16

Tratamento de exceções

OBJETIVOS

Neste capítulo, você aprenderá:

- O que são exceções e quando utilizá-las.
- A utilizar **`try`**, **`catch`** e **`throw`** para detectar, indicar e tratar exceções, respectivamente.
- A processar exceções não interceptadas e inesperadas.
- Como declarar novas classes de exceção.
- Como o desempilhamento permite que exceções não capturadas em um escopo sejam capturadas em outro escopo.
- A tratar falhas **`new`**.
- Como utilizar **`auto_ptr`** para evitar vazamentos de memória.
- A entender a hierarquia de exceções-padrão.

16.1 Introdução

- **Exceções**
  - **Indicam os problemas que ocorrem durante a execução de um programa.**
  - **Ocorrem raramente.**
- **Tratamento de exceções**
  - **Pode solucionar exceções**
    - **Permite que um progrma continue a executar ou**
    - **Notifica o usuário sobre o problema e**
    - **Encerra o programa de uma maneira controlada.**
  - **Torna os programas robustos e tolerantes a falhas.**

Dica de prevenção de erro 16.1

O tratamento de exceções ajuda a aprimorar a tolerância a falhas de um programa.

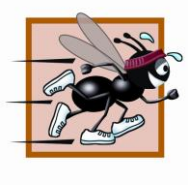
Observação de engenharia de software 16.1

O tratamento de exceções fornece um mecanismo-padrão para processar erros. Isso é especialmente importante ao trabalhar em um projeto com uma grande equipe de programadores.

16.2 Visão geral do tratamento de exceções

- **Misturando programa e lógica de tratamento de erros**
  - **Exemplo de pseudocódigo**

    Execute uma tarefa
    Se a tarefa precedente não tiver sido executada corretamente
      Realize processamento de erro
    Execute a tarefa seguinte
     Se a tarefa precedente não tiver sido executada corretamente
      Realize processamento de erro
    …
  - **Torna o programa difícil de ler, modificar, manter e depurar.**



Dica de desempenho 16.1

Se problemas em potencial ocorrem raramente, misturar a lógica do programa e a lógica do tratamento de erro pode diminuir seu desempenho, porque o programa tem de realizar testes (possivelmente freqüentes) para determinar se a tarefa foi executada corretamente e se a próxima tarefa pode ser realizada.

16.2 Visão geral do tratamento de exceções (cont.)

- **Tratamento de exceções**
  - **Remove código de tratamento de erros da 'linha principal' de execução do programa.**
  - **Os programadores podem tratar qualquer exceção que quiserem.**
    - **Todas as exceções,**
    - **Todas as exceções de um determinado tipo ou**
    - **Todas as exceções de um grupo de tipos relacionados.**

