resumo:

Este documento contém os exercícios do Módulo 08 dos módulos C++.

Versão: 7

Conteúdo

I	Introdução	2
II	Regras gerais	3
III	Regras específicas do módulo	5
IV	Exercício 00: Fácil encontrar	6
V	Exercício 01: Span	7
VI	Exercício 02: Abominação mutante	9

Capítulo I

Introdução

C++ é uma linguagem de programação de uso geral criada por Bjarne Stroustrup como uma extensão da linguagem de programação C, ou "C with Classes" (fonte: [Wikipedia](#)).

O objetivo desses módulos é apresentá-lo à **Programação Orientada a Objetos**. Este será o ponto de partida de sua jornada C++. Muitos idiomas são recomendados para aprender OOP. Decidimos escolher C++, pois é derivado de seu velho amigo C. Por se tratar de uma linguagem complexa e para manter as coisas simples, seu código estará em conformidade com o padrão C++98.

Sabemos que o C++ moderno é muito diferente em muitos aspectos. Portanto, se você deseja se tornar um desenvolvedor C++ proficiente, cabe a você ir além do 42 Common Core!

Capítulo II

Regras gerais

Compilando

- Compile seu código com `c++` e os sinalizadores `-Wall -Wextra -Werror`
- Seu código ainda deve compilar se você adicionar o sinalizador `-std=c++98`

Convenções de formatação e nomenclatura

- Os diretórios de exercícios serão nomeados desta forma: `ex00`, `ex01`, ... , `exn`
- Nomeie seus arquivos, classes, funções, funções de membro e atributos conforme exigido em As diretrizes.
- Escreva os nomes das classes no formato **UpperCamelCase**. Arquivos contendo código de classe serão sempre ser nomeado de acordo com o nome da classe. Por exemplo:
`ClassName.hpp/ClassName.h`, `ClassName.cpp` ou `ClassName.tpp`. Então, se você tiver um arquivo de cabeçalho contendo a definição de uma classe "BrickWall" que representa uma parede de tijolos, seu nome será `BrickWall.hpp`.
- A menos que especificado de outra forma, todas as mensagens de saída devem ser encerradas com uma nova linha caractere e exibido na saída padrão.
- *Adeus Norminette!* Nenhum estilo de codificação é aplicado nos módulos C++. Você pode seguir o seu favorito. Mas lembre-se de que um código que seus pares avaliadores não conseguem entender é um código que eles não podem avaliar. Faça o seu melhor para escrever um código limpo e legível.

Permitido/Proibido

Você não está mais codificando em C. Hora de C++! Portanto:

- Você tem permissão para usar quase tudo da biblioteca padrão. Portanto, em vez de se ater ao que você já sabe, seria inteligente usar o máximo possível as versões em C++ das funções C às quais você está acostumado.
- No entanto, você não pode usar nenhuma outra biblioteca externa. Isso significa que as bibliotecas C++11 (e formas derivadas) e Boost são proibidas. As seguintes funções também são proibidas: `*printf()`, `*alloc()` e `free()`. Se você usá-los, sua nota será 0 e pronto.

- Observe que, a menos que explicitamente declarado de outra forma, o namespace using <ns_name> e palavras-chave de amigos são proibidas. Caso contrário, sua nota será -42.
- **Você tem permissão para usar o STL apenas no Módulo 08 e 09.** Isso significa: sem **contêineres** (vetor/lista/mapa/e assim por diante) e sem **algoritmos** (qualquer coisa que requeira incluir o cabeçalho <algorithm>) até então. Caso contrário, sua nota será -42.

Alguns requisitos de projeto

- O vazamento de memória também ocorre em C++. Quando você aloca memória (usando o novo palavra-chave), você deve evitar **vazamentos de memória**.
- Do Módulo 02 ao Módulo 09, suas aulas devem ser elaboradas no **Ortodoxo Forma Canônica, exceto quando explicitamente declarado de outra forma**.
- Qualquer implementação de função colocada em um arquivo de cabeçalho (exceto para modelos de função) significa 0 para o exercício.
- Você deve ser capaz de usar cada um de seus cabeçalhos independentemente dos outros. Assim, eles devem incluir todas as dependências de que precisam. No entanto, você deve evitar o problema de inclusão dupla adicionando **guardas de inclusão**. Caso contrário, sua nota será 0.

Leia-me

- Você pode adicionar alguns arquivos adicionais se precisar (ou seja, para dividir seu código). Como essas atribuições não são verificadas por um programa, sinta-se à vontade para fazê-lo, desde que entregue os arquivos obrigatórios.
- Às vezes, as diretrizes de um exercício parecem curtas, mas os exemplos podem mostrar requisitos que não estão explicitamente escritos nas instruções.
- Leia cada módulo completamente antes de começar! Realmente, faça isso.
- Por Odin, por Thor! Use seu cérebro!!!



Você terá que implementar muitas classes. Isso pode parecer tedioso, a menos que você seja capaz de criar o script de seu editor de texto favorito.



Você tem uma certa liberdade para completar os exercícios. No entanto, siga as regras obrigatórias e não seja preguiçoso. Você poderia perder muita informação útil! Não hesite em ler sobre conceitos teóricos.

Capítulo III

Regras específicas do módulo

Você notará que, neste módulo, os exercícios podem ser resolvidos SEM os Containers padrão e SEM os Algoritmos padrão.


Porém, **utilizá-los é justamente o objetivo deste Módulo**. Você tem permissão para usar o STL. Sim, você pode usar os **Containers** (vetor/lista/mapa/e assim por diante) e os **Algoritmos** (definidos no cabeçalho <algoritmo>). Além disso, você deve usá-los o máximo que puder. Portanto, faça o possível para aplicá-los onde for apropriado.

