

Para onde vamos ...

- Agora veremos ...
 - V.1. Exceções
 - V.2. Classes parametrizadas: *templates*
- Referência básica
 - *Thinking in C++, Vol. 1*
 - Parte 1: Capítulo 16
 - *Thinking in C++, Vol. 2*
 - Parte 2: Capítulo 7



V.1.1. O que são exceções

- Uma função pode não ser capaz de fornecer os serviços requisitados por um cliente
 - As causas desta falha podem ser de várias naturezas, tais como:
 - Dados incorretos sendo passados para ela
 - Subscritos de vetor fora do intervalo, parâmetros de função inválidos
 - Condições de ambientes anormais
 - Não conseguir acessar um recurso do hardware, alocação de memória mal sucedida
 - Operações incorretas
 - *Overflow* aritmético, divisão por zero

V.1.1. O que são exceções

- Seja qual for a causa da falha ...
 - O programa cliente deve ser capaz de:
 - Detectar que houve uma situação anormal de funcionamento da função
 - Gerenciar esta anormalidade de maneira apropriada
 - As funções devem ter maneiras de comunicar aos programas clientes que situações excepcionais ocorreram em seu funcionamento
 - Isso torna os programas mais robustos e tolerantes a falhas

V.1.1. O que são exceções

- Exemplo:

```
class Vetor {
    double *ar;
    int size;
public:
    Vetor operator +(const Vetor& v) const;
    // etc...
};

Vetor Vetor::operator+(const Vetor& v) const{
    if (this->size != v.size ) {
        // OOPS.. AQUI OCORREU UM ERRO! COMO TRATÁ-LO ?????
        // Abortar? Mensagem de erro? Tentar corrigir o tamanho dos vetores?
        // Quem sabe o que fazer é o cliente, não eu !!!
    }
    else {
        Vetor result(v.size());
        for (int i=0; i<v.size(); ++i) result.ar[i]=ar[i]+v.ar[i];
        return result;
    }
}
```

V.1.1. O que são exceções

- Uma **exceção** é um evento que ocorre durante a execução de um programa, rompendo o fluxo normal das instruções
- O autor de uma biblioteca pode e deve prever erros que possam ocorrer durante sua execução ...
 - ... mas geralmente ele **não tem idéia de como esses erros devem ser tratados**
 - Quem sabe como tratá-los são os “clientes”!

V.1.1. O que são exceções

- O sistema de tratamento de exceções em C++
 - Fornece um mecanismo para “pular fora” da sequência normal de execução de uma função
 - Transfere a execução para um outro contexto
 - Assim, se não se tem informação suficiente no contexto atual para decidir o que fazer para tratar um erro...
 - É possível passar a informação sobre o erro para um contexto externo → o contexto do “**cliente**” da função

V.1.2. O tratamento de exceções

- Como C++ trata exceções
 - Este mecanismo é implementado por meio de três cláusulas
 - ***Throw***
 - Lança um objeto representando o erro, transferindo o controle do programa para o gerenciador de exceções
 - ***Try***
 - Indica que se quer capturar exceções que foram lançadas com um *throw*
 - ***Catch***
 - Manifesta o desejo de tratar uma exceção de um determinado tipo

V.1.2. O tratamento de exceções

■ **Throw**

- Faz com que uma cópia do objeto que você está lançando seja criada e lançada
 - Pode-se lançar qualquer tipo de objeto
- Sintaxe:

```
throw objeto_a_ser_lançado;
```

- Exemplo:

```
class Error { Error(string &s); /* ... */ };  
// Error é uma classe que contém um construtor  
// que recebe uma string como parâmetro  
throw Error ("Mensagem");
```


V.1.2. O tratamento de exceções

■ **Try**

- Indica que se quer capturar exceções que foram lançadas com um *throw*
- Sintaxe

```
try {  
    // Código que pode gerar exceções  
    // ou chamada a funções que podem lançar exceções  
}
```

■ Nota:

- No tratamento de exceções, as funções que possam lançar exceções devem ser inseridas num bloco **try** e, após este bloco, o tratamento das exceções é realizado