16.3 Exemplo: tratando uma tentativa de divisão por zero

- Classe `exception`
  - É a classe básica padrão do C++ para todas as exceções.
  - Fornece às suas classes derivadas a função `virtual what`.
    - Retorna a mensagem de erro armazenada da exceção.

```
1  // Figura 16.1: DivideByZeroException.h
2  // Definição da classe DivideByZeroException.
3  #include <stdexcept> // arquivo de cabeçalho stdexcept contém runtime_error
4  using std::runtime_error; // classe runtime_error da biblioteca-padrão do C++
5
6  // objetos DivideByZeroException devem ser lançados por funções
7  // ao detectar exceções de divisão por zero
8  class DivideByZeroException : public runtime_error
9  {
10 public:
11    // construtor especifica a mensagem de erro-padrão
12    DivideByZeroException::DivideByZeroException()
13       : runtime_error( "attempted to divide by zero" ) {}
14 }; // fim da classe DivideByZeroException
```

```
1  // Figura 16.2: Fig16_02.cpp
2  // Um exemplo simples de tratamento de exceções que verifica
3  // exceções de divisão por zero.
4  #include <iostream>
5  using std::cin;
6  using std::cout;
7  using std::endl;
8
9  #include "DivideByZeroException.h" // classe DivideByZeroException
10
11 // realiza a divisão e lança o objeto DivideByZeroException se
12 // a exceção de divisão por zero ocorrer
13 double quotient( int numerator, int denominator )
14 {
15    // lança DivideByZeroException se tentar dividir por zero
16    if ( denominator == 0 )
17       throw DivideByZeroException(); // termina a função
18
19    // retorna resultado da divisão
20    return static_cast< double >( numerator ) / denominator;
21 } // fim da função quotient
22
23 int main()
24 {
25    int number1; // numerador especificado pelo usuário
26    int number2; // denominador especificado pelo usuário
27    double result; // resultado da divisão
28
29    cout << "Enter two integers (end-of-file to end): ";
```

```
30
31    // permite ao usuário inserir dois inteiros para dividir
32    while ( cin >> number1 >> number2 )
33    {
34       // bloco try contém código que poderia lançar exceção
35       // e código que não deve executar se uma exceção ocorrer
36       try
37       {
38          result = quotient( number1, number2 );
39          cout << "The quotient is: " << result << endl;
40       } // fim do try
41
42       // handler de exceção trata uma exceção de divisão por zero
43       catch ( DivideByZeroException &divideByZeroException )
44       {
45          cout << "Exception occurred: "
46             << divideByZeroException.what() << endl;
47       } // fim do catch
48
49       cout << "\nEnter two integers (end-of-file to end): ";
50    } // fim do while
51
52    cout << endl;
53    return 0; // termina normalmente
54 } // fim de main
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

16.3 Exemplo: tratando uma tentativa de divisão por zero (cont.)

- **Blocos `try`**
  - **A palavra-chave `try` é seguida por chaves (`{}`).**
  - **Deve encerrar**
    - **Instruções que possam provocar exceções e**
    - **Instruções que devem ser puladas no caso de uma exceção.**

Observação de engenharia de software 16.2

As exceções podem emergir pelo código explicitamente mencionado em um bloco `try`, por meio de chamadas a outras funções e por meio de chamadas de função profundamente aninhadas, iniciadas pelo código em um bloco `try`.

16.3 Exemplo: tratando uma tentativa de divisão por zero (cont.)

- **Handlers `catch`**
  - **Seguem imediatamente um bloco `try`.**
    - **Um ou mais handlers `catch` para cada bloco `try`.**
  - **Palavra-chave `catch`.**
  - **Parâmetro de exceção encerrado entre parênteses**
    - **Representa o tipo de exceção a ser processado.**
    - **Pode oferecer um nome de parâmetro opcional para interagir com o objeto de exceção capturado.**
  - **Executa se o tipo de parâmetro de exceção corresponder à exceção lançada no bloco `try`.**
    - **Poderia ser uma classe básica da classe da exceção lançada.**

Erro comum de programação 16.1

É um erro de sintaxe colocar código entre um bloco `try` e seus blocos `catch` correspondentes.

Erro comum de programação 16.2

Cada handler `catch` pode ter um único parâmetro — especificar uma lista de parâmetros de exceção separados por vírgulas é um erro de sintaxe.

Erro comum de programação 16.3

É um erro de lógica capturar o mesmo tipo em dois handlers `catch` diferentes que se seguem a um único bloco `try`.

16.3 Exemplo: tratando uma tentativa de divisão por zero (cont.)

- **Modelo de terminação do tratamento de exceções**
  - **O bloco `try` expira quando ocorre uma exceção.**
    - **As variáveis locais no bloco `try` saem do escopo.**
  - **O código dentro do handler `catch` correspondente executa.**
  - **O controle é retomado com a primeira instrução após o último handler `catch` subseqüente ao bloco `try`.**
    - **O controle não retorna ao ponto em que a exceção ocorreu.**
