A Arte e a Ciência da Depuração

MAC211 - Laboratório de Programação

Fabio Kon

Departamento de Ciência da Computação

maio de 2001

(última atualização: 21/6/2002)

O Termo *Bug*

- Já existia antes da computação.
- Primeiro *bug* computacional era um *bug* mesmo!
- Sempre existiu e sempre existirá.
- Temos que aprender como lidar com eles e minimizá-los.
- Dijkstra:
  - *Não são "bugs" são "erros de programação"!*

Fatores que Levam a Erros de Programação

- Fatores Humanos:
  - Inexperiência, falta de concentração, cansaço, erros normais (errar é humano).
- Fatores Tecnológicos:
  - Linguagem de programação,
  - Ferramentas,
  - Complexidade e tamanho do software desenvolvido.

Técnicas para Garantir a Integridade de Software

- Provas formais da corretude de programas.
- Modelagem cuidadosa do problema.
- Análise dos requisitos.
- Verificação formal do que um programa faz.
- Mas isso não muda como os programas são feitos.
- Só funcionam para programas pequenos :-(

Erros Sempre Vão Existir

- Parece que não há como fugir disso (pelo menos com a tecnologia das próximas décadas).
- Solução:
  - **Testes** para descobrir os erros.
  - **Depuração** para localizar e eliminar os erros.

Dificuldades

- Depurar programas é difícil e exige muito tempo.
- Nosso objetivo deve ser evitar ter que depurar muito.
- Como fazer isso?
  - Escrevendo código de boa qualidade.
  - Estudando (e aplicando) técnicas que evitem erros.

Como Evitar Erros

- Bom projeto (*design*).
- Bom estilo.
- Limitar informações globais (e.g. variáveis globais).
- Interfaces cuidadosamente planejadas.
- Limitar as interações entre os módulos apenas às interfaces.
- Ferramentas automáticas de verificação.

A Influência das Linguagens de Programação

- Linguagem de Montagem
- BASIC, MS-Visual Basic (`goto`, argh!)
- C
- C++
- Pascal
- Java

- verificação de índices em vetores, ponteiros, coleta de lixo automática, verificação de tipos (forte, fraca)

O Que Fazer Então?

- É preciso estar ciente das características perigosas das linguagens com as quais se está lidando.

1. Escrever bom código.

2. Escrever bons testes.

3. Usar boas ferramentas de depuração.

Depuração

- Em geral, mais da metade do tempo gasto no desenvolvimento de software é gasto com depuração.
- Temos que tentar diminuir isso. Como?

1. Escrever bom código.

2. Escrever bons testes.

3. Usar boas ferramentas de depuração.

Depuradores

- Execução passo a passo
  - *step in, step through, run till return*
- *Breakpoints* (linha, função, condição)
- Impressão de valores de variáveis
- Acompanhamento de variáveis
- Seqüência de chamada de funções (*stack trace*)
- *step back* (em algumas linguagens e ambientes)

Depuradores

- Depuradores são uma ferramenta extremamente útil mas às vezes não são a melhor alternativa:
  - algumas linguagens e ambientes não tem;
  - podem variar muito de um ambiente p/ outro;
  - alguns programas, às vezes, não se dão bem com depuradores (SOs, sistemas distribuídos, múltiplos threads, sistemas de tempo real).
- Solução: uso criterioso do `print`.

Depuradores

- Depuradores podem ser muito complicados para iniciantes. Um uso criterioso do `print` pode ser mais fácil.
- Mas vocês não são mais iniciantes!!!
- **Usem o `ddd` do Linux!!!!**
- (ou o gdb no emacs :-)

Quando Há Dicas (*bugs* fáceis)

- Falha de Segmentação (*segmentation fault*) é o melhor caso possível para um erro:
  - basta executar no depurador e olhar o estado da pilha e das variáveis no momento do erro;
  - ou então: arquivo **`core`**
  - **`gdb arq_executavel core`**

Quando Há Dicas (*bugs* fáceis)

- O programa fez algo que não devia ou imprimiu algo absurdo?
- Pare para pensar o que pode ter ocorrido.
- Olhe para a saída do programa: comece do lugar onde a coisa inesperada aconteceu e volte, passo a passo, examinando cada mensagem e tente descobrir onde o erro se originou.
- Pare prá pensar... pense em marcha ré...

Procure por Padrões Familiares de Erros

- Erro comum com iniciantes:

```
int n;
scanf ("%d", n);
```

- Ao invés de:

```
int n;
scanf ("%d", &n);
```


Procure por Padrões Familiares de Erros

```
int n = 1;
double d = PI;
printf("%d\n%f\n", d, n);

1074340347
2687509847584709847584750984 56\
0659874659745693745693654569 37\
93874569387456746592.0000000
```


Procure por Padrões Familiares de Erros

- Uso de **`%f`** ao invés de **`%lf`** para ler **`double`**.
- Esquecer de inicializar variáveis locais:
  - normalmente o resultado é um número muito grande.
  - Lembre de usar **`gcc -Wall ...`**
- Esquecer de inicializar memória alocada com **`malloc()`**:
- valor será lixo também.

