Aula 10:

Templates
Programação Genérica

ECOP13A - Programação Orientada a Objetos
Prof. André Bernardi
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá

Templates

Um dos recursos de reutilização de software mais poderosos do C++.

Templates de **classes** e de Templates de **funções** possibilitam a especificação, através de um único trecho de código, uma gama inteira de **classes ou funções relacionadas entre si**, que diferem apenas por detalhes de tipo.

Essa técnica é chamada de Programação **Genérica**.

Templates podem ser também chamados, em alguns livros, de "**gabaritos**". Depende da tradução!

Templates

A partir desta técnica, podemos, por exemplo, escrever um único ***template*** para uma função de ordenação de vetores, que ordene vetores de quaisquer tipos de dados: vetores de int, de float, de strings, etc.

Além disso, podemos construir também uma classe Pilha e, através de templates, fazer com que a pilha seja genérica, ou seja, o C++ irá gerar especializações de template de classes separadas, tais como uma pilha de int, uma de float, outra de string, e assim por diante... Tudo a partir de **uma única especificação**.

Vamos iniciar pelos **templates de função**.

Quando uma mesma operação é realizada de forma idêntica para vários tipos de dados, ao invés de sobrecarregá-la uma vez para cada tipo, podemos simplesmente definir uma **função genérica** (ou "**template**"), que aceite qualquer tipo de dados para tratamento.

> É baseado nos tipos de argumentos passados para a função (explicitamente ou através de chamada de outra função) que o compilador **gera** funções separadas no código-fonte. Chamamos essas funções geradas pelo compilador de **especializações de template de função**.

Vejamos alguns exemplos de sintaxe:

```
// declara que função será template
template <typename T>
void print_vector(T *v, int sz)
{
    for(int i = 0; i < sz; i++)
        cout << v[i] << " ";
}

// A, assim como T na função anterior, representa qualquer
// tipo de dados que se queira utilizar
template <typename A>
void sort_vector(A vetor[], int tam)
{
    A aux;
    for(int i = 0; i < tam; i++)
        for(int j = 0; j < tam - 1 - i; j++)
            if (vetor[j+1] < vetor[j])
            {
                aux = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = aux;
            }
}
```

Definições de templates de função começam sempre com a palavra reservada **template**, seguida por uma lista de **parâmetros de template**.

Os parâmetros (pode haver mais de um) de template são incluídos entre < >, separados por vírgulas e cada um deve ser precedido da palavra **typename** ou **class**.

Estes parâmetros podem ser utilizados para especificar:

- Tipos de argumentos da função;
- Tipos de retorno;
- Tipos de variáveis locais;

```cpp
int main()
{
    // Chamando as funções print_vector e sort_vector <float>
    float vf[10];
    for(int i = 0; i < 10; i++)
        vf[i] = (float)(rand() % 100)/100.0;

    cout << "Vetor de float desordenado: ";
    print_vector(vf, 10);
    cout << endl;
    sort_vector(vf, 10);
    cout << "Vetor de float ordenado: ";
    print_vector(vf, 10);
    cout << endl;

    // Chamando as funções print_vector e sort_vector <int>
    int vi[10];
    for(int i = 0; i < 10; i++)
        vi[i] = rand() % 100;

    cout << "Vetor de int desordenado: ";
    print_vector(vi, 10);
    cout << endl;
    sort_vector(vi, 10);
    cout << "Vetor de int ordenado: ";
    print_vector(vi, 10);
    cout << endl;
}
```

```
Vetor de float desordenado: 0.41 0.67 0.34 0 0.69 0.24 0.78 0.58 0.62 0.64
Vetor de float ordenado: 0 0.24 0.34 0.41 0.58 0.62 0.64 0.67 0.69 0.78
Vetor de int desordenado: 5 45 81 27 61 91 95 42 27 36
Vetor de int ordenado: 5 27 27 36 42 45 61 81 91 95
```

As definições das funções se parecem com a definição de uma função comum, mas podem utilizar os tipos definidos na lista de parâmetros de template.

Na main, pouca coisa muda. No entanto, repare que chamamos uma mesma função com parâmetros de tipos diferentes. E só implementamos uma de cada!

Resumindo...

A definição da função começa com a palavra **template**, seguida da lista de parâmetros de template.

```
// função retorna um T e recebe um parâmetro T e um A
    template< typename T, typename A>
    T example_function(T par1, A par2) {...}
```

A utilização não tem nada diferente do habitual, assim como a própria definição da função, que, no exemplo, contará com dois tipos “genéricos” diferentes: A e T.