- **Desempilhamento**
  - **Ocorre se nenhum handler `catch` correspondente for encontrado.**
  - **O programa tenta localizar outro bloco `try` envolvente na função chamadora.**

Erro comum de programação 16.4

Erros de lógica podem ocorrer se você assumir que, depois que uma exceção for tratada, o controle retornará à primeira instrução depois do ponto de lançamento.

Dica de prevenção de erro 16.2

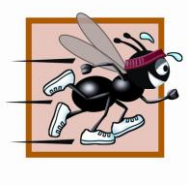
Com o tratamento de exceções, um programa pode continuar executando (em vez de encerrar) depois de lidar com um problema. Isso ajuda a assegurar um tipo de aplicativo robusto que colabora para o que é chamado de computação de missão crítica ou computação de negócios críticos.

16.3 Exemplo: tratando uma tentativa de divisão por zero (cont.)

- **Lançando uma exceção**
  - **Use a palavra-chave `throw` seguida de um operando que represente o tipo de exceção.**
    - **O operando `throw` pode ser de qualquer tipo.**
      - **Se o operando `throw` for um objeto, é chamado de objeto de exceção.**
  - **O operando `throw` inicializa o parâmetro de exceção no handler `catch` correspondente, se encontrar algum.**

Erro comum de programação 16.5

Seja atencioso ao lançar (`throwing`) o resultado de uma expressão condicional (?:), porque as regras de promoção poderiam fazer com que o valor fosse de um tipo diferente do esperado. Por exemplo, ao lançar um `int` ou um `double` da mesma expressão condicional, a expressão condicional converte o `int` em um `double`. Entretanto, o handler `catch` sempre captura o resultado como um `double`, em vez de capturá-lo como um `double` quando um `double` é lançado, e capturá-lo como um `int` quando um `int` é lançado.



Dica de desempenho 16.2

Capturar um objeto de exceção por referência elimina o overhead de copiar o objeto que representa a exceção lançada.

Boa prática de programação 16.1

Associar cada tipo de erro de tempo de execução com um objeto de exceção adequadamente nomeado aumenta a clareza do programa.

16.4 Quando utilizar o tratamento de exceções

- **Quando usar o tratamento de exceções**
  - **Para processar erros síncronos**
    - **Que ocorrem quando uma instrução é executada.**
  - **Não para processar erros assíncronos**
    - **Que ocorrem paralelamente à e independentemente da execução do programa.**
  - **Para processar problemas em elementos predefinidos do software.**
    - **Como funções e classes predefinidas.**
    - **O tratamento de erros pode ser executado pelo código do programa a ser personalizado com base nas necessidades do aplicativo.**

Observação de engenharia de software 16.3

Incorpore sua estratégia de tratamento de exceções no sistema desde o princípio do processo de projeto. Pode ser difícil incluir um tratamento de exceções eficiente depois que um sistema já foi implementado.

Observação de engenharia de software 16.4

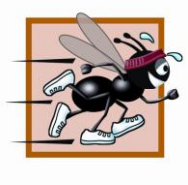
O tratamento de exceções fornece uma técnica única e uniforme para processamento de problemas. Isso ajuda os programadores em grandes projetos a entender o código de processamento de erro uns dos outros.

Observação de engenharia de software 16.5

Evite utilizar o tratamento de exceções como uma forma alternativa de fluxo de controle. Essas exceções 'adicionais' podem 'entrar no caminho' de verdadeiras exceções do tipo erro.

Observação de engenharia de software 16.6

O tratamento de exceções simplifica a combinação de componentes de software e permite trabalhar em conjunto eficientemente, possibilitando que os componentes predefinidos comuniquem problemas para componentes específicos ao aplicativo, que então podem processar os problemas de maneira específica ao aplicativo.



Dica de desempenho 16.3

Quando não ocorrer nenhuma exceção, o código de tratamento de exceções não provoca ou provoca poucos prejuízos no desempenho. Portanto, os programas que implementam o tratamento de exceções operam com mais eficiência do que os programas que mesclam o código de tratamento de erros com a lógica do programa.

Observação de engenharia de software 16.7

As funções com condições de erro comuns devem retornar `0` ou `NULL` (ou outros valores adequados) em vez de lançar exceções. Um programa que chama tal função pode verificar o valor de retorno para determinar o sucesso ou falha da chamada de função.

16.5 Relançando uma exceção

- **Relançando uma exceção**
  - **Instrução `throw;` vazia.**
  - **Usada quando um handler `catch` não pode ou só pode processar parcialmente uma exceção.**
  - **O próximo bloco `try` envolvente tenta corresponder à exceção com um de seus handlers `catch`.**

Erro comum de programação 16.6

Executar uma instrução `throw` vazia situada fora de um handler `catch` produz uma chamada à função `terminate`, que abandona o processamento de exceção e termina o programa imediatamente.

fig16_03.cpp
(1 de 2)

