

Programação e Desenvolvimento de Software 2

Estratégias de depuração e Ferramentas

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

Introdução

- Boas práticas de programação
 - Reduzem a chance de erros (eles vão existir!)
 - Medidas **proativas**
 - Programação defensiva
 - Testes de unidade
- Meu programa não funciona! E agora?!
 - ~~▪ Vou reescrever tudo do zero!~~
 - Medidas **reativas**
 - Depuração

Depuração

Motivação

- Depurar grandes programas é difícil (uma arte?)
 - Resolução de problemas → Antes, durante e depois!
- Um **bom** programador
 - Deve saber uma ampla variedade de estratégias de depuração
 - Deve conhecer/usar ferramentas que facilitam a depuração
 - Debuggers → Ajudam a acompanhar/analisar a execução
 - Sistemas de controle de versão → Ajudam a verificar as mudanças
 - IDEs → Ajudam a integrar e organizar as ferramentas
 - Editor, Compilador, Depurador, Modelagem, Controle de versão, Testes, ...

The Secret Art of Debugging: <https://dev.to/dotnet/the-secret-art-of-debugging-1lfi>



PDS 2 - Estratégias de depuração e Ferramentas

3

Depuração vs. Revisão

- Revisão de código → Verificação e validação
 - Etapas relacionadas ao estabelecimento da **existência** de inconsistências (falhas) em um determinado programa
- Depuração (debugging)
 - Relacionado à **localização** e **reparação** das falhas (bugs)



Primeiro *bug* na computação (Harvard Mark II)



PDS 2 - Estratégias de depuração e Ferramentas

4

Depuração vs. Testes

- Depurar
 - O que fazer para tentar **consertar** o programa?
- Testar
 - O que fazer para tentar **quebrar** o programa?



Depuração



Quando fazer?

- Código dá *crash*, parando a execução
- Resultados diferentes dos esperados
- Necessidade de melhorar o desempenho
- Entender melhor como o código funciona

Depuração

Tipos de erros

▪ Sintáticos

- Erros associados ao fato de que a **sintaxe** específica daquela linguagem não está sendo respeitada

▪ Semânticos

- Erros associados à **utilização** indevida de algumas declarações do programa (a sintaxe pode estar correta!)

▪ Lógica

- Erros associados ao fato de que o **comportamento** desejado (especificação do programa) não está sendo alcançado



Depuração

Tipos de erros – Sintáticos

```
int main() {  
  
    int a = 5  
  
    return 0;  
}
```

- Falta adicionar ';' ao final.

```
int main() {  
  
    int x = (3 + 5;  
  
    return 0;  
}
```

- Falta fechar o parênteses.

Depuração

Tipos de erros – Sintáticos

▪ Leia e entenda as mensagens de erro!

```
main.cpp: In function 'int main()':
main.cpp:15:3: error: expected ';' or ';' before 'return'
    return 0;
    ^
```

O compilador
é seu amigo!

- **-Wall:** Exibe na tela todos os warnings que ele encontrar no código. Um warning não é um erro, mas sim uma advertência sobre o uso incorreto (não recomendado) de alguma função/instrução da linguagem.
- **-pedantic:** Essa flag faz com que o compilador seja mais “pedante”, emitindo warnings para todas as partes do código que podem estar com algum problema.

Depuração

Tipos de erros – Semânticos

```
int main() {

    int i;
    i++;

    return 0;
}
```

- Variável não inicializada.
- Ocorre apenas um Warning.

```
int main() {

    int a = "hello";

    return 0;
}
```

- Atribuição incorreta de tipo.
- Erro de compilação.

Depuração

Tipos de erros – Momento da detecção

- Tempo de **compilação**
 - Erros de sintaxe e erros semânticos estáticos geralmente são indicados pelo próprio compilador
- Tempo de **execução**
 - Erros semânticos dinâmicos e erros lógicos não podem ser detectados pelo compilador (muito difícil)
 - É necessário depurar o código!

Depuração

Tipos de erros – Momento da detecção

```
int main() {
    int a, b, x;
    a = 10;
    b = 0;
    x = a / b;

    return 0;
}
```

**Erro semântico
e de lógica!**

O programa compila, e o problema só será detectado durante a execução do programa.