Importante:

Caso o template seja chamado com um tipo definido pelo usuário (uma classe), e este template utilize operadores (por exemplo ==, <, >, <<, etc.) com objetos do tipo desta classe, é necessário que os operadores estejam sobrecarregados para este tipo. Caso não estejam, será gerado um erro de compilação.

Exemplo: No caso da impressão, precisaríamos sobrecarregar o operador **<<**. Para a ordenação, o operador **<** de comparação.

Além disso, lembre-se de que, apesar de escrever apenas uma vez, será gerada, no tempo de compilação, **uma versão para cada tipo utilizado**. Cada especialização do template será independente e ocupará seu próprio espaço na memória.

Isso é um problema?
Não! Você teria que escrever o código de qualquer maneira.

Templates de Classes

É possível compreender o funcionamento de uma estrutura de dados do tipo “**pilha**” (LIFO, ou “último a entrar, primeiro a sair”) sem especificar o **tipo** de itens que estão sendo manipulados por ela. No entanto, ao instanciar uma pilha, um tipo de dados deve ser especificado.

Isso gera uma grande oportunidade de **reutilização de software**: desejamos descrever o funcionamento de uma pilha através de uma **classe genérica** e instanciar objetos a partir dela que sejam versões específicas seu próprio tipo.

Isso pode ser feito utilizando **templates**!

Templates de Classes

Templates de classes são chamados, também, de **tipos parametrizados**, pois exigem um ou mais parâmetros de tipo para especificar como personalizar um template de uma “classe genérica” para formar uma especialização de template de classe específica.

Com este recurso poderíamos, portanto, criar uma pilha genérica que, através de uma **sintaxe extremamente concisa**, poderá ser especializada em uma pilha de inteiros, pilha de floats, pilha de Pessoas (!), pilha de Tarefas, etc.

Vamos ver o exemplo:

Templates de Classes

Para começar, a declaração da classe template será muito parecida com a de uma classe normal, exceto por um detalhe muito parecido com os templates de função. É preciso incluir um cabeçalho do tipo:

template < typename T >

Indicando que é um template de classe e listando os parâmetros de template. Ficaria da seguinte maneira:

```
template <typename T>
class Pilha { //...
```

Importante ressaltar que o nome do parâmetro não precisa ser **T**, ainda que isso seja bastante comum. **T** precisa ser um identificador válido, pelo qual este tipo "genérico" chamado **T** será referenciado durante toda a classe.

Templates de Classes: Definição da classe

```
template <typename T>
class Pilha { //...
    private:
        int tamanho;
        int topo;
        T *pilha;
    public:
        Pilha(int = 10);
        ~Pilha() {
            delete[] pilha;
        }

        // insere no topo, retira do topo
        bool push(T&);
        bool pop(T&);

        // Não precisam de template<> pois estão
        // dentro da definição do template da classe
        // está vazia?
        bool is_empty() {
            if(topo == -1) return true;
            return false;
        }

        // está cheia?
        bool is_full() {
            if(topo == tamanho-1) return true;
            return false;
        }
};
```

No exemplo, temos a definição de uma classe Pilha, que contém um **vetor de objetos de um tipo genérico T**, que pode ser qualquer tipo primitivo ou tipo definido pelo usuário.

Além disso, temos **outras variáveis de tipo fixo** (int) que não precisam ser genéricas e guardam o tamanho da pilha e o índice do elemento no topo.

A pilha tem um tamanho *default* de 10 e suas funções membro, quando implementadas dentro da definição da classe, não diferem em nada de classes não-template.

Templates de Classes: Definição de métodos

```
// as funções definidas fora da classe precisam
// também ser funções template, com escopo Pilha<T>
template <typename T>
Pilha<T>::Pilha(int sz)
    : tamanho{sz > 0 ? sz : 10}, topo{-1}
{
    pilha = new T[tamanho];
}

template <typename T>
bool Pilha<T>::push(T &valor) {
    if(!is_full())
    {
        topo++;
        pilha[topo] = valor;
        return true;
    }
    return false;
}

template <typename T>
bool Pilha<T>::pop(T &valor) {
    if(!is_empty())
    {
        valor = pilha[topo];
        topo--;
        return true;
    }
    return false;
}
```

Cada implementação de uma função membro feita **fora da definição** do template da classe deve ser tratada como **templates de função** e precisam ser iniciadas com o cabeçalho:

template < typename T >

O construtor aloca memória para a pilha em si, enquanto as funções de inserção e remoção realizam as operações básicas de acordo com as regras da pilha.

