



ALGORITMOS E PROGRAMAÇÃO DE COMPUTADORES

Depuração

Prof. Daniel Sundfeld Lima
daniel.sundfeld@unb.br



INTRODUÇÃO

- O uso de boas práticas de programação, a experiência e uma boa sequência de testes auxiliam na produção de um bom código que possui poucos erros e bugs
- No entanto, até mesmo os programadores mais experientes podem cometer deslizes, erros de digitação, que podem induzir erros simples ou complexos em um código



INTRODUÇÃO

- É inevitável que existam bugs em produtos de software, alguns dizem que todos os softwares possuem algum bug
- Por isso, é necessário que os programadores tenham hábitos para reduzir esses problemas para detectá-los o mais cedo possível



INTRODUÇÃO

- Além disso, ao se deparar com um problema e/ou comportamento inesperado, o programador deve conhecer um conjunto de ferramentas que o auxiliará em resolver o problema
- Na aula de hoje, estudaremos algumas dessas técnicas que auxiliam em detectar e corrigir bugs



AGENDA

- Warnings do compilador
- GDB
- Valgrind



COMPILANDO CORRETAMENTE

- O compilador é capaz de detectar vários erros durante a compilação
- No entanto, as vezes não é um erro, mas um código mal construído que é válido
- As flags de compilação com “Warning” podem ajudar o programador a receber avisos de possíveis erros



COMPILANDO CORRETAMENTE

- No entanto, algumas construções podem parecer estranhas, o compilador é capaz de produzir um código válido
- É recomendado que o programador habilite a maior quantidade de avisos possíveis



COMPILANDO CORRETAMENTE

- Para compilar o código “fonte.c” use:
- `gcc -Wall -o executavel fonte.c`
- Desta forma os avisos irão ser mostrados na tela
- Existem vários erros que podem ser detectados, vamos ver alguns simples



COMPILANDO CORRETAMENTE

- Observe o código

```
int main()
{
    int i = 0;

    if (i = 1)
        printf("Verdadeiro\n");
    return 0;
}
```



COMPILANDO CORRETAMENTE

- Nesse caso, o uso de “-Wall” irá detectar o problema

```
[user@default APC]$ gcc codigo.c -o executavel
[user@default APC]$ gcc -Wall codigo.c -o executavel
codigo.c: In function 'main':
codigo.c:7:9: warning: suggest parentheses around assignment used as truth value [-Wparentheses]
   7 |         if (i = 1)
     |             ^
     |             -
```



COMPILANDO CORRETAMENTE

- Observe o código

```
int main()
{
    int i = 6;

    if (i % 2 == 0);
    {
        printf("Par");
    }
    return 0;
}
```



COMPILANDO CORRETAMENTE

- Neste caso, o “-Wall” também irá detectar esse problema

```
[user@default APC]$ gcc codigo.c -o executavel
[user@default APC]$ gcc -Wall codigo.c -o executavel
codigo.c: In function 'main':
codigo.c:7:5: warning: this 'if' clause does not guard... [-Wmisleading-indentation]
   7 |     if (i % 2 == 0);
     |     ^~
```



COMPILANDO CORRETAMENTE

- Encontrar erros em tempo de compilação é muito mais fácil do que detectá-los em tempo de execução
- Por isso, recomendamos utilizar as flags apropriadas sempre!



COMPILANDO CORRETAMENTE

- No entanto, pode ser que o compilador gere um warning, que não represente um perigo, mas que foi realmente a intenção do programador



```
int par(int arg)
{
    if ((arg % 2) == 0)
        return 1;
    return 0;
}

int main()
{
    int ret;
    int i = 2;

    if (ret = par(i))
    {
        printf("E' par\n");
    }
}
```



COMPILANDO CORRETAMENTE

- Neste caso, é gerado o seguinte aviso

```
[user@default APC]$ gcc -Wall -o exe codigo.c
```

```
codigo.c: In function 'main':
```

```
codigo.c:15:9: warning: suggest parentheses around assignment used as t
```

```
15 |      if (ret = par(i))  
    |          ^~~
```

as truth value [-Wparentheses]



