Depuração & Testes

Algoritmos e Programação de Computadores

Guilherme N. Ramos
gnramos@unb.br

2018/2

Introdução

Origens de erros: especificação, algoritmo, codificação.

"*Se depurar é o processo de remover bugs, então programar deve ser o processo de inserí-los.*"

Edsger W. Dijkstra

Introdução

"*Há duas maneiras de se escrever programas sem erros; apenas a terceira funciona.*"

Alan Perlis

Erros de programação *sempre* existirão [pelo menos enquanto o processo de geração de código for o que conhecemos]...

O desenvolvimento de código [bem feito] segue etapas fases:

1. entendimento/análise do problema
2. elaboração de um algoritmo
3. implementação do algoritmo
4. *depuração*
5. *testes*

Depuração

Depuração

Feita quando se sabe que o programa não funciona (erros de execução, de segmentação de memória), não tem o desempenho desejado, ou simplesmente não tem o comportamento esperado.

Uma das melhores práticas de programação é *realizar pequenas alterações no código e testá-las adequadamente a medida que são feitas.*

Depuração

1. *Teste* o código para descobrir quais problemas existem.
2. *Defina* as condições que o erro pode ser reproduzido.
3. *Encontre* onde no código está a instrução que causa o erro.
4. *Corrija* a instrução;
5. *Verifique* que a correção funciona (com testes).

Depuração

00-iniciante.c

```c
int main() {
  int n;

  printf("Digite um número: ");
  scanf("%d", n);
  printf("Você digitou: %d", n);

  return 0;
}
```


Depuração

01-iniciante.c

```c
int main() {
  const float PI = 3.141569;
  int r = 10;

  float area = PI*r*r;

  printf("A área de um círculo de raio %d é %f.\n", area, r);

  return 0;
}
```


Depuração

02-iniciante.c

```c
#include <stdio.h>

int main() {
  int a, b;

  printf("Digite um número inteiro:");
  scanf("%d", &a);

  printf("Digite outro número inteiro:");
  scanf("%d", &b);

  if(a != b)
    printf("São diferentes, tudo bem.\n");
  else
    printf("São iguais, consertando...\n");
    ++b;

  return 0;
}
```

Depuração

03-iniciante.c

```
1 /* Especificação de algum comportamento que lida com
2 argumentos da linha de comando (ex: o comando gcc).
3
4 Exemplo de uso para depuração (supondo que este programa seja
5 o executável "03-iniciante"):
6
7 ./03-iniciante -o meu_executavel -f123 -t500 01-iniciante.c
8
9 O resultado esperado é:
10
11 executavel [meu_executavel]
12     inicio [123]
13        fim [500]
14 */
```


Depuração

A *depuração* é inevitável... Há diferentes formas de avaliar a execução:

- *pensar* a respeito;
- o bom e velho `printf`;
- busca binária;
- depuradores;
- etc.

Depuração

Antes de consertar um *bug*, é preciso encontrá-lo:

- ao manipular qual variável?
- ao chamar qual função?
- em que linha?

O depurador é um programa que facilita este processo!

`http://pythontutor.com/visualize.html`

Depuração

05-busca_binaria

Dados os arquivos busca_binaria.c e busca_binaria.py, implemente a função `bb` e os testes indicados de modo que o arquivo possa ser executado sem erros.

Depuração

busca_binaria_it.py

```
1 def busca_binaria(lista, valor):
2     inf = 0
3     sup = len(lista) - 1
4     while inf <= sup:
5         meio = (inf + sup) // 2
6         if lista[meio] > valor:
7             sup = meio - 1
8         elif lista[meio] < valor:
9             inf = meio + 1
10        else:
11            return meio
12    return -1
```


Depuração

busca_binaria_re.py

```
1 def busca_binaria(lista, valor):
2   if not lista:
3     return -1
4   inf = 0
5   sup = len(lista) - 1
6   meio = (inf + sup) // 2
7   if lista[meio] > valor:
8     return busca_binaria(lista[:meio], valor)
9   elif lista[meio] < valor:
10    return meio + 1 + busca_binaria(lista[meio + 1:], valor)
11  else:
12    return meio
```


Depuração

"*Depure agora, não depois.*"

Brian W. Kernighan & Rob Pike

The GNU Project Debugger

01-depurador/gdb.c

```
1 int divisao(int x, int y) {
2   return (x / y);
3 }
4
5 int main() {
6   int x = 5;
7   int y = 2;
8   printf("%d/%d = %d\n", x, y, divisao(x, y));
9   x = 5;
10  y = 0;
11  printf("%d/%d = %d\n", x, y, divisao(x, y));
12
13  return 0;
14 }
```

The GNU Project Debugger

"Tradicional"

```
Segmentation fault (core dumped)
```

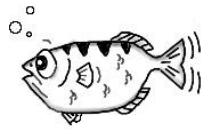
"Depurável"

```
Program received signal SIGFPE, Arithmetic exception.
0x0000000000400534 in divisao (x=3, y=0) at gdb.c:13
13 return (x / y);
```


gdb

The GNU Project Debugger

Permite que se veja o que ocorre "dentro" de um programa durante sua execução – ou o que o programa estava fazendo até o momento que falhou.