Repare que o **escopo** de cada função membro será a especialização do template **Pilha<T>.**

Templates de Classes: Programa de teste

```
#include "pilha.h"
#include <iostream>

using namespace std;

int main()
{
    // criando uma pilha de inteiros com 50 posições (máx.)
    Pilha<int> p{50};
    int v = 0;

    cout << "inserindo itens: (-1 para terminar)\n";
    while(v != -1)
    {
        cin >> v;
        p.push(v);
    }

    cout << "Retirando itens... ";
    while(!(p.is_empty()))
    {
        p.pop(v);
        cout << v << " ";
    }
}
```

Para a pilha declarada, o compilador irá gerar sua especialização, substituindo o **T** do parâmetro de template pelo tipo fundamental **int**, colocado entre os símbolos < >.

A utilização da classe é feita da mesma maneira que para classes comuns.

```
inserindo itens: (-1 para terminar)
10
20
51
68
74
95
2
-1
Retirando itens... -1 2 95 74 68 51 20 10
```

Duas observações:

1) A maioria dos compiladores em C++ requer que a **definição completa** de um template esteja no arquivo de código-fonte do cliente que usa o template. Por este motivo, os **templates são definidos e implementados no próprio arquivo de cabeçalho**, que então são incluídos (#include) no código-fonte do cliente. Ou seja, **implemente** funções membro de uma classe template no próprio arquivo **.h**.

2) Note a **distinção entre templates** e **especializações do template**: templates são como réguas com as quais traçamos formas. As especializações de template são como os traçados separados, todos têm a mesma forma mas podem, por exemplo, ter cores diferentes.

Parâmetros não tipo e tipos default para templates de classe

O template de classe Pilha utiliza apenas um parâmetro de **tipo** no cabeçalho do template. Também é possível utilizar parâmetros **não tipo**, que são tratados como ***const***. Um exemplo de cabeçalho:

```
template < typename T, int elementos >
```

A partir deste template, uma possível declaração seria:

```
Pilha < double, 100 >
```

Que poderia ser utilizada, por exemplo, para criar (em tempo de compilação) uma classe especializada de pilha de double com 100 elementos.

```
T pilha[ elementos ];
```

Parâmetros não tipo e tipos default para templates de classe

Além disso, um parâmetro de tipo pode especificar um **tipo default**. Por exemplo:

```
template < typename T = int >
```

Desta maneira, podemos simplificar a declaração de uma pilha caso queiramos que ela tenha o tipo default. Uma pilha de inteiros seria declarada da seguinte maneira:

```
Pilha<> notas;
```

Os parâmetros de tipo default precisam ser os parâmetros mais a direita da lista (como em valores default de funções).

Parâmetros não tipo e tipos default para templates de classe

Dicas:

- Quando for apropriado, especifique o tamanho de uma classe container (como um Array ou uma Pilha, que sirva para armazenar dados) no tempo de compilação (através, por exemplo, de um parâmetro de template não tipo). Isso elimina o *overhead* do tempo de execução do uso de ***new*** para alocar espaço dinamicamente.

- Fazer isso evita o erro possivelmente fatal no tempo de execução se o ***new*** for incapaz de obter a memória necessária (bad_alloc).

Templates e Informações sobre Herança

Templates e **herança** se relacionam de diversas maneiras:

- Um **template de classe** pode ser derivado de uma **especialização de template de classe**;
- Um **template de classe** pode ser derivado de uma **classe não template**;
- Uma **especialização de template de classe** pode ser derivada de uma **especialização de template de classe**;
- Uma **classe não template** pode ser derivada de uma **especialização de template de classe**.

```
class PilhaDouble : public Pilha<double>
{
    private:
        double soma_itens;
    public:
        PilhaDouble() {}
        ~PilhaDouble() {}
};
```

Templates e Membros static

Em uma classe não template, uma **cópia** de cada membro ***static*** é compartilhada entre todos os objetos daquela classe, e o membro ***static*** deve ser inicializado no escopo de *namespace* global.

Com templates, **cada especialização de template de classe instanciada** a partir do template tem **sua própria cópia** de cada membro de dados ***static***: todos os objetos gerados a partir de uma mesma especialização, compartilham seus membros ***static***. Da mesma maneira que acontece em classes não-template, os membros *static* precisam ser inicializados em escopo de *namespace* global (não pode ser *in-class*).

Templates e Friends

Funções e classes inteiras podem ser declaradas como *friends* de classes não template.

Com **templates de classe**, os tipos óbvios de *friends* possíveis podem ser declarados normalmente. A relação de *friend* pode ser estabelecida entre um *template de classe* e:

- Uma função global;
- Uma função-membro de outra classe (possivelmente uma especialização de template de classe);
- Até uma classe inteira (possivelmente até uma especialização de template de classe)

Referências

- https://cplusplus.com/reference/

- Notas de aula da disciplina Programação Orientada a Objetos, Prof. André Bernardi, Prof. João Paulo Reus Rodrigues Leite.