

C++ EXCEPTIONS

Thiago R Adams - 2009

<http://www.thradams.com/>
thiago.adams@gmail.com

Agenda

- Introdução
- Visão geral
- Mecanismo de exceções
- Exception safety
- Comparações com código de retorno
- Adições para C++ ox
- Perguntas

Objetivos

Abordar o problema da propagação de erros com a utilização de exceções sob o ponto de vista prático, fazendo comparações com outros mecanismos de propagação

Visão geral

Por que tratamos erros?

- Erros ocorrem
- Evitar estado ilegal da aplicação
- Evitar fechar a aplicação

Por que propagar erros?

- Nem sempre é possível tratar o erro no ponto aonde ele é detectado e vice versa.

Formas de propagação do erro

- Retornar código de erro
- Sim ou não + "GetLastError"
- Retornar valor inválido
- **Exceções**

Motivação para exceções

- Não modifica assinatura da função
- Propagação automática
- Impossível ignorar o erro

Mecanismo

Mecanismo

```
void Inicio()  
{  
  try {  
    Parte1();  
  }  
  catch(Erro)  
  {  
    // tratamento  
  }  
}
```

void Parte1()
{
 Parte2();
}

void Parte2()
{
 //erro?
 throw Erro();
}



Características

- Mecanismo não local
- Síncrono
- Permite polimorfismo do erro
ex: `class ZeroDiv : public MathError { }`
- Não obriga classe base
- Sucesso é implícito

Detalhes do catch

- A ordem do catch importa (ifs)

```
catch(const ZeroDiv&)  
{  
}  
catch(const MathError&)  
{  
}
```

- A maneira como declarar o tipo importa (object slicing)

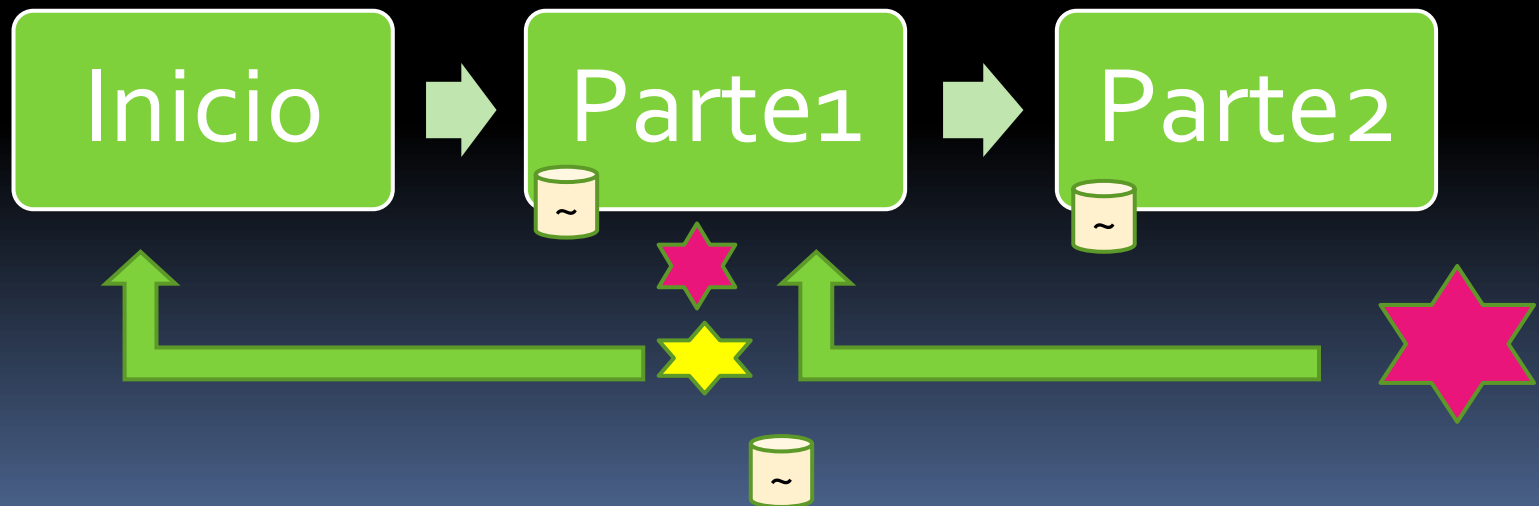
Esquecimento do catch?

