



Gerenciamento de erros e depuração de código


Dr. Rodrigo Mologni Gonçalves dos Santos



Conteúdo

1. Introdução
2. Erros de compilação
3. Erros de ligação
4. Erros de execução
5. Exceções
6. Erros de lógica
7. Depuração

Introdução

- 
- Erros são simplesmente **inevitáveis** no processo de desenvolvimento de programas. Apesar de que, idealmente, o programa deva ser **livre de erros** ou, ao menos, livre de erros **inaceitáveis**.



Classificação de erros

- **Erros de compilação:**
Erros encontrados pelo compilador.
- **Erros de ligação:**
Erros encontrados pelo ligador ao tentar combinar arquivos objeto para criar um programa executável.
- **Erros de execução:**
Erros encontrados por verificações feitas em um programa em execução.
- **Erros de lógica:**
Erros encontrados pelo programador ao procurar as causas de resultados errôneos.



Requisitos de um programa comum


1. Deve produzir os resultados desejados para todas as entradas válidas.
2. Deve gerar mensagens de erro razoáveis para todas as entradas inválidas.
3. Não deve se preocupar com o mau funcionamento do *hardware*.
4. Não deve se preocupar com o mau funcionamento do *software* do sistema.
5. Pode terminar após encontrar um erro.



Como produzir programas confiáveis?

1. Organizar o *software* para minimizar erros.
2. Eliminar a maior parte dos erros cometidos através de **depuração** e **testes**.
3. Assegurar que os erros remanescentes não sejam sérios.

Erros de compilação

- 
- O **compilador** é a primeira linha de defesa contra erros. Antes de gerar o código-objeto, ele analisa o código-fonte para detectar **erros de sintaxe** e **erros de tipo**.
 - O código-fonte **não** será compilado até que os esses erros sejam corrigidos.

```
int area(int comprimento, int largura); // calcula a área de um retângulo
```



Erros de sintaxe

- Erros de sintaxe são gerados quando o código-fonte **não** está **formatado** de acordo com a gramática de C++.
- Erros de sintaxe tendem a ser **triviais**, porém a mensagem de erro pode ser **enigmática**. Isto porque o problema precisa ler as linhas a frente do problema para identificá-lo.

```
int s1 = area(7; // erro: falta )
int s2 = area(7) // erro: falta ;
Int s3 = area(7); // Int não é um tipo
int s4 = area('7); // erro: falta o caractere de terminação (')
```



Erros de tipo

- Removido os erros de sintaxe, o compilador vai começar a detectar **erros de tipo**; isto é, ele vai reclamar de usos incorretos dos tipos em variáveis, funções, valores etc.

```
int x0 = arena(7); // erro: função não declarada
int x1 = area(7); // erro: número incorretos de argumentos
int x2 = area("sete", 2); // erro: 1º argumento com tipo errado
```




Não erros

- Um bom compilador pode alertar o programador de possíveis problemas, mas não necessariamente emitirá um erro.

```
int x4 = area(10, -7); // OK: mas o que é um retângulo com largura -7?  
int x5 = area(10.7, 9.3); // OK: mas chama area(10,9)  
char x6 = area(100, 9999); // OK: mas trunca o resultado
```

Erros de ligação

- 
- Um programa consiste em diversas partes computadas separadamente, chamadas de **unidades de tradução**. Cada função em um programa deve ser declarada exatamente com o **mesmo nome e tipo** cada unidade de tradução. Arquivos de cabeçalho são usados para assegurar isto.

```
int area(int comprimento, int largura); // onde está a definição de area?
```

```
int main()  
{  
    int x = area(2,3);  
}
```

Erros de execução



- Se um programa **não** tem **erros de compilação** e **não** tem **erros de ligação**, então ele poderá ser executado.
- O que fazer com o erro capturado durante a execução?


```
int area(int comprimento, int largura) // calcula a área de um retângulo
{
    return comprimento * largura;
}

int area_quadro(int x, int y) // calcula a área dentro de um quadro
{
    return area(x - 2, y - 2);
}

int main()
{
    int x = -1;
    int y = 2;
    int z = 4;
    // ...
    int area1 = area(x, y); // produz valor negativo
    int area2 = area_quadro(1, z); // produz valor negativo
    int area3 = area_quadro(y, z); // produz valor zero
    double razao = double(area1) / area3; // divisão por zero (erro detec. por hardware)
}
```



Soluções

- Deixar o chamador de `area()` lidar com os argumentos incorretos.
- Deixar `area()` (a função chamada) lidar com os argumentos incorretos.