```
1  // Figura 16.3: Fig16_03.cpp
2  // Demonstrando o relançamento de exceção.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <exception>
8  using std::exception;
9
10 // lança, captura e relança a exceção
11 void throwException()
12 {
13    // lança a exceção e a captura imediatamente
14    try
15    {
16       cout << "  Function throwException throws an exception\n";
17       throw exception(); // gera a exceção
18    } // fim do try
19    catch ( exception & ) // trata a exceção
20    {
21       cout << "  Exception handled in function throwException"
22          << "\n  Function throwException rethrows exception";
23       throw; // relança a exceção para processamento adicional
24    } // fim do catch
25
26    cout << "This also should not print\n";
27 } // fim da função throwException
```

Relança uma exceção.

fig16_03.cpp
(2 de 2)

```
28
29 int main()
30 {
31    // lança a exceção
32    try
33    {
34       cout << "\nmain invokes function throwException\n";
35       throwException();
36       cout << "This should not print\n";
37    } // fim do try
38    catch ( exception & ) // trata a exceção
39    {
40       cout << "\n\nException handled in main\n";
41    } // fim do catch
42
43    cout << "Program control continues after catch in main\n";
44    return 0;
45 } // fim de main
```

Captura a exceção relançada.

```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

16.6 Especificações de exceção

- **Especificações de exceção (listas também conhecidas como `throw`)**
  - **Palavra-chave `throw`**
  - **Lista separada por vírgulas de classes de exceção entre parênteses**
  - **Exemplo**
    - ```
      int someFunction( double value )
          throw ( ExceptionA, ExceptionB,
                  ExceptionC )
      {
          ...
      }
      ```
      - **Indica que `someFunction` pode lançar exceções dos tipos `ExceptionA`, `ExceptionB` e `ExceptionC`.**
      ```

16.6 Especificações de exceção (cont.)

- **Especificações de exceção (cont.)**
  - **Uma função pode `throw` (lançar) exceções somente dos tipos em sua especificação ou de tipos derivados desses tipos.**
    - **Se uma função lançar uma exceção que não pertença ao tipo especificado, a função `unexpected` será chamada.**
      - **Isso normalmente encerra o programa.**
  - **Se não houver nenhuma especificação, isso quer dizer que a função pode lançar qualquer exceção.**
  - **Uma especificação de exceção vazia, `throw()`, indica que a função não pode lançar nenhuma exceção.**

Erro comum de programação 16.7

Lançar uma exceção que não foi declarada em uma especificação de exceção da função produz uma chamada à função `unexpected`.

Dica de prevenção de erro 16.3

O compilador não gerará um erro de compilação se uma função contiver uma expressão `throw` para uma exceção não listada na especificação de exceção da função. Só ocorrerá erro quando essa função tentar lançar essa exceção em tempo de execução. Para evitar surpresas em tempo de execução, verifique cuidadosamente seu código para que as funções não lancem exceções não listadas em suas especificações de exceção.

16.7 Processando exceções inesperadas

- **Função `unexpected`**
  - **É chamada quando uma função lança uma exceção que não se encontra em sua especificação de exceção.**
  - **Chama a função registrada com a função `set_unexpected`.**
  - **A função `terminate` é chamada por padrão.**
- **Função `set_unexpected` of `<exception>`**
  - **Aceita como argumento um ponteiro para uma função sem nenhum argumento e um tipo de retorno `void`.**
  - **Retorna um ponteiro para a última função chamada por `unexpected`.**
    - **Retorna `0` na primeira vez.**

16.7 Processando exceções inesperadas (cont.)

- **Função `terminate`**
  - **É chamada quando**
    - **Não é encontrado nenhum `catch` correspondente à exceção lançada.**
    - **Um destrutor tenta lançar uma exceção durante o desempilhamento.**
    - **Há a tentativa de relançar uma exceção no momento em que nenhuma exceção está sento tratada.**
    - **Se a função `unexpected` for chamada antes de registrar uma função com a função `set_unexpected`.**
  - **Chama a função registrada com a função `set_terminate`.**
  - **A função `abort` é chamada por padrão.**

16.7 Processando exceções inesperadas (cont.)

- Função `set_terminate`
  - Aceita como argumento um ponteiro para uma função sem nenhum argumento e um tipo de retorno `void`.
  - Retorna um ponteiro para a última função chamada por `terminate`.
    - Retorna 0 na primeira vez.
- Função `abort`
  - Termina o programa sem chamar destrutores para objetos de classe de armazenamento automático ou estático.
    - Pode provocar vazamento de recurso.

16.8 Desempilhamento de pilha

- **Desempilhamento**
  - **Ocorre quando uma exceção lançada não é capturada em um determinado escopo.**
  - **Desempilhar uma função encerra essa função.**
    - **Todas as variáveis locais da função são destruídas.**
    - **O controle retorna à instrução que invocou a função.**
  - **São feitas tentativas de capturar a exceção em blocos `try`...`catch` externos.**
  - **Se a exceção nunca for capturada, a função `terminate` será chamada.**

fig16_04.cpp
(1 de 3)

```
1  // Figura 16.4: Fig16_04.cpp
2  // Demonstrando o desempilhamento.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <stdexcept>
8  using std::runtime_error;
9
10 // function3 lança erro de tempo de execução
11 void function3() throw ( runtime_error )
12 {
13    cout << "In function 3" << endl;
14
15    // nenhum bloco try, o desempilhamento ocorre, retorna controle a function2
16    throw runtime_error( "runtime_error in function3" );
17 } // fim de function3
18
19 // function2 invoca function3
20 void function2() throw ( runtime_error )
21 {
22    cout << "function3 is called inside function2" << endl;
23    function3(); // o desempilhamento ocorre, retorna o controle a function1
24 } // fim de function2
25
26 // function1 invoca function2
27 void function1() throw ( runtime_error )
```

```
28  {
29     cout << "function2 is called inside function1" << endl;
30     function2(); // o desempilhamento ocorre, retorna controle a main
31  } // fim de function1
32
33  // demonstra o desempilhamento
34  int main()
35  {
36     // invoca function1
37     try
38     {
39        cout << "function1 is called inside main" << endl;
40        function1(); // chama function1 que lança runtime_error
41     } // fim do try
42     catch ( runtime_error &error ) // trata erro de tempo de execução
43     {
44        cout << "Exception occurred: " << error.what() << endl;
45        cout << "Exception handled in main" << endl;
46     } // fim do catch
47
48     return 0;
49  } // fim de main
```