- `std::terminate();`
- Customização – `std::set_terminate`



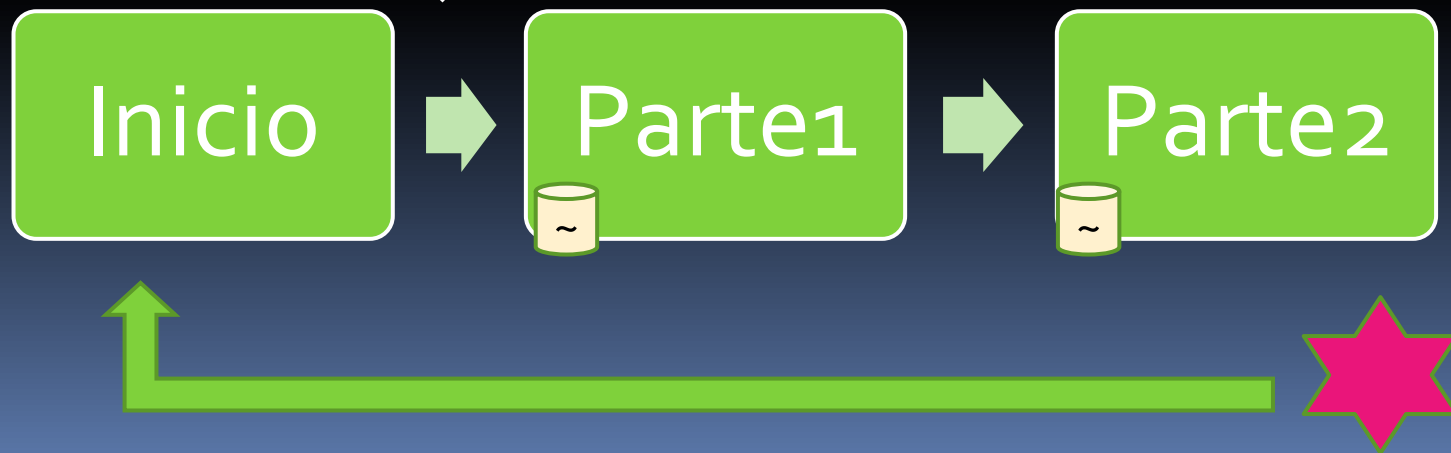
Re-lançar uma exceção

- *catch(...)* "catch all"
- *try { } catch(...) { throw; }*
- *try { } catch(...) { throw Outro(); }*



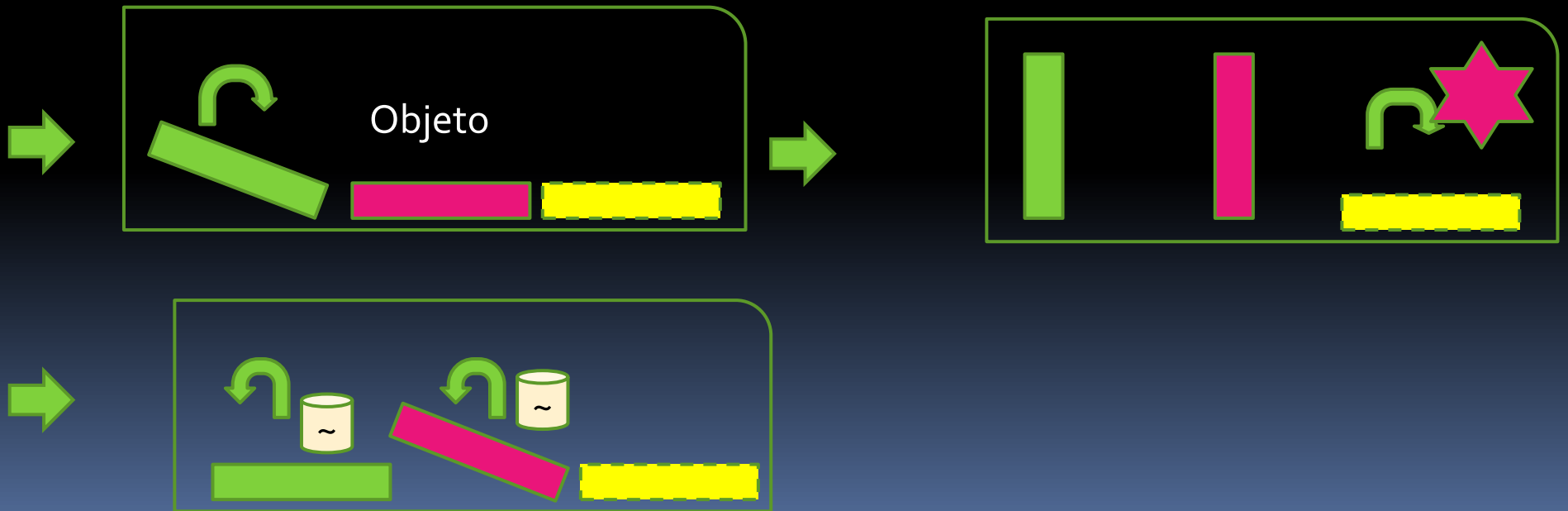
Stack unwinding

- Processo da troca de escopo do ponto do throw até o ponto do catch
- Todos destrutores são chamados
- Requer RAII – (Resource acquisition is initialization)