Examine a Mudança Mais Recente no Código

- Qual foi a última mudança?
- Se você roda os testes a cada mudança e um teste falha, o que provocou o erro foi a última mudança;
  - ou o *bug* está no código novo,
  - ou o código novo expôs o *bug* de outro lugar.
- Se você não roda testes a cada mudança, veja se o *bug* aparece nas versões anteriores do código.

Gerenciadores de Versões

- **RCS:** excelente ferramenta para gerenciamento de versões de arquivos.
  - Funciona bem com código, artigos, documentação, etc.
  - Pode ser usado por um usuário sozinho ou por um grupo de usuários compartilhando os mesmos arquivos.
- **CVS:** ferramenta ainda mais sofisticada para ambientes distribuídos e para software com muitos diretórios.

Não Faça o Mesmo Erro Duas (ou Três) Vezes

- Quando você corrigi um erro, pergunte a si mesmo se você pode ter feito o mesmo erro em algum outro lugar.
- Em caso afirmativo, vá lá e corrija, não deixe prá mais tarde.
- Procure aprender com os seus erros de modo a não repetí-los.

Não Faça o Mesmo Erro Duas (ou Três) Vezes

```
for (i = 1; i < argc; i++)
  {
      if (argv[i][0] != '-')
            break; // options finished
      switch (argv[i][1])
      {
            case 'o':
                  outname = &argv[i][2];
            case 'f':
                  from = atoi (argv[i]); break;
            case 't':
                  to = atoi (argv[i]); break;
            ...
```


Depure Agora Não Mais Tarde

- Quando um erro aparece pela primeira vez, tente achá-lo imediatamente.
- Não ignore um *crash* agora pois ele pode ser muito mais desastroso mais tarde.
- Mars Pathfinder, julho de 1997.
- Reiniciava todo dia no meio do trabalho.
  - Depuraram, acharam o erro e se lembraram que tinham ignorado esse mesmo erro antes e se esquecido dele.

Tenha Calma Antes de Começar a Mexer no Código

- Erro comum de programadores inexperientes:
  - quando acham um *bug*, começam a mudar o programa aleatoriamente esperando que uma das mudanças vá corrigir o defeito.
  - Não faça isso.
- Pare prá pensar.
- Estude a saída defeituosa.
- Estude o código.

Procurando o Erro

- Respire fundo (bom prá dar uma "ressetada" no cérebro :-).
- Olhe o código um pouco mais.
- Olhe a saída um pouco mais.
- Imprima o pedaço chave do código que pode ser o culpado.
- Não imprima a listagem inteira
  - é mau para o meio-ambiente
  - não vai ajudar pois vai estar obsoleta logo

Se Ainda Não Achou o Erro

- Vá tomar um suco de graviola.
- Volte e experimente mais um pouco.
- Se ainda não funcionou:
  - Provavelmente o que você está vendo não é o que está escrito mas o que você teve a intenção de escrever.

Se Ainda Não Funcionou

- Chame um amigo prá ajudar:
  1. Explique o seu código prá ele.
- Muitas vezes isso já é suficiente.
  2. (mas se 1. não foi suficiente) peça prá ele te ajudar na depuração.
- Quase sempre funciona.

Programação Pareada

- Erro de um detectado imediatamente pelo outro.
  - Leva a uma grande economia de tempo.
- Maior diversidade de idéias, técnicas, algoritmos.
- Enquanto um escreve, o outro pensa em contra-exemplos, problemas de eficiência, etc.

Programação Pareada

- Vergonha de escrever código feio (*hacking*) na frente do seu par.
- Pareamento de acordo com especialidades.
  - Ex.: Videogame Orientado a Objetos.

Quando Não Há Dicas (*bugs* difíceis)

- Não tenho a menor idéia do que está acontecendo!!!!!! Socorro!!!
- Às vezes o *bug* faz o programa não funcionar mas não deixa nenhum indício do que pode estar acontecendo.
- não se desespere...

Torne o Erro Reprodutível

- O pior *bug* é aquele que só aparece de vez em quando.
- Faça com que o erro apareça sempre que você quiser.
  - Construa uma entrada de dados e uma lista de argumentos que leve ao erro.
- Se você não consegue repetir o erro quando você quer, pense no porquê disto.

Tornando o Erro Reprodutível

- Se o programa tem opções de imprimir mensagens de depuração, habilite todas estas opções.
- Se o programa usa números aleatórios com semente saída do relógio, desabilite isso e fixe a semente numa semente que gere o erro.
- É uma boa prática oferecer sempre a opção do usuário entrar com a semente.