V.1.2. O tratamento de exceções

■ **Catch**

- Uma exceção lançada deve parar em algum lugar
 - Este lugar é chamado gerenciador de exceções (*exception handler*)
 - Você precisa de um gerenciador para cada tipo de exceção que deseja capturar
- Gerenciadores de exceções seguem imediatamente o bloco *try* e são representados pela palavra chave *catch*
 - **catch**: manifesta o desejo de tratar uma exceção de um determinado tipo

V.1.2. O tratamento de exceções

- Exemplo com a sintaxe completa:

```
try {  
    // Código que pode gerar exceções  
} catch(tipo_erro1& erro1) {  
    //trata exceções do tipo_erro1  
} catch(tipo_erro2& erro2) {  
    // trata exceções do tipo_erro2  
}  
  
    //...  
catch(tipo_erroN& erroN) {  
    // trata exceções do tipo_erroN  
}
```

V.1.2. O tratamento de exceções

- Exemplo 1:

Divisao por Zero

```
#include <iostream>
#include <string>
using namespace std;
class Erro {
    string tipo_erro;
public:
    Erro(const string &tp) : tipo_erro(tp) {};
    void out() {cout<<tipo_erro;};
};
int h (int a,int b){
    if (b == 0)
        throw Erro ("Divisao por Zero");
    return a/b;
}
int g (int i) { // ....
    int j = i-3;
    return h (i,j);
};
int f (int p) { return g(p); };

int main() {
    try { //Exceções acontecidas aqui são tratados nos catch
        int k = 3;
        f(k);
    }
    catch (Erro &e) {
        e.out();
        // trata o erro devidamente
    }
}
```

V.1.2. O tratamento de exceções

■ Exemplo 2:

```
Enter two integers (EOF to end): 5
2
The quotient is: 2.5

Enter two integers (EOF to end): 7
4
The quotient is: 1.75

Enter two integers (EOF to end): 8
0
Exception occurred: Divisao por zero

Enter two integers (EOF to end): 1
2
The quotient is: 0.5

Enter two integers (EOF to end): EOF
```

```
#include<iostream>
using namespace std;
class DivideByZeroException {
    const char *message;
public:
    DivideByZeroException(): message("Divisao por zero") {}
    const char *what() const {return message; }
};

double quotient (int num, int den) {
    if (den == 0)
        throw DivideByZeroException();
    return (double) num/den;
}

int main() {
    int n1, n2;
    double result;
    cout<<"Enter two integers (EOF to end): ";
    while (cin >>n1 >>n2) {
        try { // código que pode gerar exceções
            result = quotient( n1, n2 );
            cout<<"The quotient is: "<< result<< endl;
        }
        catch(DivideByZeroException& ex){
            cout << "Exception occurred: "<< ex.what() << '\n';
        }
        cout<<"\nEnter two integers (EOF to end): ";
    }
    return 0;
}
```

V.1.2. O tratamento de exceções

- Funcionamento

- **throw** lança um objeto do tipo especificado

- Transfere o controle do programa para o gerenciador de exceções em alguma função que, direta ou indiretamente, chamou a função que lançou a exceção
 - Para fazer isto, a implementação vai “desempilhando” as diversas funções chamadas para se chegar na função que gerou a exceção
 - Objetivo: chegar em um bloco **try** que contenha o **catch** correspondente ao tipo da exceção lançada (nos exemplos, **Erro** e **DivideByZeroException**)
 - Se um objeto for lançado por um **throw** e não houver nenhum **catch** correspondente ao seu tipo ...
 - Todos os gerenciadores do bloco **try** serão ignorados e a exceção lançada continua o seu percurso de “desempilhamento”

V.1.2. O tratamento de exceções

■ Funcionamento (cont.)

- Neste processo de “desempilhamento” as diversas variáveis (e objetos) locais das funções chamadas vão tendo seu contexto encerrado
 - Seus destrutores são chamados e elas também são retiradas da pilha
- Se chegar ao **main** e nenhum **catch** capturar a exceção, a função ***std::terminate()*** será chamada
 - Por *default*, ***terminate()*** chama ***abort()*** e o programa é encerrado

Exercícios

1. Crie uma classe chamada **Teste** que possua uma função que aloca a quantidade de inteiros recebida como parâmetro. Essa quantidade deverá ser menor ou igual a 10. Caso esse valor seja maior que 10, uma exceção deverá ser lançada. Insira também uma função para desalocar a memória alocada. Crie na função **main()** um bloco **try** e uma cláusula **catch** que chama a função para desalocar a memória alocada no bloco **try**. Mostre a recuperação da exceção e também que a memória é desalocada.