Exceções em construtores

- Recomendado
- `~Destrutor()` Nunca é chamado



Invariante no construtor

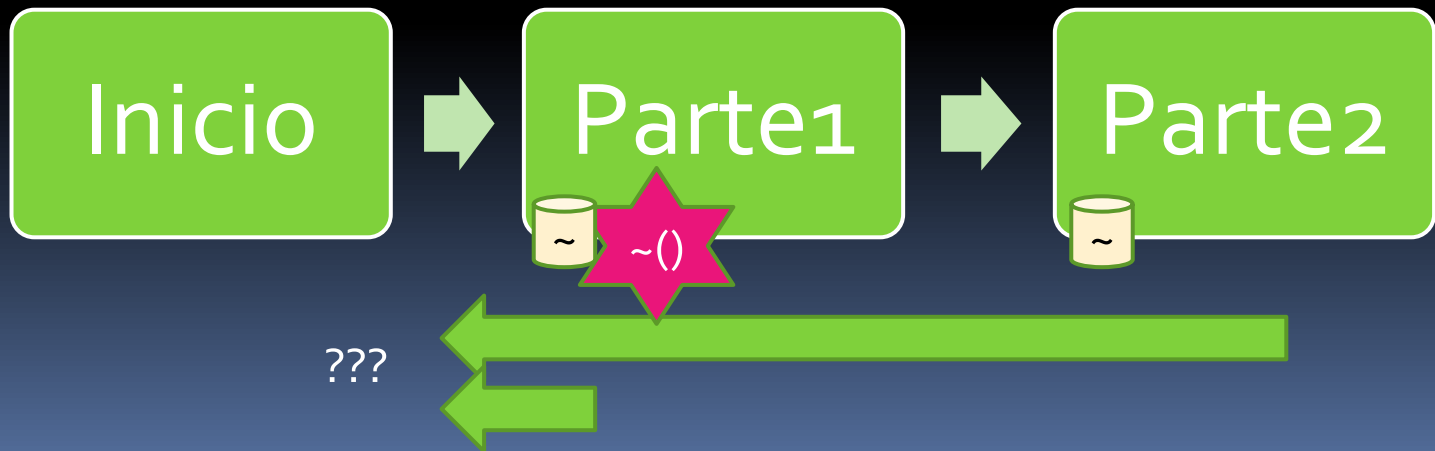
```
class X {  
    Pointer * p;  
    X(const char* s) {  
        p = Create(s);  
    }  
  
    void F() {  
        //p is valid !  
    }  
};
```

```
class X {  
    Pointer * p;  
    X() { p = null; }  
  
    ErrorCode Create() {  
        p = Create(s);  
        if (p == null)  
            return error;  
    }  
  
    ErrorCode F() {  
        if (!p) ErrorCode;  
    }  
};
```

Exceções em destrutores

- O destructor pode ser chamado durante o stack unwinding.

*Don't interrupt me
while I'm interrupting.
– Winston S. Churchill*

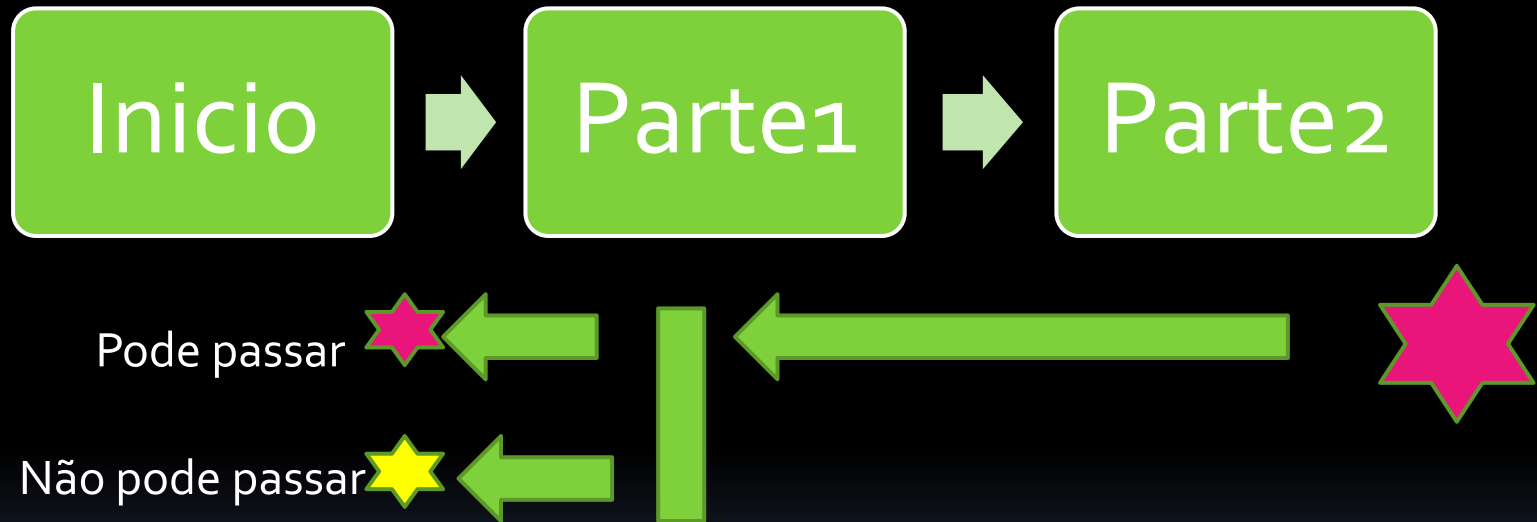


Exceções em destrutores

- STL requer que os tipos não lancem exceções no destrutor.
- O stack unwinding pode ser detectado:

```
~X()  
{  
    if (!uncaught_exception())  
        throw Y;  
}
```
- De maneira geral não é recomendado lançar exceções no destrutor.

