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

Aluno (a):			
Matrícula:		**Data:**	

Prática 03

OBS: Essa prática faz uso de conhecimentos exercitados nas Práticas 1 e 2.

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 o método `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: Trabalhando com *templates* 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) { ... }

int maximo(const int a, const int b) { ... }

int minimo(const int a, const int b) { ... }
```

> Dê implementações adequadas às funções. Veja que a assinatura de `trocar()` usa passagem por referência.

Passo 3: Em **pratica3.cpp**, na função `main()`, coloque o código a seguir:

```
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 5: Compile e teste a aplicação, verificando se o resultado é o esperado.

Passo 4: 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 5: 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, por exemplo. A alternativa a isso é o uso de templates de função.

Passo 6: Modifique as assinaturas das três funções em **funcoes.h** de forma que aceitem um tipo genérico, 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). Faça as alterações necessárias no corpo de `trocar()`. Veja que nas funções `minimo()` e `maximo()`, o tipo de retorno deve ser o mesmo dos parâmetros.

Passo 5: 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. Nesses casos, é possível dar uma implementação especializada para um tipo específico usando a sintaxe a seguir:

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

`Tipo` é o tipo concreto a ser usado na especialização (`int`, `double`, `char *`, etc.).

Passo 6: Especialize as funções `minimo()` e `maximo()` para lidar com `char *` usando a sintaxe acima.

Use a função `strcmp()`; será preciso fazer `#include <cstring>`. Faça modificações no `main()` para testar essas funções. Compile e teste em seguida.

Passo 7: Coloque as funções dentro de um *namespace* chamado *funcoes*.

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

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: Adicione um novo arranjo, dessa vez com itens do tipo `float` com 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 de exceção e use `throw` como descrito material de aula.

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

Use `try ... catch`. Force o acesso a elemento fora do tamanho do array para testar a exceção. Informe ao usuário em caso de exceção.

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) {
	// atribua nome e mat individualmente para o item do array
}

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:

```
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 1)** Sobrecarregue o operador = **dentro** de `Aluno` usando:

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

Passo 6: **(Desafio 2)** Sobrecarregue o operador << **fora** de `Aluno` (dever ser `friend` também):

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