INSTITUTO FED. DE EDUCAÇÃO, CIÊNC. E TEC. DE PERNAMBUCO
CURSO: TEC. EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: ALGORITMOS E ESTRUTURAS DE DADOS
PROFESSOR: RAMIDE DANTAS
ASSUNTO: C++: TEMPLATES, NAMESPACES, EXCEÇÕES

Prática 03

Parte 1: Preparação

Passo 1: Crie um novo projeto chamado Pratica3.

Passo 2: Crie um novo arquivo fonte nesse projeto chamado **pratica3.cpp**.

Esse arquivo deve conter a função `main()` da aplicação. Faça as modificações necessárias para usar a entrada/saída padrão de C++.

Passo 3: Compile a rode a aplicação para se certificar que o projeto está corretamente configurado.

Parte 2: Criando uma biblioteca de funções

Passo 1: Crie um arquivo chamado **funcoes.h**, que vai conter uma série de funções.

Passo 2: Em **funcoes.h**, implemente as funções com as assinaturas a seguir:

```
void trocar(int & a, int & b) { /* troca valores de a e b entre si */ }

int maximo(const int a, const int b) { /* retorna maior entre a e b */ }

int minimo(const int a, const int b) { /* retorna menor entre a e b */ }
```

Dê implementações adequadas às funções. Veja que a assinatura de `trocar()` usa a passagem por referência de C++. Nenhuma dessa funções deve exibir mensagens na tela. Na funções `maximo()` e `minimo()`, caso os valores dos parâmetros `a` e `b` sejam iguais, tanto faz qual é retornado ao final.

Passo 3: Em **pratica3.cpp**, teste as funções usando o código a seguir dentro da função `main()`:

```
int x = 5, y = 10, z = 30;

cout << "Antes: x = " << x << " y = " << y << endl;
trocar(x, y);
cout << "Depois : x = " << x << " y = " << y << endl;
cout << "Minimo entre " << x << " e " << y << ": " << minimo(x, y) << endl;
cout << "Maximo entre " << y << " e " << z << ": " << maximo(y, z) << endl;
```

Lembre-se de incluir o arquivo **funcoes.h** que criamos no passo anterior.

Passo 4: Compile e teste a aplicação, verificando se o resultado é o esperado.

Passo 5: Em **funcoes.h**, faça as funções ficarem dentro de um *namespace* chamado *funcoes*.

Ajuste **pratica3.cpp** de acordo. Veja o material de aula em caso de dúvidas.

Passo 6: Compile e teste a aplicação, verificando se o resultado é o mesmo do anterior.

Parte 3: Trabalhando com *templates* de funções

Passo 1: Em **pratica3.cpp**, na função `main()`, mude a declaração de `x`, `y` e `z`:

```
float x = 5.5, y = 10.15, z = 30.7;
```

Passo 2: Compile e teste a aplicação.

Verifique que deve ocorrer um erro na chamada de `trocar()`, uma vez que ela foi declarada para lidar com referência para `int` e está sendo chamada com `float`. Comentando a chamada de `trocar()`, o programa deve compilar e rodar normalmente, porém os resultados de `minimo()` e `maximo()` serão truncados, uma vez que as variáveis foram convertidas de `float` para `int`. Para corrigir ambos os problemas, teríamos que declarar novas versões de `trocar()`, `minimo()` e `maximo()` que lidassem com o tipo `float`. Isso traz ao menos dois problemas: ter que duplicar as funções para cada tipo novo; e ter que dar manutenção no código de todas as cópias caso um *bug* seja encontrado. A solução é o uso de *templates* de função.

Passo 3: Modifique as assinaturas das três funções em **funcoes.h** de forma que aceitem um tipo genérico `T`, usando a sintaxe a seguir:

```
template <class T>
void trocar(T & a, T & b) { ... }
```

Nesse código, `T` é um nome arbitrário para o tipo (poderia ser qualquer nome válido). Faça as alterações que forem necessárias no corpo de `trocar()` (se houver).
ATENÇÃO: Nas funções `minimo()` e `maximo()`, o tipo de retorno também é `T`.

Passo 4: Compile e teste a aplicação, verificando se o resultado é o esperado.

Nesse ponto a aplicação deve compilar e rodar normalmente. Experimente outros tipos para as variáveis `x`, `y` e `z`. O compilador gera automaticamente uma nova versão de cada função para cada novo tipo que é usado. Veja que para tipos integrais (`char`, `int`, `float`, `double`) não deve haver problemas, porém com tipos como `char *` e objetos as implementações podem gerar resultados estranhos ou nem compilar.

Passo 5: Especialize as funções `minimo()` e `maximo()` para lidar com `const char *` usando a sintaxe abaixo:

```
template <>
Tipo funcao(Tipo param1, Tipo param2, ...) { ... }
```

Tipo é o tipo concreto a ser usado na especialização (`int`, `double`, `char *`, etc.). O tipo `const char *` permite que a função trabalhe com *strings hardcoded* e não só variáveis, isto é, `minimo("stringA", "stringB")`. Use a função `strcmp()` para comparar *strings* (Veja seu funcionamento aqui). Faça o `#include <cstring>`.
ATENÇÃO: A especialização cria novas versões da função genérica (*template*) para tipos específicos. A função original continua existindo, não devendo ser modificada.