Depuração

Tipos de erros – Lógica

- Geralmente observados durante execução
 - Retorno de um resultado incorreto
 - Loops infinitos
 - *Segmentation fault*
- Podem ser bastante imprevisíveis
 - De acordo com entradas (bem) específicas
 - Dependentes de plataforma ou hardware

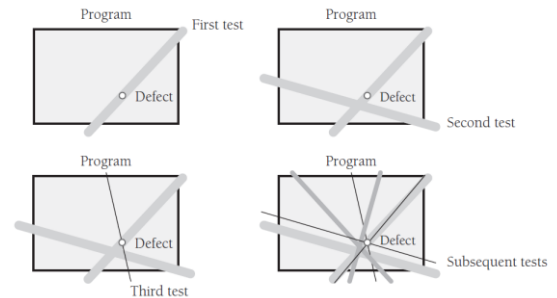
Depuração

Procedimento

- **Reproduzir** o problema
 - Determinar as condições / Estabilizar a execução
- **Identificar** o local e provável causa
 - Dados → Hipótese → Experimentos (repetir)
- **Alterar** o código para a correção do erro
 - Considerar o real motivo do problema (causa → sintoma)
- **Avaliar** a solução e procurar erros similares
 - Cuidado com possíveis efeitos colaterais!

Depuração Procedimento

- Os erros geralmente surgem pela combinação de diferentes fatores
- Reproduzir um erro de várias maneiras distintas pode ajudar a diagnosticar a causa do erro!



“Pense nessa abordagem como uma triangulação do defeito.”

Fonte: Code Complete



PDS 2 - Estratégias de depuração e Ferramentas

15

Como **NÃO** fazer depuração

- Tentar encontrar defeitos adivinhando (na sorte)
- Fazer alterações aleatórias até funcionar
- Não manter um histórico das alterações feitas
- Corrigir o erro com a solução mais simples e rápida
 - Não identificar a origem/razão do problema
 - O sintoma é removido, mas não a causa
 - O erro “sumiu”, o problema está resolvido!

When debugging, novices insert corrective code; experts remove defective code.

– R. Pattis



PDS 2 - Estratégias de depuração e Ferramentas

16

Depuração

Dicas gerais

- A melhor estratégia é tentar evitar bugs
 - Faça uma modelagem antes de implementar
 - Siga as boas práticas de programação
 - Pense no futuro e não apenas no presente
 - Considere a facilidade de manutenção e a legibilidade
 - Maximize a modularidade e a reutilização de código



Depuração

Estratégia raiz

- printf / cout
 - “Acompanhamento” da execução do programa
 - Selecionar alguns pontos chave no programa e exibir os valores de variáveis importantes que deveriam (ou não) ter sido modificadas
- **Prós**
 - Simples e fácil (rápido?)
 - Você já “sabe fazer”
- **Contras**
 - Pode prejudicar a legibilidade do código
 - Várias compilações para diferentes cenários
 - Não é possível pausar ou controlar a execução
 - Não é possível modificar valores de variáveis

GDB

- **GNU Debugger**
 - Depurador “padrão” de C/C++
- Permite acompanhar o que está acontecendo dentro do programa (estado) enquanto ele está sendo executado
 - Analisar o código uma linha de cada vez
 - Adicionar pontos específicos de parada (*breakpoints*)
 - Verificar/alterar valores de variáveis durante a execução

<https://www.gnu.org/software/gdb/>

<https://betterexplained.com/articles/debugging-with-gdb/>



PDS 2 - Estratégias de depuração e Ferramentas

19

GDB Comandos

- Compilar com flag “-g” (debug)
- Acessível pela linha de comando
- Ferramentas auxiliares
 - <https://www.gnu.org/software/ddd/>
 - <https://gdbgui.com>
 - <https://www.onlinegdb.com/>