V.2. Classes parametrizadas: *templates*

- Stroustrup inicialmente chamou os *templates* de tipos parametrizados
 - Uma classe ou uma função *template* serve como uma “receita” para geração automática de classes e funções baseada na escolha de um tipo pelo usuário
- As classes e algoritmos presentes na biblioteca padrão (*vector<T>*, *list<T>*, *sort*, *find*, etc) são exemplos de classes e funções *template*

V.2. Classes parametrizadas: *templates*

- Exemplo - Uma pilha de inteiros simples



```
#include <iostream>
using namespace std;
class IntStack{
    int *stack ;    // ponteiro para a pilha
    int top, ssize; // topo e tamanho da pilha
public: // construtor e destrutor
    IntStack(int size):top(0),ssize(size){stack = new int[size];};
    ~IntStack() { delete[] stack; }
    void push(int i) { // função para empilhar
        if(top >= ssize){
            cout<<"Too many push()es";
            return;
        }
        stack[top++] = i;
    }
    int pop() { // função para desempilhar
        if (top <= 0){
            cout<<"Too many pop()s";
            return(0);
        }
        return stack[--top];
    }
};

int main() {
    IntStack is(5);
    for(int i = 0; i < 5; i++)
        is.push(i);
    for(int k = 0; k < 5; k++)
        cout << is.pop() << endl;
}
```

V.2. Classes parametrizadas: *templates*

- E se quisermos trabalhar com uma pilha de objetos de qualquer tipo?
 - Podemos copiar o código da pilha de inteiros e realizar os ajustes necessários para o novo tipo
 - Ex: Pilha de Avião
 - Desvantagens:
 - Cansativo
 - Confuso
 - Propenso a erros
 - Deselegante



V.2. Classes parametrizadas: *templates*

- Solução: utilizar *templates*
 - Os *Templates* C++ permitem a execução de um molde genérico *Stack<T>* com parâmetro T de qualquer tipo, podendo este tipo ser primitivo da linguagem ou definido pelo próprio usuário
 - Os *Templates* C++ fornecem uma maneira de reuso do código fonte, ao contrário da herança e composição, que fornecem uma maneira de reuso do código objeto
 - *Template* é a palavra chave para o compilador identificar que a classe manipulará um tipo não especificado
 - Pode ser implementado na forma de ***classes template*** ou de ***funções template***

V.2. Classes parametrizadas: *templates*

- Os *templates*, como as macros, permitem a reutilização de código
- Os *templates* de função eliminam os efeitos colaterais das macros e permitem ao compilador realizar a verificação de tipos

- **Efeito colateral 1:**

```
#define F (x) (x + 1) // note espaço ente F e (x)  
F(1) ⇒ (x) (x + 1) (1)
```

- **Efeito colateral 2:**

```
#define CUBO(x) x*x*x  
CUBO(a+b) ⇒ a+b*a+b*a+b = a + 2.ba + b
```

V.2. Classes parametrizadas: *templates*

- Função *template*: sintaxe de definição

```
template <class Tipo1, class Tipo2,...>  
protótipo_função_template()  
{ /* código da função template */ }
```

Lista de parâmetros
de *template*

- Exemplo:

```
template <class T>  
T soma (const T &p1, const T &p2) {  
    return(p1+p2);  
}
```

- Uso da função *template*:

```
int i=1 , j=2;  
double f = 3., g = 4.;  
Matriz A(20,20), B(20,20);  
int k = soma(i,j); //gera int soma(const int& , const int&)  
double h = soma(f,g); //gera double soma(const double& , const double&)  
Matriz C = soma(A,B); //gera Matriz soma(const Matriz& , const Matriz&)  
// se existir Matriz::operator+
```