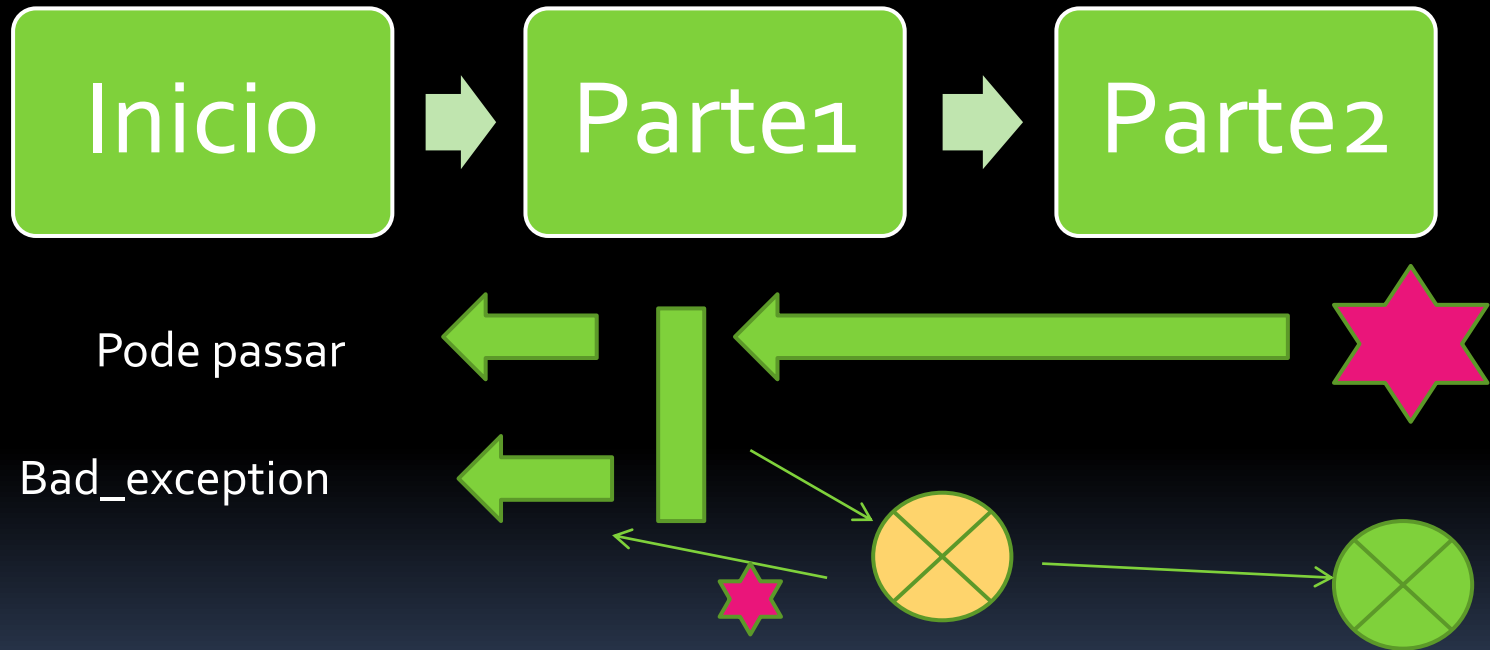
Especificação de exceções



Especificação de exceções

- `void f() throw();`
`void f2() throw (...);`
`void f3() throw (X, Y);`
- Visual C++ não suporta
- Somente em runtime
- Faz parte do tipo?
- Templates ?
- Não é recomendado