O chamador trata os erros

```
if (x <= 0 || y <= 0)
    error("argumento de area() não positivo");
int area1 = area(x, y);

if (z <= 2)
    error("argumento não positivo de area() chamada por area_quadro()");
int area2 = area_quadro(1, z);

if (y <= 2 || z <= 2)
    error("argumento não positivo de area() chamada por area_quadro()");
int area3 = area_quadro(y, z);
```




O chamado trata os erros

```
int area(int comprimento, int largura)
{
    if (comprimento <= 0 || largura <= 0)
        error("argumento não positivo de
              area()");
    return comprimento * largura;
}
```

```
int area_quadro(int x, int y)
{
    const int largura = 2;
    if (x - largura <= 0 || y - largura <= 0)
        error("argumento não positivo de
              area() chamada por
              area_quadro()");
    return area(x - largura, y - largura);
}
```

Exceções

- 
- C++ fornece um mecanismo para ajudar a lidar com erros: as **exceções**.
 - A ideia fundamental é separar a **detecção** de um erro (que deve ser feita na função chamada) do **tratamento** do erro (que deve ser feito na função chamadora) e, ao mesmo tempo, assegurar que um erro detectado não possa ser ignorado.
 - Vocês aprenderão a tratar exceções mais tarde, na aula **3.6 – Tratamento de exceções**.

Erros de lógica



- **Erros de lógica** são geralmente os mais difíceis de encontrar e eliminar, porque neste estágio o computador faz o que o programa pediu para ele fazer.


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

int main()
{
    double temp = 0;
    double soma = 0;
    double temp_alta = 0;
    double temp_baixa = 0;
    int num_temps = 0;

    while (cin >> temp)
    {
        ++num_temps;
        soma += temp;
        if (temp > temp_alta) temp_alta = temp;
        if (temp < temp_baixa) temp_baixa = temp;
    }

    cout << "Maior temperatura: " << temp_alta << endl;
    cout << "Menor temperatura: " << temp_baixa << endl;
    cout << "Temperatura média: " << soma / num_temps << endl;
}
```

```
-16.5 -23.2 -24.0 -25.7 -26.1 -18.6 -9.7 -2.4  
  7.5  12.6  23.8  25.3  28.0  34.8  36.7  41.5  
 40.3  42.6  39.7  35.4  12.6   6.5  -3.7 -14.3
```

^Z

Maior temperatura: 42.6

Menor temperatura: -26.1

Temperatura média: 9.29583

```
76.5 73.5 71.0 73.6 70.1 73.5 77.6 85.3  
88.5 91.7 95.9 99.2 98.2 100.6 106.3 112.4  
110.2 103.6 94.9 91.7 88.4 85.2 85.4 87.7
```

```
^Z
```

```
Maior temperatura: 112.4
```

```
Menor temperatura: 0.0
```

```
Temperatura média: 89.2083
```

```

#include <iostream>
using namespace std;

int main()
{
    double temp = 0;
    double soma = 0;
    double temp_alta = -1000; // inicializar com impossível baixo
    double temp_baixa = 1000; // inicializar com impossível alto
    int num_temps = 0;

    while (cin >> temp)
    {
        ++num_temps;
        soma += temp;
        if (temp > temp_alta) temp_alta = temp;
        if (temp < temp_baixa) temp_baixa = temp;
    }