V.2. Classes parametrizadas: *templates*

- Função *template*: definição
 - O código anterior mostra claramente uma característica das funções *template*: elas só podem ser instanciadas (isto é, ter seu tipo genérico T substituído por um “tipo de verdade”) caso o “tipo de verdade” satisfaça alguns pré-requisitos
 - No caso específico da função soma, quais seriam os pré-requisitos?

```
template <class T>
T soma (const T &p1, const T &p2){
    return(p1+p2);
}
```

- 1) O tipo T tem que ter implementado o operador+
- 2) Deve possuir também um construtor de cópia (criado pelo programador ou sintetizado pelo compilador) para permitir o retorno por cópia

V.2. Classes parametrizadas: *templates*

- Função *template*: definição
 - Quando uma função *template* é chamada, os tipos dos argumentos da função determinam qual versão do *template* será usada
 - Os parâmetros do *template* são deduzidos a partir dos argumentos da função
 - Cada vez que o compilador encontrar uma chamada de função *template* ainda não realizada para aqueles tipos, ele definirá um novo código
 - As várias funções terão endereços e códigos diferentes
 - Este processo é chamado de **instanciação da função *template***

```
int k = soma(i,j);  
double h = soma(f,g);  
Matriz C = soma(A,B);
```


V.2. Classes parametrizadas: *templates*

- Funções *template*: exemplo

```
template <class T>
T max (T t1, T t2) { // obtém o maior de 2 objetos de mesmo tipo
    return (t1 > t2 ? t1 : t2);
}

int main(){
    int i = 10, j = 20, k;
    k = max(i,j); // instancia max<int>
    cout<<"k="<<k<<endl;
    char a='a', d='d', x;
    x = max(a,d); // instancia max<char>
    cout<<"x="<<x<<endl;
    double pi = 3.1415, e = 2.72, q;
    q = max(pi,e); // instancia max<double>
    cout<<"q="<<q<<endl;
    return 0;
}
```

```
k=20
x=d
q=3.1415
```

V.2. Classes parametrizadas: *templates*

- Funções *template*: exemplo

```
#include<iostream>
using namespace std;

template <class T>
void printArray(const T array[], int count) {
    for (int i = 0; i < count; i++)
        cout<<array[i]<<" ";
    cout<<endl;
}

int main() {
    int a[5] = {1,2,3,4,5};
    double b[7] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
    char c[6] = "HELLO";

    cout<<"Array a possui:"<<endl;
    printArray(a,5);

    cout<<"Array b possui:"<<endl;
    printArray(b,7);

    cout<<"Array c possui:"<<endl;
    printArray(c,6);
}
```

```
Array a possui:
1 2 3 4 5
Array b possui:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c possui:
H E L L O
```

V.2. Classes parametrizadas: *templates*

- *Template*: argumentos explícitos
 - Em algumas situações não é possível para o compilador deduzir qual o tipo de argumentos do *template*

```
template <class T>
T Max (T t1, T t2) {
    return (t1 > t2 ? t1 : t2);
}
```

```
unsigned int ui = 10;
int ia = 1;
max(ui, ia); // Erro! deve instanciar max<unsigned int> ou max<int> ?
```

- Solução: indicar explicitamente qual instância de **max** se quer:

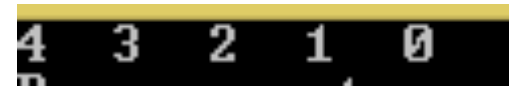
```
max<unsigned int> (ui, ia);
```

V.2. Classes parametrizadas: *templates*

■ Classes *Template*: Pilha

```
class IntStack{
    int *stack; // ponteiro para a pilha
    int top, ssize; // topo e tamanho da pilha
public: // construtor e destrutor
    IntStack(int size):top(0),ssize(size){stack = new int[size];};
    ~IntStack() { delete[] stack; }
    void push(int i) { // função para empilhar
        if(top >= ssize){
            cout<<"Too many push()es";
            return;
        }
        stack[top++] = i;
    }
    int pop() { // função para desempilhar
        if (top <= 0){
            cout<<"Too many pop()s";
            return(0);
        }
        return stack[--top];
    }
};

int main() {
    IntStack is(5);
    for(int i = 0; i < 5; i++)
        is.push(i);
    for(int k = 0; k < 5; k++)
        cout << is.pop() << endl;
}
```



A diagram representing a stack. It consists of a horizontal row of five boxes. From left to right, the boxes contain the numbers 4, 3, 2, 1, and 0. The box containing '4' is highlighted with a yellow background, indicating it is the top element of the stack.