Especificação de exceções

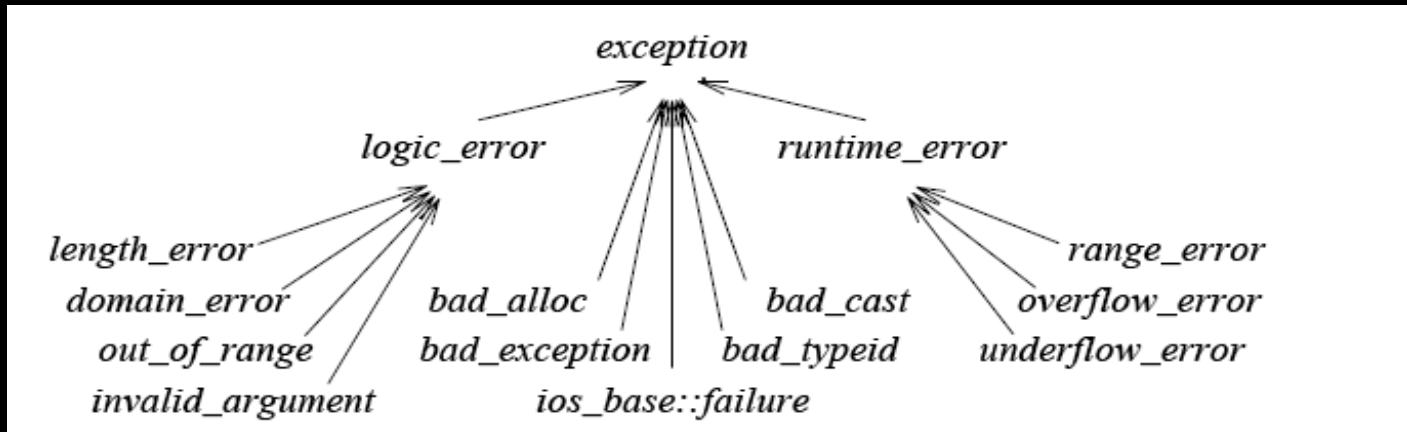


Exceções do C++

Standard Exceptions (thrown by the language)			
Name	Thrown by	Reference	Header
<i>bad_alloc</i>	<i>new</i>	§6.2.6.2, §19.4.5	<new>
<i>bad_cast</i>	<i>dynamic_cast</i>	§15.4.1.1	<typeinfo>
<i>bad_typeid</i>	<i>typeid</i>	§15.4.4	<typeinfo>
<i>bad_exception</i>	<i>exception specification</i>	§14.6.3	<exception>

Standard Exceptions (thrown by the standard library)			
Name	Thrown by	Reference	Header
<i>out_of_range</i>	<i>at()</i>	§3.7.2, §16.3.3, §20.3.3	<stdexcept>
	<i>bitset<>::operator[]()</i>	§17.5.3	<stdexcept>
<i>invalid_argument</i>	<i>bitset constructor</i>	§17.5.3.1	<stdexcept>
<i>overflow_error</i>	<i>bitset<>::to_ulong()</i>	§17.5.3.3	<stdexcept>
<i>ios_base::failure</i>	<i>ios_base::clear()</i>	§21.3.6	<ios>

Exceções do C++



```
class exception
{
public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
};
```

0 caso do new

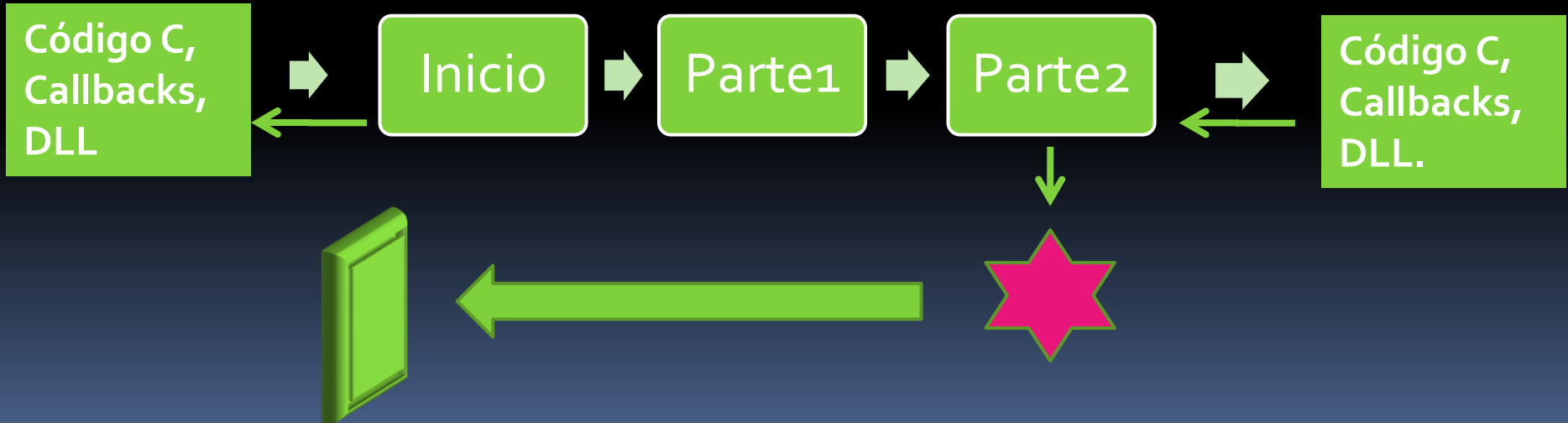
- Falha -> throw bad_alloc
- Difícil de ser contornado
- Libs podem mudar este comportamento

0 caso do new nothrow

- Operador new executa duas tarefas.
 - 1 - Alocação da memória do objeto
 - 2 - Criação do objeto
- Na primeira fase o new nothrow retorna null em caso de falha.
- Na segunda fase exceções escapam e o retorno não é null