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

16.9 Construtores, destrutores e tratamento de exceções

- **Exceções e construtores**
  - **As exceções permitem que os construtores, que não podem retornar valores, informem erros ao programa.**
  - **As exceções lançadas por construtores fazem com que qualquer objeto de componente já construído chame seus destrutores.**
    - **Apenas os objetos que já tiverem sido construídos serão destruídos.**
- **Exceções e destrutores**
  - **Os destrutores são chamados para todos os objetos automáticos no bloco `try` encerrado quando uma exceção é lançada.**
    - **Os recursos adquiridos podem ser colocados nos objetos locais a fim de liberá-los automaticamente no momento em que ocorrer uma exceção.**
  - **Se um destrutor invocado por desempilhamento lançar uma exceção, a função `terminate` será chamada.**

Dica de prevenção de erro 16.4

Quando uma exceção é lançada do construtor de um objeto que é criado em uma expressão `new`, a memória dinamicamente alocada desse objeto é liberada.

16.10 Exceções e herança

- **Herança com classes de exceção**
  - **Novas classes de exceção podem ser definidas para herdar de classes de exceção existentes.**
  - **Um handler para captura para uma determinada classe de exceção também pode capturar exceções de classes derivadas dessa classe.**

Dica de prevenção de erro 16.5

Utilizar herança com exceções permite um handler de exceção para capturar erros relacionados com a notação concisa. Uma abordagem é capturar cada tipo de ponteiro ou referência para um objeto de exceção de classe derivada individualmente, mas uma abordagem mais concisa é capturar as referências ou ponteiros para objetos de exceção de classe básica. Além disso, capturar ponteiros ou referências a objetos de exceção de classe derivada individualmente pode provocar erros, especialmente se o programador se esquecer de testar explicitamente um ou mais dos tipos de referência ou de ponteiros de classe derivada.

16.11 Processando falhas `new`

- **Falhas `new`**
  - **Alguns compiladores lançam uma exceção `bad_alloc`.**
    - **Compatíveis com a especificação-padrão do C++.**
  - **Alguns compiladores retornam `0`.**
    - **Os compiladores compatíveis com o padrão C++ também têm uma versão de `new` que retorna `0`.**
      - **Use a expressão `new( nothrow )`, onde `nothrow` é do tipo `nothrow_t`.**
  - **Alguns compiladores lançam `bad_alloc` se `<new>` estiver incluído.**

fig16_05.cpp
(1 de 2)

```
1  // Figura 16.5: Fig16_05.cpp
2  // Demonstrando new pré-padrão retornando 0 quando a memória
3  // não é alocada.
4  #include <iostream>
5  using std::cerr;
6  using std::cout;
7
8  int main()
9  {
10    double *ptr[ 50 ];
11
12    // aloca memória para ptr
13    for ( int i = 0; i < 50; i++ )
14    {
15       ptr[ i ] = new double[ 50000000 ];
16
17       if ( ptr[ i ] == 0 ) // fez new falhar na alocação de memória
18       {
19          cerr << "Memory allocation failed for ptr[ " << i << " ]\n";
20          break;
21       } // fim do if
22       else // alocação bem-sucedida de memória
23          cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
24    } // fim do for
25
26    return 0;
27 } // fim de main
```

Aloca **50000000** valores **double**.

new retornaria **0** se a operação de alocação de memória falhasse.

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
Memory allocation failed for ptr[ 3 ]
```

fig16_06.cpp
(1 de 2)

```
1  // Figura 16.6: Fig16_06.cpp
2  // Demonstrando new-padrão lançando bad_alloc quando a memória
3  // não pode ser alocada.
4  #include <iostream>
5  using std::cerr;
6  using std::cout;
7  using std::endl;
8
9  #include <new> // operador new padrão
10 using std::bad_alloc;
11
12 int main()
13 {
14    double *ptr[ 50 ];
15
16    // aloca memória para ptr
17    try
18    {
19       // aloca memória para ptr[i]; new lança bad_alloc em caso de falha
20       for ( int i = 0; i < 50; i++ )
21       {
22          ptr[ i ] = new double[ 50000000 ]; // pode lançar exceção
23          cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
24       } // fim do for
25    } // fim do try
```

Aloca **50000000** valores **double**.

```
26
27      // trata exceção bad_alloc
28      catch ( bad_alloc &memoryAllocationException )
29      {
30         cerr << "Exception occurred: "
31            << memoryAllocationException.what() << endl;
32      } // fim do catch
33
34      return 0;
35   } // fim de main
```

new lançará uma exceção **bad_alloc** se a operação de alocação de memória falhar.

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
Exception occurred: bad allocation
```

Observação de engenharia de software 16.8

Para tornar os programas mais robustos, utilize a versão de `new` que lança exceções `bad_alloc` em caso de falhas.