Divisão e Conquista

- Julio César; governo da ditadura militar.
- Voltando à depuração.
  - Dá prá dividir a entrada em pedaços menores e encontrar um pedaço bem pequeno que gere o erro? (faça busca binária).
  - Dá prá jogar fora partes do seu programa e ainda observar o mesmo erro.
- Quanto menor for o programa e os dados de entrada, mais fácil será achar o erro.

Numerologia e Tarot

- Em alguns casos, estatísticas ou padrões númericos sobre o erro podem ajudar a localizá-lo.
- Exemplo:
  - Erros de digitação em um texto do Rob Pike.
  - Não estavam no arquivo original (*cut&paste*).
  - Descobriu que ocorriam a cada 1023 chars.
  - Buscou por 1023 no código, buscou por 1024 no código.

Coloque Mensagens de Depuração no Código

```
#define DEBUG 1
#ifdef DEBUG
  #define printDebug(msg) fprintf (stderr, "%s\n", msg)
#elif
  #define printDebug(msg)
#endif

int main (int argc, char **argv)
  {
     printDebug ("Chamando init()");
     init ();
     printDebug ("Voltei do init()");
     ...
```


Escreva Código "Auto-Depurante"

```
void check (char *msg)
{
  if (v1 > v2)
    {
      printf ("%s: v1=%d v2=%d\n", msg, v1, v2);
      fflush (stdout); // Não se esqueça disso!
      Abort ();        // Terminação anormal.
    }
}
...
check ("antes do suspeito");
/* código suspeito */
check ("depois do suspeito");
```

Código "Auto-Depurante"

- Depois de achar o erro, não apague as chamadas ao **`check()`**, coloque-as entre comentários.
- Se as chamadas não fazem com que o programa fique lento, deixe-as lá.

Escreva um Arquivo de Log

- Quando dá pau no programa, o log é a "prova do crime".
- Dê uma olhada rápida no final do log e vá subindo.
- Não imprima.
- Use ferramentas: **`grep, diff, emacs.`**
- Cuidado com o *buffering:*
  - **`setbuf (fp, NULL);`**
  - é default em **`stderr, cerr, System.err`**

Desenhe Gráficos

- Quando a saída é muito extensa, é difícil processá-la a não ser graficamente.
- Exemplo: projeto Protetores do Espaço.
- Ferramenta: **`gnuplot`** (ou, talvez **`xwc`**).

Faça Bom Uso das Facilidades do Ambiente

- Ferramentas: **`grep, diff, emacs, gnuplot, shell scripts, RCS, gcc –Wall, lint, strings`**, etc.
- Escreva programinhas teste:

```
int main (void)
 {
    free (NULL);
    return 0;
 }
```


Mantenha um "Diário de Bordo"

- Se a caça a um erro está sendo muito demorada, vá anotando todas as possibilidades que você está tentando.
- Quando localizar o erro, se for o caso, anote a solução caso precise dela novamente no futuro.

O que Fazer em Último Caso???

- E se tudo isso falha?
- Talvez seja a hora de usar um bom depurador (**`ddd`**) e acompanhar toda a execução do programa passo a passo.
- O seu modelo mental de como o programa funciona pode ser diferente da realidade.

Enganos Comuns

- `if (x & 1 == 0)`
  `func();`
- `float x = 3/2;`
- `while ((c == getchar()) != EOF)`
  `if (c = '\n')`
  `break;`

Enganos Comuns

- `for (i = 0; i < n; i++)`
  `a[i++] = 0;`
- `memset (p, n, 0);`

ao invés de

`memset (p, 0, n);`

Enganos Comuns

```
while(scanf("%s %d", nome, &valor) != EOF)
  {
    p = novoItem (nome, valor);
    lista1 = adicionaComeco (lista1, p);
    lista2 = adicionaFim (lista2, p);
  }
for (p = lista1; p != NULL; p = p->proximo)
  printf("%s %d\n", p->nome, p->valor);
```


Culpando Outros

- Não coloque a culpa em
  - compiladores
  - bibliotecas
  - sistema operacionais
  - hardware
  - vírus
- Infelizmente, a culpa é provavelmente sua.
- A não ser em alguns casos raros....