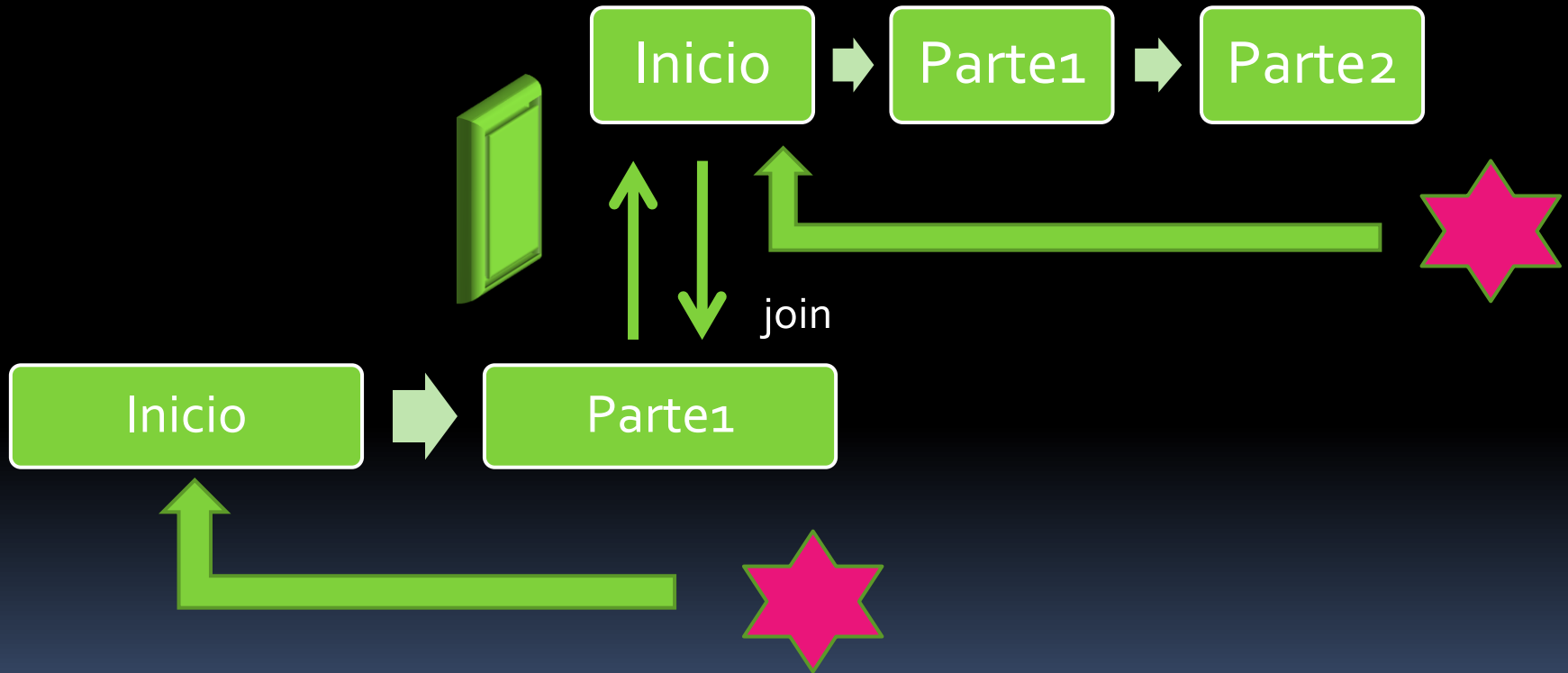
Fronteira C – C++

- Exceções não vem de código C
Exceções não podem ir para código C



Barreira com catch

Fronteira C++ - thread



O que lançar?

- Tipos que só tem esta finalidade
- Derivados de uma classe base
- Representação de códigos de erro
- Homogeneidade
- Não lançar ponteiros
- Mensagem não é para o usuário
- Classe mais especializada que existir

Quando lançar uma exceção?

- Quando é possível
- Pelos mesmos motivos aonde se propaga erros
- Tarefa não pode ser completada de acordo com a assinatura da função

T * find(key)

T* get(key)

T& get(key)

Quando lançar uma exceção?

- Construtores
- Operadores
- Não lançar exceções em destrutores
- Considerar a performance
- Considerar escopo do uso
- Considerar relação com API nativa
- Considerar criar uma função para lançar exceção

Aonde colocar o catch?

- Aonde é preciso
- Aonde for possível tratar o erro
- Quando a operação tem um significado completo
- Não abuse
- Catch ... Throw ? RAI

Como testar?

- Nada especial
- Verificar caminho do erro
- Forçar o erro
- Unit test

Exception safety

O que é Exception safety?

- Bom comportamento
- Sem memory leaks
- Invariante intacto
- Neutralidade para templates e funções desconhecidas
- Não é privilégio das exceções.
- É um problema local

Classificação das funções na presença de erros

- Garantia básica – *basic guarantee*
- Garantia forte – *strong guarantee*
- Não falha – *no fail*

Garantia básica

- Estado pode estar alterado, mas consistente
- Sem leaks
- Seguro para destruição