    cout << "Maior temperatura: " << temp_alta << endl;
    cout << "Menor temperatura: " << temp_baixa << endl;
    cout << "Temperatura média: " << soma / num_temps << endl;
}

```

Depuração

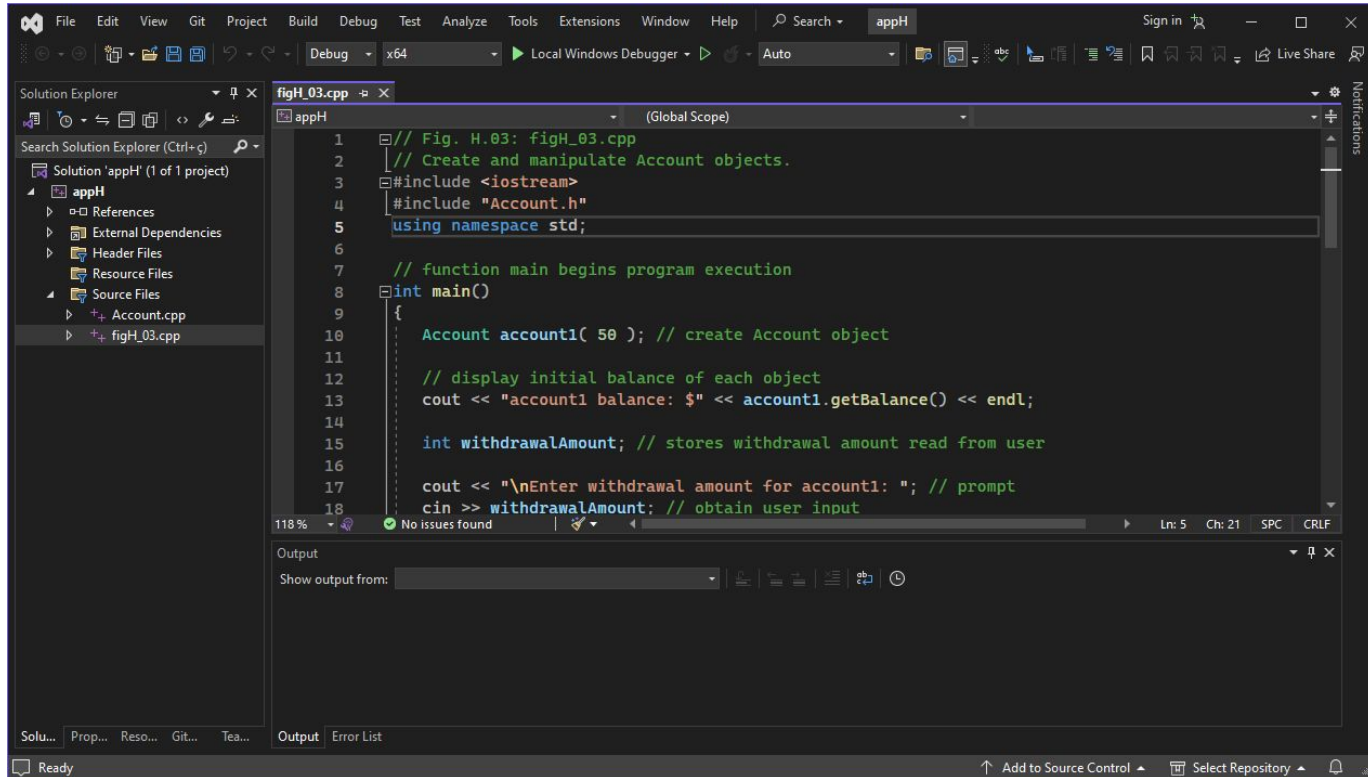


- Os **erros de lógica** (também chamado de *bugs*) não impedem que um programa compile com sucesso, mas fazem com que o programa produza resultados errôneos ao executar.
- A maioria dos fornecedores de compilador C++ oferece um *software* chamado **depurador**, que permite monitorar a execução de seus programas para localizar e remover erros de lógica.



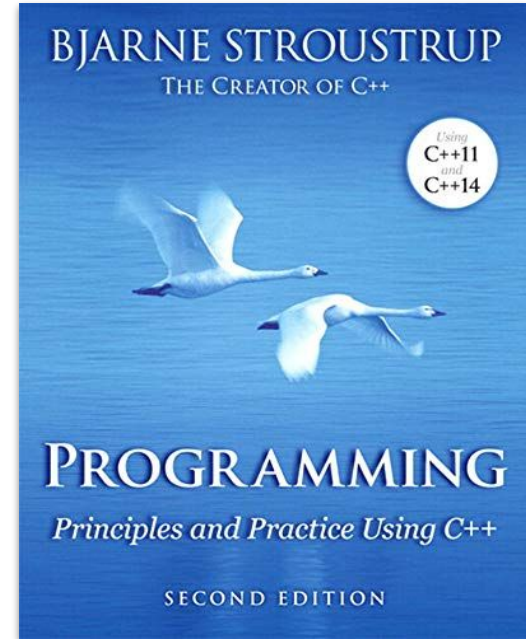
Utilizando o depurador do Visual Studio

1. Pontos de interrupção e o comando *Continue*
2. As janelas *Locals* e *Watch*
3. Controlando a execução utilizando os comandos *Step Into*, *Step Over*, *Step Out* e *Continue*
4. A janela *Autos*



Referências

Chapter 5: Errors



Chapter H: Using the Visual Studio Debugger