O gdb oferece várias facilidades para a depuração de programas [compilados com o gcc], permitindo:

1. iniciar o programa especificando qualquer coisa que possa afetar seu comportamento;
2. interromper o programa conforme condições específicas;
3. examinar o que aconteceu (quando o programa for interrompido);
4. alterar coisas no programa (para avaliar os efeitos).

gdb

gdb é um depurador para diversas linguagens de programação.[1]

- gera informação para depuração [conforme o sistema operacional] para o gdb (pode funcionar com outros depuradores. ou não)
- gdb aceita otimização (-O), mas lembre-se que a isso é coisa do ~~tinhoso~~ compilador.
- http://www.gnu.org/software/gdb/gdb.html

Depuração:

```
$ gcc [flags] -g <arquivo> -o <saída>
```

[1] C, C++, D, Go, Objective-C, OpenCL, Fortran, Pascal, Modula-2, Ada

gdb

gdb tem uma interface interativa (com histórico, *auto-complete*, etc.)

help	é inestimável...
file	define o arquivo [executável, compilado com a opção -g] a ser depurado
run	executa o programa [em depuração]
kill	finaliza a execução do programa
break	interrompe a execução na linha ou função especificada
print	imprime o resultado da expressão
step/next	avança a execução (passo a passo)
continue	continua a execução
watch	interrompe a execução quando o valor da expressão muda
set	"avalia expressão e atribui variável"
backtrace	mostra o traço de cada elemento na pilha de execução
quit	termina o gdb

gdb

04-intermediario.c

```
1 int main() {
2     int x = 2;
3
4     soma_tres(&x);
5     printf("x = %d\n", x);
6
7     return 0;
8 }
```


Valgrind

http://valgrind.org/

Software livre (GPL2) para depuração. É, na verdade, uma máquina virtual que possibilita a análise dinâmica (*checker*/*profiler*) da execução de programa.

Testes

Por que testar programas?

- seres humanos cometem erros
- programas contêm erros: sintaxe / lógica
- software *robusto* deve conter o mínimo possível de erros
- bugs podem causar desconfortos e catástrofes

Testes

"*Testes de programas podem ser usados para revelar a existência de erros, mas nunca para mostrar sua ausência!*"

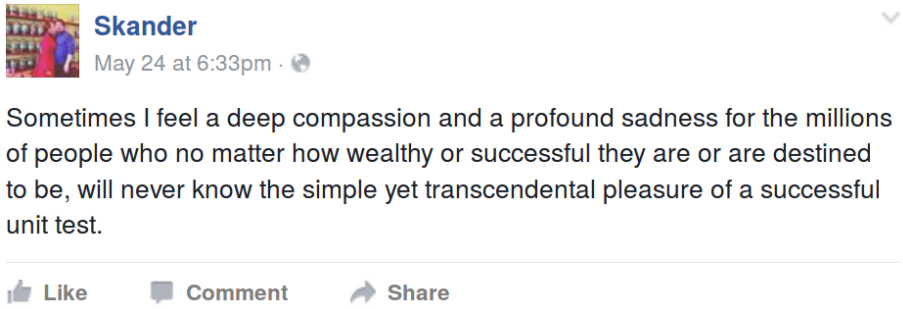
Edsger W. Dijkstra

Testes buscam investigar a qualidade do programa no contexto em que ele deve operar.

"*Em um típico projeto de programação, 50% do tempo e mais de 50% do custo total são gastos em testes do programa ou sistema em desenvolvimento.*"

Myers, Badgett & Sandler

Testes



Testes

Idealmente, toda possível execução do programa deveria ser testada, mas como isso é inviável, a qualidade do testes depende da qualidade dos profissionais que definem *o que testar*.

Origem de erros?

- especificação incompleta, errada ou impossível;
- falha(s) na implementação.

Testes
Como testar?

- Teste manuais
  - cansativos
  - testam apenas alguns casos
  - testam apenas algumas vezes
- Testes automatizados
  - bateria de testes que cobre o máximo possível do seu código
  - executada rotineiramente várias vezes por dia

Testes
Como testar?

- Pense cuidadosamente nos casos em que seu programa pode falhar
- Pense nos diferentes tipos de entrada que exercitam caminhos diferentes no seu programa
- Pense nos casos diferentes do seu código
- Escreva teste automatizados para todos esses casos

Testes de Caixas

Teste de Caixa-Preta

Testar a funcionalidade do programa sem analisar a implementação: análise de pares entrada/saída. Quanto mais abrangentes as entradas, em função das especificações, melhor a qualidade do teste.

```
1 assert(min(1, 2, 3) == 1);
2 assert(min(1, 3, 2) == 1);
3 assert(min(2, 1, 3) == 1);
4 assert(min(2, 3, 1) == 1);
5 assert(min(3, 1, 2) == 1);
6 assert(min(3, 2, 1) == 1);
```

```
1 assert(min(1, 1, 1) == 1);
2 assert(min(1, 1, 2) == 1);
3 assert(min(1, 2, 2) == 1);
4 assert(min(-1, 2, 3) == -1);
5 assert(min(1, -2, 3) == -2);
6 assert(min(1, 2, -3) == -3);
```


Testes de Caixas

Teste de Caixa-Branca

Testar a implementação do sistema: análise de fluxo (processos, decisões e condições).

```
1 /* Implementação */
2 if(A > 1 && B == 0)
3     x /= A;
4 if(A == 2 || x > 1)
5     ++x;
```

```
1 A=2, B=0; /* todos os processos */
2
3 A=3, B=0, X=0; /* todas decisões e */
4 A=2, B=1, X=1; /* todas as condições */
```


Test Driven Development