Garantia forte

- Estado inalterado
- Estilo - Commit / Rollback

Garantia de não falhar

- Nunca falha
- Exemplos:
get(), destructor, release, destroy, swap,

Garantias na STL

Container-Operation Guarantees				
	vector	deque	list	map
<i>clear()</i>	nothrow (<i>copy</i>)	nothrow (<i>copy</i>)	nothrow	nothrow
<i>erase()</i>	nothrow (<i>copy</i>)	nothrow (<i>copy</i>)	nothrow	nothrow
<i>1-element insert()</i>	strong (<i>copy</i>)	strong (<i>copy</i>)	strong	strong
<i>N-element insert()</i>	strong (<i>copy</i>)	strong (<i>copy</i>)	strong	basic
<i>merge()</i>	—	—	nothrow (<i>comparison</i>)	—
<i>push_back()</i>	strong	strong	strong	—
<i>push_front()</i>	—	strong	strong	—
<i>pop_back()</i>	nothrow	nothrow	nothrow	—
<i>pop_front()</i>	—	nothrow	nothrow	—
<i>remove()</i>	—	—	nothrow (<i>comparison</i>)	—
<i>remove_if()</i>	—	—	nothrow (<i>predicate</i>)	—
<i>reverse()</i>	—	—	nothrow	—
<i>splice()</i>	—	—	nothrow	—
<i>swap()</i>	nothrow	nothrow	nothrow	nothrow (<i>copy-of-comparison</i>)
<i>unique()</i>	—	—	nothrow (<i>comparison</i>)	—

Técnicas para gerar funções exception safety

- RAII
- Estilo commit
- Temporários
- Smart pointers
- Use funções com garantia “no fail”

Exemplo – Montar um vector em uma função com garantia forte.

```
void make(std::vector<int>& v)
{
    std::vector<int> temporario;
    for (int i = 0; i < 100; i++)
    {
        temporario.push_back(NewInteger(i));
    }

    v.swap(temporario); //não falha
}
```

Exemplo: Fazer transferência de dono

```
void AddNew(vector<T*>& v)
{
    std::auto_ptr<T> sp(new T());
    v.push_back(sp.get());
    sp.release();
}
```

Exemplo – caso do construtor

```
class X
{
    T * m_p1, m_p2;
    X {
        auto_ptr<T> sp1(new T);
        auto_ptr<T> sp2(new T);
        m_p1 = sp1.release(); //commit
        m_p2 = sp2.release(); //commit
    }
    ~X() {
        delete m_p1;
        delete m_p2;
    }
}
```

```
class X
{
    auto_ptr<T> m_p1, m_p2;
    X : m_p1(new T),
        m_p2(new T)
    {
    }
}
```

Estilo commit

- ```
void f()
{
 temporários
 Parte "nothrow"
 Parte throw(...)
 Parte "nothrow"
 //commit
}
```
- Preparado para throw - RAll
- Rollback
- Rollback sempre?

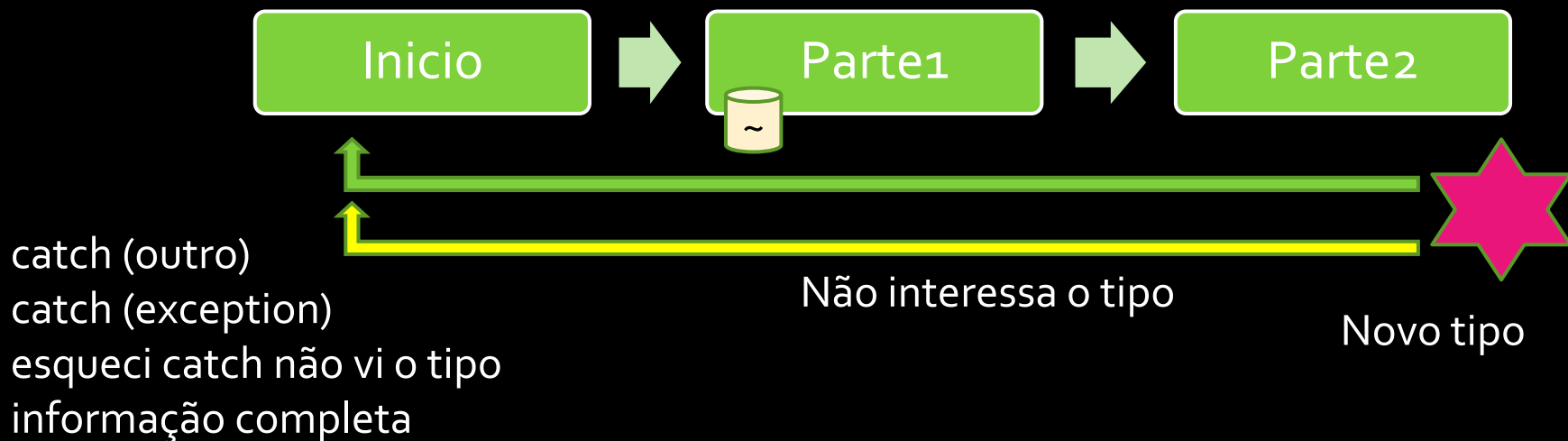
# Discussão

- Existe uma melhor ou pior garantia?
- É possível sempre ter a garantia forte?
- Documentação das garantias