Passo 10: Faça modificações na `main()` para testar essas funções, como abaixo:

```
   cout << "Minimo de \"strA\" e \"strB\": " << minimo("strA", "strB") << endl;
   cout << "Maximo de \"strA\" e \"strB\": " << maximo("strA", "strB") << endl;
```

OBS.: Ajuste o código acima para considerar o *namespace* **funcoes**, se necessário.

Passo 11: Compile e teste o seu código.

Parte 3: Trabalhando com *templates* de classes

Passo 1: Crie um arquivo chamado **arranjo.h** e coloque o código abaixo, implementando os métodos como descrito nos comentários:

```
template <class T>
class Arranjo {

private:
	int tamanho; // tamanho do arranjo
	T * items; //items do arranjo

public:

	Arranjo(int tam) {
	  // instanciar o array de items com new (pratica 1) e setar tamanho;
	}

	virtual ~Arranjo() {
	  // destruir o array de items (prática 1);
	}

	virtual T get(int idx) {
	  // retornar um item do array a partir do indice;
	}

	virtual void set(int idx, const T & item) {
	  // set o item do array apontado pelo indice usando =
	}

	virtual void exibir();
};

template<class T>
void Arranjo<T>::exibir() {
	// exibir cada item numa linha da forma "<idx>: <item>"
}
```

Faça as mudanças necessárias em **arranjo.h** para usar a saída padrão.

Passo 2: Em **pratica3.cpp**, na função `main()`, adicione o seguinte código:

```
Arranjo<int> arr(10);
arr.set(4, 5);
arr.set(7, 15);
arr.set(8, 22);
arr.exibir();
```

Passo 3: Compile e teste a aplicação, verificando a saída gerada.

Passo 4: Na `main()`, adicione um novo arranjo com itens do tipo `float` e tamanho 5.

Adicione valores com casas decimais em várias posições do arranjo.

Passo 5: Compile e teste a aplicação.

> O código atual não deve tratar o caso de tentativa de acesso em uma posição errada. Uma forma de tratar essa situação é lançar uma exceção caso isso aconteça.

Passo 6: Faça com que as funções `set()` e `get()` de Arranjo lancem exceções em caso de acesso fora do array.

> Crie uma classe a `IndiceInvalido` e use `throw` para lançar uma instância dessa classe nos métodos `set()` e `get()` como descrito no material de aula.

Passo 7: Adapte a função `main()` para capturar a exceção.

> Use `try ... catch` como descrito no material de aula. Force o acesso a um elemento fora do tamanho do *array* para testar a exceção. Informe ao usuário em caso de exceção (isto é, exiba uma mensagem de erro na tela no `catch`).

Passo 8: Compile e teste a aplicação. Veja se a exceção está sendo lançada e capturada.

Passo 9: **(Desafio/Opcional)** Sobrecarregue o operador [] na classe Arranjo.

> Faça com que você possa usar seus objetos como *arrays* de C++ só que fazendo a verificação de índices, assim como ocorre em Java.

Parte 4: Especializando *templates* de classes

Passo 1: Crie um arquivo chamado **aluno.h**. Nele crie uma classe aluno com o seguinte código:

```
class Aluno {

private:
	string nome;
	string mat;

public:
	Aluno() {}

	Aluno(const char * nome, const char * mat) : nome(nome), mat(mat) {}

	friend class Arranjo<Aluno>;
};
```

> Veja que colocamos `Arranjo<Aluno>` como *friend* de Aluno, dessa forma temos acesso aos atributos privados `nome` e `mat` dentro de `Arranjo<Aluno>`.

Passo 2: Ainda em **aluno.h**, especialize os métodos `set()` e `exibir()` de `Arranjo<Aluno>` usando as declarações a seguir (depois da declaração de `Aluno`).

```
template<>
void Arranjo<Aluno>::set(int idx, const Aluno & aluno) {
	// verifique o índice como no set() default; lance exceção se preciso
	// atribua nome e mat individualmente para o item do array
	// isto é, pegue o aluno no array, e atribua cada campo um a um
}

template<>
void Arranjo<Aluno>::exibir() {
	// exiba cada aluno do array no formato "idx : mat = nome"
}
```

Passo 3: Em **pratica3.cpp**, na função `main()`, adicione o seguinte código ao final:

```
Arranjo<Aluno> turma(3);

turma.set(0, Aluno("Joao","1234"));
turma.set(1, Aluno("Maria","5235"));
turma.set(2, Aluno("Jose","2412"));

turma.exibir();
```

Passo 4: Compile e teste a aplicação.

> Nesse ponto a aplicação deve funcionar como esperado. Sem a especialização feita no passo 2 o compilador não vai saber como realizar a exibição na saída (`cout << aluno`), usada no método `exibir()`. Se quisermos usar os métodos não especializados, podemos sobrecarregar os operadores = e <<.

Passo 5: **(Desafio/Opcional)** Sobrecarregue o operador = **dentro** de `Aluno` usando a sintaxe:

```
Aluno & operator=(const Aluno & aluno) { ... }
```

Passo 6: **(Desafio/Opcional)** Sobrecarregue o operador << **fora** de `Aluno` (dever ser `friend` também):

```
ostream & operator<<(ostream & out, const Aluno & aluno) { ... }
```