GDB cheatsheet - page 1			
Running	<where>	next	Go to next instruction (source line) but don't step into functions.
# gdb -<program> [core dump]	function_name	finish	Continue until the current function returns.
Start GDB with optional core dump.	line_number	continue	Continue normal execution.
# gdb --args -<program> -<args...>	file:line_number	Variables and memory	
# gdb --pid <pid>		print/<format> <var>	Print content of variable/memory location/register.
Start GDB and pass arguments		display/<format> <var>	Like 'print', but print the information after each stepping instruction.
set args <args...>	Conditions	undisplay <display>	Remove the 'display' with the given number.
Set arguments to pass to program to be debugged.	break/watch <where> [if <condition>]	enable display <display>	Enable the 'display' with the given number.
run	Break at the given location if the condition is met.	disable display <display>	Disable the 'display' with the given number.
kill	Conditions may be almost any C expression that evaluate to true or false.	x/<fmt> <address>	Print memory.
Breakpoints	condition <breakpoint> <condition>	n How many units to print (default 1).	
break <where>	Set/breakpoint. Set/breakpoint. Set/breakpoint.	f Format character (like 'gint').	
delete <breakpoint>	Examining the stack	u Unit.	Unit is one of:
Remove a breakpoint.	backtrace		b Byte
clear	where		h Half word (two bytes)
Delete all breakpoints.	Show call stack.		w Word (four bytes)
enable <breakpoint>	backtrace full		g Giant word (eight bytes).
Enable a disabled breakpoint.	where full		
disable <breakpoint>	Show call stack, also print the local variables in each frame.		
Disable a breakpoint.	frame <frame>		
Watchpoints	Select the stack frame to operate on.		
watch <var>	step		
Set a new watchpoint.	Go to next instruction (source line), diving into function.		
delete/enable/disable <watchpoint>			
Like breakpoints.			

<https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf>



PDS 2 - Estratégias de depuração e Ferramentas

20

GDB

Comandos

- **Breakpoint:** ponto de interrupção colocado em uma linha específica e que o debugger usará para pausar a execução
- **Step:** usado para continuar a execução de diferentes formas
 - Step-into: vai para próxima instrução acessando subrotinas (step)
 - Step-over: próxima instrução sem entrar em subrotinas (next)
 - Step-out: executar até o final da subrotina (finish)
 - Continue: prossegue a execução até próximo ponto de parada
- **Display:** exibir a informação de uma determinada variável

GDB

Exemplo 1

```
#include<iostream>

using namespace std;

int findMax(int *array, int len) {
    int max = -1;
    for (int i=1; i <= len; i++) {
        if (max < array[i]) {
            max = array[i];
        }
    }

    return 0;
}
```


```
int main() {
    int arr[5] = {60, 17, 21, 44, 2};

    int max = findMax(arr, 5);
    cout << "Valor maximo e: " << max << endl;

    return 0;
}
```

[OnlineGDB](#)

```
$ g++ -std=c++11 -g gdb_ex01.cpp -o gdb_ex01
$ gdb gdb_ex01
```



GDB

Exemplo 2

- Calcular o valor da seguinte série

- Entradas: x e n

$$\frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!}$$

- Por onde começar a implementação?

- Quantas operações estão envolvidas?
- Modularizar em 2 funções → Fatorial e Serie

GDB

Exemplo 2

```
#include <iostream>
#include <cmath>

using namespace std;

int fatorial(int n) {
    int fat = 1;
    for (int i = 1; i <= n; i++)
        fat = fat * i;
    return fat;
}

double series(double x, int n) {
    double xpow, seriesValue;

    for (int k = 1; k <= n; k++) {
        xpow = pow(x, k);
        seriesValue += xpow / fatorial(k);
    }

    return seriesValue;
}
```

- Esse código funciona?

```
int main() {
    cout << series(2, 3) << endl;
    return 0;
}
```

[OnlineGDB](#)

- Como descobrir o erro?

- fatorial(int n)
 - int fat = 1;
 - for (int i = 1; i <= num; i++)
- series(double x, int n)
 - for (int k = 0; k <= n; k++)
 - double seriesValue = 0.0;

Valgrind

- Valgrind → Conjunto de ferramentas (instrumentação)
- Memcheck
 - Utilitário para detectar erros no gerenciamento da **memória**
 - Memory leak, erros de alocação ou desalocação, ...
- Programas auxiliares/alternativos
 - Valkyrie: <http://valgrind.org/downloads/guis.html>
 - Dr. Memory: <http://drmemory.org/>

<http://valgrind.org/docs/manual/quick-start.html>



PDS 2 - Estratégias de depuração e Ferramentas

25

Valgrind Comandos

- Compilar com flag “-g” (debug)
- Parâmetro ao chamar o utilitário
 - `--leak-check=full`
 - Fornece informações detalhadas em vez de apenas um resumo
- Acesse o manual para entender o relatório de saída!