- É possível entender o conceito de pilha independente do tipo de itens que são colocados na pilha
 - inserimos itens na parte superior
 - recuperamos na ordem último a entrar, primeiro a sair

V.2. Classes parametrizadas: *templates*

■ Classes *Template*: Pilha

```
template <class T>
class Stack{
    T *stack ;
    int top, ssize;
public:
    Stack(int size) : top(0), ssize(size) {stack = new T[size];};
    ~Stack() { delete[] stack; }
    void push(T i) {
        if(top >= ssize) {
            cout << "Too many push()es";
            return;
        }
        stack[top++] = i;
    }
    T pop() { // função para desempilhar
        if (top <= 0) {
            cout << "Too many pop()s";
            return(0);
        }
        return stack[--top]; // retorna elemento que estava no topo
    }
};
```

```
int main(){
    Stack<int> s(5);
    Stack<float> f(7);
    for (int i=0; i < 5; i++)
        s.push(i);
    for (int i = 0; i < 5; i++)
        cout<<s.pop()<<" ";
    cout<<endl;
    for (int i=0; i < 7; i++)
        f.push(i);
    for (int i = 0; i < 7; i++)
        cout<<f.pop()<<" ";
}
```

4	3	2	1	0			
6	5	4	3	2	1	0	

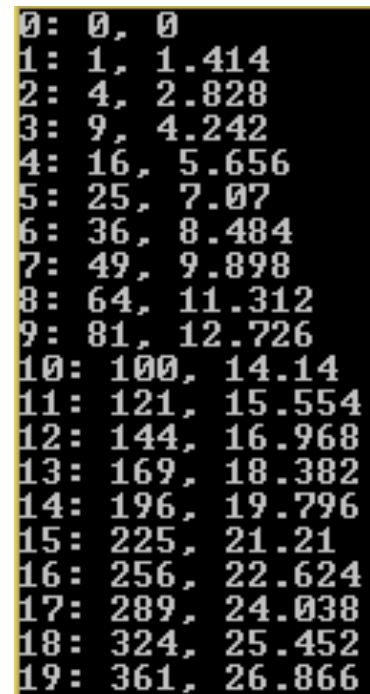
V.2. Classes parametrizadas: *templates*

■ Classes *Template*: Array

```
#include<iostream>
using namespace std;

template <class T>
class Array{
private:
    T* A;
    int size;
public:
    Array(int tam): size(tam) {A = new T[tam];}
    T& operator[](int index) {
        // insere e extrai elementos
        return A[index];
    }
};

int main(){
    Array<int> ia(20);
    Array<float> fa(20);
    for(int i=0; i<20; i++){
        ia[i]=i*i;
        fa[i]=i*1.414;
    }
    for(int j=0; j<20; j++)
        cout<<j<<": " << ia[j]<< ", " << fa[j] << endl;
}
```



0:	0,	0
1:	1,	1.414
2:	4,	2.828
3:	9,	4.242
4:	16,	5.656
5:	25,	7.07
6:	36,	8.484
7:	49,	9.898
8:	64,	11.312
9:	81,	12.726
10:	100,	14.14
11:	121,	15.554
12:	144,	16.968
13:	169,	18.382
14:	196,	19.796
15:	225,	21.21
16:	256,	22.624
17:	289,	24.038
18:	324,	25.452
19:	361,	26.866

Nota: A função `operator[]` será *inline*, pois está declarada e definida dentro do *template*

V.2. Classes parametrizadas: *templates*

- Classes *Template*: funções não *inline*

Se não for de interesse gerar função *inline*, o compilador precisa ver a declaração *template* antes da definição da função membro

```
template <class T>
class Array {
    T *A;
    int size;
public:
    Array(int tam);
    T &operator[] (int index);
};
// Funções não inline
template <class T>
Array<T>:: Array(int tam):size(tam ) {
    A = new T[tam];
}
template <class T>
T &Array<T>:: operator[] (int index) {
    return A[index];
}
```