16.11 Processando falhas `new` (cont.)

- **Falhas `new` (cont.)**
  - **Função `set_new_handler`**
    - **Registra uma função para tratar falhas `new`.**
      - **A função registrada é chamada por `new` quando uma operação de alocação de memória falha.**
    - **Aceita como argumento um ponteiro para uma função que não aceita nenhum argumento e retorna `void`.**
    - **O C++ padrão especifica que a função handler de `new` deve:**
      - **Disponibilizar mais memória e permitir que `new` tente novamente**
      - **Lançar uma exceção `bad_alloc` ou**
      - **Chamar a função `abort` ou `exit` para encerrar o programa.**

fig16_07.cpp
(1 de 2)

```
1  // Figura 16.7: Fig16_07.cpp
2  // Demonstrando set_new_handler.
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6
7  #include <new> // operador new-padrão e set_new_handler
8  using std::set_new_handler;
9
10 #include <cstdlib> // protótipo da função abort
11 using std::abort;
12
13 // trata falha de alocação de memória
14 void customNewHandler()
15 {
16    cerr << "customNewHandler was called";
17    abort();
18 } // fim da função customNewHandler
19
20 // utilizando set_new_handler para tratar alocação de memória malsucedida
21 int main()
22 {
23    double *ptr[ 50 ];
```

Cria um função handler de **new** **customNewHandler** definida pelo usuário handler function.

```
24
25    // especifica que customNewHandler deve ser chamado em
26    // caso de falha na alocação de memória
27    set_new_handler( customNewHandler );
28
29    // aloca memória para ptr[i]; customNewHandler será
30    // chamado na falha na alocação de memória
31    for ( int i = 0; i < 50; i++ )
32    {
33       ptr[ i ] = new double[ 50000000 ]; // pode lançar exceção
34       cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
35    } // fim do for
36
37    return 0;
38 } // fim de main
```

Registra **customNewHandler** com **set_new_handler**.

Aloca **50000000** valores **double**.

```
Allocated 50000000 doubles in ptr[ 0 ]
Allocated 50000000 doubles in ptr[ 1 ]
Allocated 50000000 doubles in ptr[ 2 ]
customNewHandler was called
```

16.12 Classe `auto_ptr` e alocação de memória dinâmica

- **Template de classe `auto_ptr`**
  - **É definido no arquivo de cabeçalho `<memory>`.**
  - **Mantém um ponteiro para alocar memória dinamicamente.**
    - **Seu destrutor realiza uma operação `delete` no membro de dados do ponteiro.**
      - **Isso evita vazamento de memória por meio da exclusão da memória alocada dinamicamente, mesmo se ocorrer uma exceção.**
    - **Fornece os operadores sobrecarregados `*` e `->` da mesma maneira que uma variável de ponteiro regular.**
    - **Pode transferir a posse da memória por meio do operador de atribuição sobrecarregado ou do construtor de cópia.**
      - **O último objeto `auto_ptr` que estiver mantendo o ponteiro realizará uma operação `delete` na memória.**

Integer.h

```
1  // Figura 16.8: Integer.h
2  // Definição da classe Integer.
3
4  class Integer
5  {
6  public:
7     Integer( int i = 0 ); // construtor-padrão Integer
8     ~Integer(); // destrutor Integer
9     void setInteger( int i ); // função para configurar Integer
10    int getInteger() const; // função para retornar Integer
11 private:
12    int value;
13 }; // fim da classe Integer
```

```
1  // Figura 16.9: Integer.cpp
2  // Definição da função-membro Integer.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Integer.h"
8
9  // construtor-padrão Integer
10 Integer::Integer( int i )
11    : value( i )
12 {
13    cout << "Constructor for Integer " << value << endl;
14 } // fim do construtor Integer
15
16 // destrutor Integer
17 Integer::~Integer()
18 {
19    cout << "Destructor for Integer " << value << endl;
20 } // fim do destrutor Integer
21
22 // configura o valor Integer
23 void Integer::setInteger( int i )
24 {
25    value = i;
26 } // fim da função setInteger
27
28 // retorna o valor Integer
29 int Integer::getInteger() const
30 {
31    return value;
32 } // fim da função getInteger
```

fig16_10.cpp
(1 de 2)

```
1  // Figura 16.10: Fig16_10.cpp
2  // Demonstrando auto_ptr.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <memory>
8  using std::auto_ptr; // definição da classe auto_ptr
9
10 #include "Integer.h"
11
12 // utiliza auto_ptr para manipular o objeto Integer
13 int main()
14 {
15    cout << "Creating an auto_ptr object that points to an Integer\n";
16
17    // "aponta" auto_ptr para o objeto Integer
18    auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
19
20    cout << "\nUsing the auto_ptr to manipulate the Integer\n";
21    ptrToInteger->setInteger( 99 ); // usa auto_ptr para configurar o valor Integer
22
23    // utiliza auto_ptr para obter o valor Integer
24    cout << "Integer after setInteger: " << ( *ptrToInteger ).getInteger();
25    return 0;
26 } // fim de main
```

Cria um **auto_ptr** para apontar para um objeto **Integer** dinamicamente alocado.