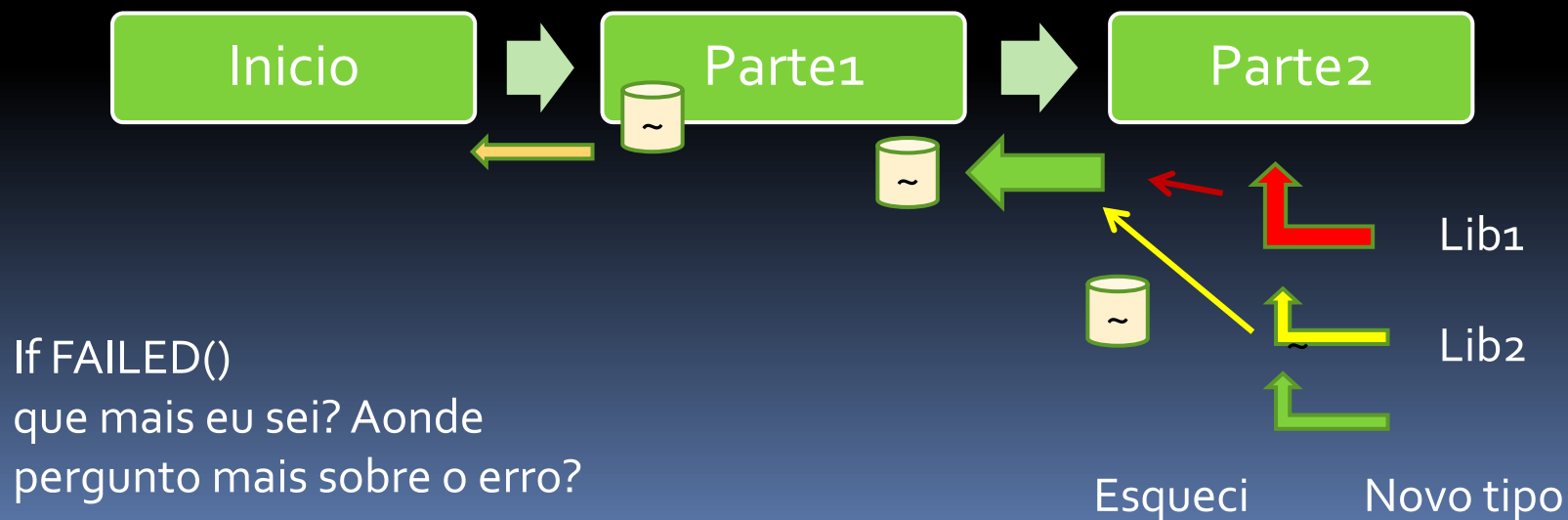
# Comparações

# Comparação com código de retorno

Exceções



Código retorno



# Comparação com código de retorno

- Nem todos compiladores suportam exceções
- Código de retorno tem o teste imediato – são mais simples quando este é o objetivo
- Uma função auxiliar escrita com código de retorno se integra facilmente nos dois tipos de propagação de erros. A recíproca não é verdadeira

# Cuidado ao misturar os dois estilos

- Uma função só deve ter um tipo de propagação de erro
- É melhor ter um função com código de retorno auxiliando a função que lança exceções do que o contrário.
- O método de propagação se propaga
- Só propagar um método de propagação de erros no módulo
- Usa STL ? Então usa propagação por exceções.

# Quando usar um ou outro?

- Você pode usar os dois no mesmo projeto.  
Não na mesma função!
- Prefira exceções quando você tem que quebrar a assinatura da sua função para acomodar o código de erro.
- Integração com C
- Homogenidade

# Quando usar um ou outro?

- Construtores , operadores
- Quanto maior seu “call stack” maior o sinal que a propagação por código de retorno é manual e suscetível a erros.

# Como manter a clareza ao usar dois métodos de propagação?

- Funções com retorno void não propagam código de erro
- Funções com retorno void e documentadas com `throw()` não propagam exceções  
ex: `void f() throw();`

# Como manter a clareza ao usar dois métodos de propagação?

- Funções com código de retorno não propagam exceções  
ex: HRESULT Create();
- Todas as outras podem lançar exceções
- Tenha uma política para tratamento de erros e um padrão homogêneo no mínimo por módulos

Novidades para o C++ 0x

# Exceptions e C++ 0x

- `exception_ptr current_exception();`
- `void rethrow_exception(exception_ptr p);`  
`template<class E> exception_ptr`  
`copy_exception(E e);`

# Exceptions in C++ 0x

- `nested_exception`
- `template<class T> void  
throw_with_nested(T&& t); // [[noreturn]]`
- `template <class E> void  
rethrow_if_nested(const E& e);`

Perguntas?

# Referências

- The C++ Programming Language  
- Stroustrup
- Exceptional C++ Style: 40 New Engineering Puzzles,  
Programming Problems, and Solutions - Sutter
- Working Draft, Standard for Programming  
Language C++ [N2914=09-0104]
- [http://www.boost.org/community/exception\\_safety.html](http://www.boost.org/community/exception_safety.html)
- [http://msdn.microsoft.com/en-us/library/ms229014\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms229014(VS.80).aspx)