Erros em Bibliotecas

```
/* header file <ctype.h> */
#define isdigit (c) ((c) >= '0' && (c) <= '9')
```

- O que acontece quando faço o seguinte?

```
while (isdigit (c = getchar())
  ...
```

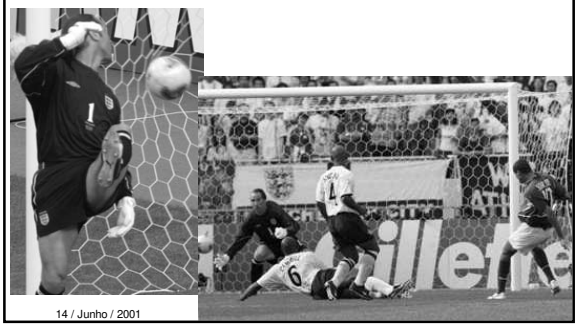

Erros no Hardware

- Erro do ponto-flutuante do Pentium em 94.
- Erro do VIC-20 em 1982. Raiz de 4.
- Computador multiprocessador:
  - às vezes 1/2 = 0.5
  - às vezes 1/2 = 0.7432
- Estimativa da temperatura do computador de acordo com o número de bits errados nas contas com ponto-flutuante.

Depuração quase impossível: *Sistema Defensivo Brasileiro*

Às Vezes Parece que É Impossível Consertar

Mas Nada É Impossível!



Erros Não-Reprodutíveis

- São os mais difíceis.
- O fato de ser não-reprodutível já é uma informação importante.
- O seu programa está utilizando informações diferentes cada vez que é executado.
- Verifique se todas as variáveis estão sendo inicializadas.

Erros Não-Reprodutíveis

- Se o erro desaparece quando você roda o depurador, pode ser problema de alocação de memória.
  - Usar posições de um vetor além do tamanho alocado.
  - Posição de memória que é liberada mais do que uma vez.
  - Mau uso de apontadores (próximo slide)

Problemas com Apontadores

```
char *msg (int n, char *s)
  {
    char buf[256];
    sprintf (buf, "error %d: %s\n", n, s);
    return buf;
  }
```

```
for (p = lista; p != NULL; p = p->proximo)
    free (p);
```

Ferramentas de Monitoramento de Memória

- Purify (para Solaris)
- Bounds Checker (para Windows)
- Tipos de Verificações:
  - vazamentos de memória (*memory leaks*)
  - violação de limites de vetores e matrizes.
  - uso de posição não alocada.
  - uso de posição não inicializada.
  - free sem malloc, malloc sem free, duplo free, free em quem não foi alocado.

Erros em Código Escrito por Outros

- É muito comum termos que depurar código dos outros.
- As mesmas técnicas de depuração se aplicam.
- Mas temos que nos familiarizar um pouco com o código antes de começarmos.
  - Rode o programa com o depurador passo a passo.

Submetendo Relatórios de Erros (*Bug Reports*)

- Tenha certeza de que o erro é realmente um erro (não passe ridículo!).
- Tenha certeza de que o erro é novo (você tem a versão mais recente do programa?).
- Escreva um relatório sucinto mas contendo todas as informações relevantes.
- Não diga: "rodei o programa mas não funcionou".

Um Bom Relatório de Erro

- Versão do Programa e linha de comando.
- Sistema Operacional e versão.
- Compilador e versão.
- Versão das bibliotecas (se relevante).
- Uma pequena entrada que gera o erro.
- Uma descrição do erro.
- Se possível, a linha de código errada.
- Se possível, a correção.

Um Bom Relatório de Erro

- Se for o caso, um programinha que evidencia o erro:

```
/* Teste para o erro do isdigit () */
int main (void)
{
  int c;
  while (isdigit (c = getchar ()) && c != EOF)
     printf ("%c");
  return 0;
}

%echo 1234567890 | teste_do_isdigt
24680
%
```


Resumindo

- Quando um erro é avistado, pense bem em quais dicas o erro está lhe dando.
- Como ele pode ter acontecido?
- Ele é parecido com algo que você já viu?
- Você acabou de mexer em alguma coisa?
- Há algo de especial na entrada que causou o erro?
- Alguns poucos testes e alguns poucos **`prints`** podem ser suficientes.

Resumindo

- Se não há dicas, o melhor é pensar muito cuidadosamente sobre o que pode estar acontecendo.
- Daí, tente *sistematicamente* localizar o problema eliminando pedaços da entrada e do código.
- Explique o código prá mais alguém.
- Use o depurador prá ver a pilha.
- Execute o programa passo a passo.

Resumindo

- Use todas as ferramentas que estão a sua disposição.
- Conheça-se a si mesmo. Quais os tipos de erros que você costuma fazer?
- Quando encontrar um erro, lembre de eliminar possíveis erros parecidos em outras partes do seu código.
- Tente evitar que o erro se repita no futuro.

Moral da História

- Depuração pode ser divertido desde que feita de forma sistemática e organizada.
- É preciso praticar para obter experiência.
- Mas, o melhor a se fazer é escrever bom código pois
  - ele tem menos erros e
  - os erros são mais fáceis de achar.

Bibliografia

Brian W. Kernighan e Rob Pike.
The Practice of Programming.
Addison-Wesley, 1999.

Capítulo 5: *Debugging*.