<https://www.valgrind.org/docs/manual/mc-manual.html>



PDS 2 - Estratégias de depuração e Ferramentas

26

Valgrind

Exemplo 1

```
#include<iostream>

using namespace std;

int main() {
    int a[10];
    for (int i = 0; i < 9; i++)
        a[i] = i;

    for (int i = 0; i < 10; i++){
        cout << a[i] << endl;
    }

    return 0;
}
```

```
$ g++ -std=c++11 -g valgrind_ex01.cpp -o valgrind_ex01
$ valgrind --leak-check=full ./valgrind_ex01
```

<http://valgrind.org/docs/manual/mc-manual.html#mc-manual.uninitvals>



PDS 2 - Estratégias de depuração e Ferramentas

27

Valgrind

Exemplo 2

```
#include<iostream>

using namespace std;

struct TADExemplo {
    int atributo;
};

int main() {
    TADExemplo *c = new TADExemplo();
    c->atributo = 10;

    cout << c->atributo << endl;
    delete c;

    c->atributo = 99;
    cout << c->atributo << endl;

    // delete c;

    return 0;
}
```

https://en.wikipedia.org/wiki/Undefined_behavior

<http://valgrind.org/docs/manual/mc-manual.html#mc-manual.badrw>



PDS 2 - Estratégias de depuração e Ferramentas

28

Valgrind Exemplo 3

```
#include<iostream>

using namespace std;

struct TADExemplo {
    int atributo;
};

int main() {

    TADExemplo* vetor[10];

    vetor[0] = new TADExemplo();
    vetor[1] = new TADExemplo();

    return 0;
}
```

<http://valgrind.org/docs/manual/mc-manual.html#mc-manual.leaks>



PDS 2 - Estratégias de depuração e Ferramentas

29

Considerações finais

- Procure por problemas comuns
 - Dividir para conquistar
 - Verifique o valor de variáveis importantes
 - Concentre-se em mudanças recentes
- Utilize ferramentas auxiliares
 - Debuggers → GDB, Valgrind

The 5 Stages of Debugging

At some point in each of our lives, we must face errors in our code. Debugging is a natural healing process to help us through these times. It is important to recognize these common stages and realize that debugging will eventually come to an end.



Denial

This stage is often characterized by such phrases as "What? That's impossible," or "I know this is right." A strong sign of denial is recompiling without changing any code, "just in case."



Bargaining/Self-Blame

Several programming errors are uncovered and the programmer feels stupid and guilty for having made them. Bargaining is common: "If I fix this, will you please compile?" Also, "I only have 14 errors to go!"



Anger

Cryptic error messages send the programmer into a rage. This stage is accompanied by an hours-long and profanity-filled diatribe about the limitations of the language directed at whomever will listen.



Depression

Following the outburst, the programmer becomes aware that hours have gone by unproductively and there is still no solution in sight. The programmer becomes listless. Posture often deteriorates.



Acceptance

The programmer finally accepts the situation, declares the bug a "feature", and goes to play some Quake.



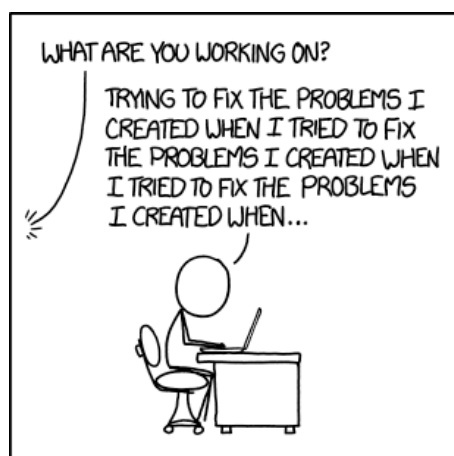
PDS 2 - Estratégias de depuração e Ferramentas

30

Considerações finais

- Correto gerenciamento da memória é fundamental!
- Quando utilizar o Valgrind?
 - Se está ocorrendo algum erro/comportamento estranho
 - Após uma alteração muito grande ter sido feita
 - Antes de uma versão final ser lançada
 - Testes automatizados → Verificação de rotina
- Valgrind possui outras funcionalidades → Profiler

Considerações finais



<https://xkcd.com/1739/>