V.2. Classes parametrizadas: *templates*

- Classes *Template*: parâmetros *default*
 - Um parâmetro de tipo pode especificar um valor *default*
 - Os parâmetros *default* devem ser os parâmetros mais à direita em uma lista de parâmetros de *template*

```
#include<iostream>
using namespace std;

template <class T = int, class X = std::string>
class par {
    T _first;
    X _second;
public:
    par(T first, X second): _first(first), _second(second) {};
    void first(T f) { _first = f; }
    T first() const { return _first; }
    void second(X s) { _second = s; }
    X second() const {return _second;}
};

int main() {
    par<double, int> p1(3.,4); // par<double, int>
    par<double> p2(3.1415, "pi"); // par<double, std::string>
    par<> p3(2, "dois"); // par<int, std::string>
    cout << p3.first() << " " << p3.second();
}
```

2 dois

V.2. Classes parametrizadas: *templates*

- Classes *Template*: parâmetros *default*
 - É possível utilizar parâmetros não-tipo que podem ser argumentos-padrão e são tratados como **consts**
 - Exemplo:

```
template<class T, int elements>
```

para instanciar uma especificação de *template*:

```
Stack<double, 100> s;
```

V.2. Classes parametrizadas: *templates*

- Classes *template* e herança
 - Os conceitos básicos associados à herança também podem ser utilizados em conjunto com classes *templates*
 - Além de gerar classes, podemos gerar hierarquias de classes com *templates*, sendo que cada hierarquia é parametrizada
 - Pode-se criar:
 - Classes *template* derivadas de classes *template*
 - Classes comuns derivadas de instâncias de classes *template*
 - Classes *template* derivadas de classes comuns
 - Classes *template* derivadas de instâncias de classes *template*
 - Funções virtuais são permitidas na hierarquia
 - Consequentemente, podemos ter polimorfismo
 - Pode-se ter também classes *template* abstratas, etc...

V.2. Classes parametrizadas: *templates*

- Classes *template* e herança

```
template <class T> class Base {  
    T b; // Classe template  
};  
template<class T> class D1 : public Base <T> {  
    T c; // template derivada de template  
};  
template <class T> class D2 : public Base <int> {  
    T d; // template derivada de instancia  
};  
class D3 : public Base <float> {  
    int f; // classe comum derivada de instancia  
};  
template <class T> class D4 : public D3 {  
    T g; // template derivada de classe comum  
};
```

V.2. Classes parametrizadas: *templates*

■ *Templates e header files*

- A maioria dos compiladores C++ requer a exibição da definição completa de um *template* no arquivo de código-fonte cliente que utiliza o *template*
- Os *templates* são frequentemente definidos em arquivos de cabeçalho (.h), que são incluídos nos arquivos de código-fonte dos clientes
 - Para os *templates* de classe, isso quer dizer que as funções membro também são definidas no arquivo de cabeçalho
 - Isto pode parecer que viola a regra normal dos arquivos de cabeçalho, mas definições de *templates* são especiais, pois o compilador necessita efetuar a substituição de código fonte

Exercícios

2. Escreva uma função *template* que retorna o n -ésimo termo da série de Fibonacci podendo o valor de retorno ser int, long, float, etc.
3. Por que você escolheria utilizar um *template* de função em vez de uma macro?
4. Que problema de desempenho pode resultar do uso de *templates* de função e *templates* de classe?

Exercícios

5. Sobrecarregue o *template* de função **printArray** apresentado abaixo com um outro *template* de função que aceite dois argumentos do tipo inteiro adicionais (**lowSubscript** e **highSubscript**). Se esses inteiros possuírem valores válidos, a função deve retornar a quantidade de elementos impressos. Caso contrários, ela deve retornar 0. Escreva a função **main()** para testar a função sobrecarregada e o *template*.

```
template <class T>
void printArray(const T array[], int count) {
    for (int i = 0; i < count; i++)
        cout<<array[i]<<" ";
    cout<<endl;
}
```