Manipula **auto_ptr** como se fosse um ponteiro para um **Integer**.

A memória alocada dinamicamente é automaticamente excluída de **auto_ptr** quando ele sai do escopo.

```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7

Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99

Terminating program
Destructor for Integer 99
```

Observação de engenharia de software 16.9

Um `auto_ptr` tem restrições em certas operações. Por exemplo, um `auto_ptr` não pode apontar para um `array` ou uma classe contêiner padrão.

16.13 Hierarquia de exceções da biblioteca-padrão

- **Classes de hierarquia de exceções**
  - Classe básica `exception`
    - Contém a função `virtual what` para o armazenamento de mensagens de erro.
    - Classes de exceção derivadas de `exception`
      - `bad_alloc` – lançada por `new`
      - `bad_cast` – lançada por `dynamic_cast`
      - `bad_typeid` – lançada por `typeid`
      - `bad_exception` – lançada por `unexpected`
        - Em vez de encerrar o programa ou chamar a função especificada por `set_unexpected`
        - Usada apenas se `bad_exception` estiver na lista `throw` da função.

Erro comum de programação 16.8

Colocar um handler `catch` que captura um objeto de classe básica antes de um `catch` que captura um objeto de uma classe derivada dessa classe básica é um erro de lógica. A classe básica `catch` captura todos os objetos de classe derivada dessa classe básica. Desse modo, a classe derivada `catch` nunca executará.