Você receberá uma nota muito ruim se não o fizer, mesmo que seu código funcione conforme o esperado. Por favor, não seja preguiçoso.

Você pode definir seus modelos nos arquivos de cabeçalho como de costume. Ou, se desejar, você pode escrever suas declarações de modelo nos arquivos de cabeçalho e escrever suas implementações em arquivos .tpp. Em qualquer caso, os arquivos de cabeçalho são obrigatórios enquanto os arquivos .tpp são opcionais.

Capítulo IV

Exercício 00: Fácil encontrar

	Exercício: 00
Fácil encontrar	
Diretório de entrega: ex00/	
Arquivos a serem entregues: Makefile, main.cpp, easyfind.{h, hpp} e arquivo opcional: easyfind.tpp Funções	
proibidas: Nenhuma	

Um primeiro exercício fácil é a maneira de começar com o pé direito.

Escreva um modelo de função `easyfind` que aceite um tipo `T`. Ele recebe dois parâmetros. O primeiro é do tipo `T` e o segundo é um inteiro.

Assumindo que `T` é um contêiner **de inteiros**, esta função deve encontrar a primeira ocorrência do segundo parâmetro no primeiro parâmetro.

Se nenhuma ocorrência for encontrada, você pode lançar uma exceção ou retornar um valor de erro da sua escolha. Se precisar de alguma inspiração, analise como os contêineres padrão se comportam.


Claro, implemente e entregue seus próprios testes para garantir que tudo funcione conforme o esperado.



Você não precisa lidar com contêineres associativos.

Capítulo V

Exercício 01: Span

	Exercício: 01
Período	
Diretório de entrega: ex01/	
Arquivos a serem entregues: Makefile, main.cpp, Span.{h, hpp}, Span.cpp	
Funções proibidas: Nenhuma	

Desenvolva uma classe **Span** que possa armazenar no máximo N inteiros. N é uma variável int sem sinal e será o único parâmetro passado para o construtor.

Essa classe terá uma função de membro chamada `addNumber()` para adicionar um único número ao Span. Ele será usado para preenchê-lo. Qualquer tentativa de adicionar um novo elemento se já houver N elementos armazenados deve gerar uma exceção.

Em seguida, implemente duas funções de membro: `shortestSpan()` e `LongSpan()`

Eles descobrirão respectivamente o intervalo mais curto ou o intervalo mais longo (ou distância, se preferir) entre todos os números armazenados e o retornarão. Se não houver números armazenados, ou apenas um, nenhum span poderá ser encontrado. Assim, lance uma exceção.

Claro, você escreverá seus próprios testes e eles serão muito mais completos do que os abaixo. Teste seu Span pelo menos com um mínimo de 10.000 números. Mais seria ainda melhor.

Executando este código:

```
int principal()
{
    Span sp = Span(5);

    sp.addNumber(6);
    sp.addNumber(3);
    sp.addNumber(17);
    sp.addNumber(9);
    sp.addNumber(11);

    std::cout << sp.shortestSpan() << std::endl; std::cout <<
    sp.longestSpan() << std::endl;

    retornar 0;
}
```

Deve produzir:

```
$> ./ex01 2
```

```
14$>
```


Por último, mas não menos importante, seria maravilhoso preencher seu Span usando vários **iteradores**. Fazer milhares de chamadas para `addNumber()` é muito chato. Implemente uma função de membro para adicionar muitos números ao seu Span em uma chamada.



Se você não tem a menor ideia, estude os Containers. Algumas funções de membro usam uma série de iteradores para adicionar uma sequência de elementos para o recipiente.

Capítulo VI

Exercício 02: Abominação mutante

	Exercício: 02
abominação mutante	
Diretório de entrega: ex02/	
Arquivos a serem entregues: Makefile, main.cpp, MutantStack.{h, hpp} e arquivo opcional: MutantStack.tpp Funções proibidas: Nenhuma	

Agora, é hora de seguir em frente com coisas mais sérias. Vamos desenvolver algo estranho.

O contêiner `std::stack` é muito bom. Infelizmente, é um dos únicos contêineres STL que **NÃO** é iterável. Isso é ruim.

Mas por que aceitaríamos isso? Especialmente se pudermos tomar a liberdade de massacrar o pilha original para criar recursos ausentes.

Para reparar essa injustiça, você deve tornar o contêiner `std::stack` iterável.

Escreva uma classe **MutantStack**. Ele será **implementado em termos de** um `std::stack`. Ele oferecerá todas as suas funções de membro, além de um recurso adicional: **iteradores**.

Claro, você escreverá e entregará seus próprios testes para garantir que tudo funcione conforme o esperado.

Encontre um exemplo de teste abaixo.

```
int principal()
{
    MutantStack<int> mstack;

    mstack.push(5);
    mstack.push(17);

    std::cout << mstack.top() << std::endl;

    mstack.pop();

    std::cout << mstack.size() << std::endl;

    mstack.push(3);
    mstack.push(5);
    mstack.push(737); //[...]

    mstack.push(0);

    MutantStack<int>::iterator it = mstack.begin();
    MutantStack<int>::iterator ite = mstack.end();

    ++isso;
    --isto;
    while (it != ite) {

        std::cout << *it << std::endl; ++isso;

    }
    std::stack<int> s(mstack); retornar
    0; }
```

Se você executá-lo pela primeira vez com seu MutantStack e uma segunda vez substituindo o MutantStack por, por exemplo, um std::list, as duas saídas devem ser as mesmas. Obviamente, ao testar outro contêiner, atualize o código abaixo com as funções de membro correspondentes (push() pode se tornar push_back()).