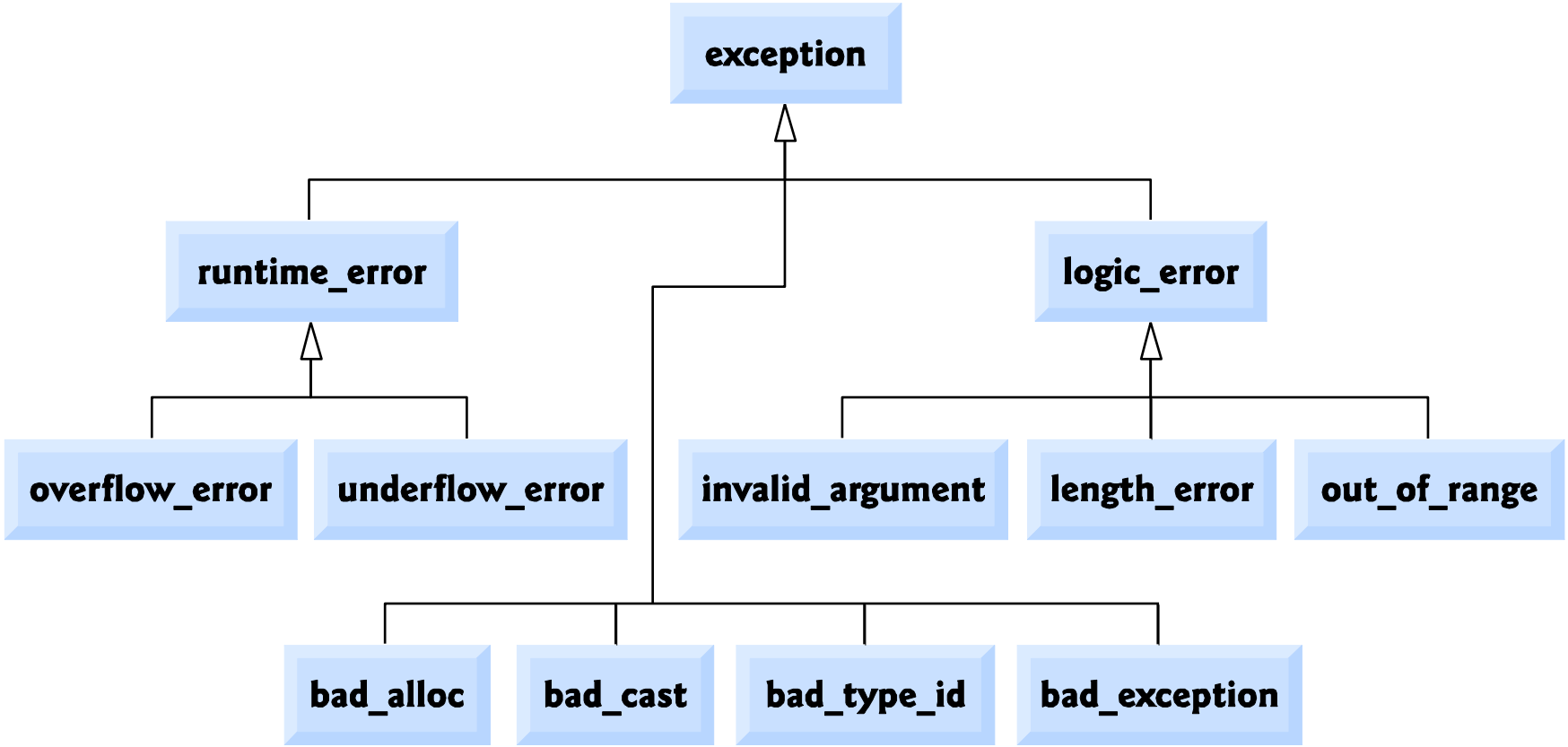


Fig. 16.11 | Classes de exceções da biblioteca-padrão.

16.13 Hierarquia de exceções da biblioteca-padrão (cont.)

- **Classes de hierarquia de exceções (cont.)**
  - Classe `logic_error`, derivada de `exception`
    - Indica erros na lógica do programa.
    - Classes de exceção derivadas de `logic_error`
      - `invalid_argument`
        - Indica um argumento inválido para uma função.
      - `length_error`
        - Indica que foi utilizado um comprimento superior ao tamanho máximo usado para alguns objetos.
      - `out_of_range`
        - Indica que um valor, como um subscrito de array, ultrapassou o intervalo permitido.

16.13 Hierarquia de exceções da biblioteca-padrão (cont.)

- **Classes de hierarquia de exceções (cont.)**
  - **Classe `runtime_error`, derivada de `exception`**
    - **Indica erros de tempo de execução.**
    - **Classes de exceção derivadas de `runtime_error`**
      - **`overflow_error`**
        - **Indica um erro de overflow aritmético — um resultado aritmético é maior que o maior número armazenável.**
      - **`underflow_error`**
        - **Indica um erro de underflow aritmético — um resultado aritmético é menor que o menor número armazenável.**

Erro comum de programação 16.9

As classes de exceções definidas pelo programador não precisam ser derivadas da classe `exception`. Portanto, escrever `catch( exception anyException )` não garante a captura (`catch`) de todas as exceções que um programa poderia encontrar.

Dica de prevenção de erro 16.6

Para capturar todas as exceções potencialmente lançadas em um bloco `try`, utilize `catch(...)`. Um ponto fraco desse método de captura de exceções é que o tipo da exceção capturada é desconhecido em tempo de compilação. Outro ponto fraco é que, sem um parâmetro identificado, não há nenhuma maneira de referenciar o objeto exceção dentro do handler de exceção.

Observação de engenharia de software 16.10

A hierarquia `exception` padrão é um bom ponto de partida para criar exceções. Os programadores podem construir programas que podem lançar exceções-padrão, lançar exceções derivadas das exceções-padrão ou lançar suas próprias exceções não derivadas das exceções-padrão.

Observação de engenharia de software 16.11

Utilize `catch(...)` para realizar uma recuperação que não depende do tipo de exceção (por exemplo, liberar recursos comuns). A exceção pode ser relançada para alertar handlers `catch` envolventes mais específicos.

16.14 Outras técnicas de tratamento de erro

- **Outras técnicas de tratamento de erro**
  - **Ignore a exceção**
    - **Isso é devastador para softwares comerciais e de missão crítica.**
  - **Aborte o programa**
    - **Isso evita que um programa forneça resultados incorretos aos usuários.**
    - **É inapropriado para aplicativos de missão crítica.**
    - **Se conseguir um recurso, o programa deve liberá-lo antes da terminação do programa.**
  - **Configure indicadores de erro**
  - **Emita uma mensagem de erro e passe um código de erro apropriado por meio de `exit` ao ambiente do programa.**

Erro comum de programação 16.10

Abortar um componente de programa em decorrência de uma exceção não interceptada poderia fazer com que um recurso — como um fluxo de arquivo ou um dispositivo de E/S — ficasse indisponível para outros programas. Isso é conhecido como ‘vazamento de recurso’.

16.14 Outras técnicas de tratamento de erro (cont.)

- **Outras técnicas de tratamento de erro (cont.)**
  - **Use as funções `setjump` e `longjump`**
    - **Definida na biblioteca `<csetjmp>`.**
    - **Usada para pular imediatamente de uma chamada de função profundamente aninhada para um handler de erro.**
      - **Desempilhe a pilha sem chamar destrutores de objetos automáticos.**
  - **Use uma capacidade de tratamento de erros dedicada.**
    - **Como a função `new_handler` registrada com `set_new_handler` para o operador `new`.**