COMPILANDO CORRETAMENTE

- Normalmente, esses avisos podem ser eliminados com o uso de parênteses, cast, ou outra técnica

```
int main()
{
    int ret;
    int i = 2;

    if ((ret = par(i)) == 0)
    {
        printf("E' par\n");
    }
}
```



COMPILANDO CORRETAMENTE

- Neste caso, não há warnings:

```
[user@default APC]$ gcc -o exe codigo.c  
[user@default APC]$ gcc -Wall -o exe codigo.c  
[user@default APC]$ █
```



GDB

- O GDB, the GNU Project debugger, permite que você veja, durante a execução, as linhas de código sendo executadas, valor que as variáveis possuem
- Ele também permite dizer qual instrução foi executada no momento que o programa parou de executar (segmentation fault)



GDB

- O gdb pode iniciar a execução do seu programa, iniciar o debug de um programa que já está em execução e possui muitos recursos
- Iremos ver as funções básicas da ferramenta



GDB

- O gdb é um programa diferente do que o compilador
- E também é diferente do editor de texto que usa para escrever o código fonte
- Como ele irá saber qual o código que está sendo executado?



GDB

- Para isso, seu compilador precisa adicionar informações de debug no seu programa compilado
- Para isso devemos compilar da seguinte forma:
- `gcc -Wall -g -o executavel fonte.c`
- Assim, o gcc irá produzir um executável maior, pois contém mais informações



GDB

- Para executar um programa com gdb, inicie-o como argumento:
- `$ gdb ./executavel`
`[user@default APC]$ gdb ./executavel`
GNU gdb (GDB) Fedora 12.1-1.fc36
Copyright (C) 2022 Free Software Foundation, Inc.
...
Reading symbols from `./executavel...`
(gdb)
Assim, o gdb carregou e leu símbolos



GDB

- Demonstração de comandos do GDB:
- `r = run` -> inicia a execução do seu código
- `p = print` -> imprime uma variável
- `d = display` -> sempre imprime uma variável
- `b = break` -> pare a execução em um determinado ponto do código



VALGRIND

- Muitos problemas podem ser evitados com os avisos do compilador
- Mas esse conjunto é limitado a dados que possam ser determinados em tempo de compilação, muitas vezes a execução do seu programa varia pela entrada do usuário



VALGRIND

- O Valgrind é uma ferramenta de depuração durante o tempo de execução
- Conjunto com mais de 10 ferramentas (memcheck, cachegrind, callgrind, helgrind, drd, massif, dhat, lackey, none, exp-bbv, etc)
- A ferramenta padrão é a que usaremos: memcheck



- O memcheck ajuda o programador a determinar erros de memória, acessos inválidos, que o programador pode passar muito tempo procurando mas facilmente com uma execução
- Também é útil para verificar área de memória que não foi liberada (tópico avançado, com alocação dinâmica de memória)



VALGRIND

- Verifique o código abaixo e diga o erro:

```
int main()
{
    char str[425];
    int contador = 0;

    for (int i = 0; str[i] != '\0'; ++i)
    {
        if (str[i] >= '0' && str[i] <= '9')
            contador++;
    }
    printf("%d\n", contador);
    return 0;
}
```



VALGRIND

- Verifique o código abaixo e diga o erro:

```
int main()
{
    char str[425];
    int contador = 0;

    for (int i = 0; str[i] != '\0'; ++i)
    {
        if (str[i] >= '0' && str[i] <= '9')
            contador++;
    }
    printf("%d\n", contador);
    return 0;
}
```

Variável não inicializada



VALGRIND

- Como vimos, esse erro pode ser verificado pelos warnings do compilador

```
[user@default APC]$ gcc -Wall -o cod codigo.c
codigo.c: In function 'main':
codigo.c:10:16: warning: 'str' may be used uninitialized [-Wmaybe-uninitialized]
   10 |         if (str[i] >= '0' && str[i] <= '9')
       |                ~~~^~~
codigo.c:5:10: note: 'str' declared here
     5 |     char str[425];
       |         ^~~
codigo.c:10:33: warning: 'str' may be used uninitialized [-Wmaybe-uninitialized]
   10 |         if (str[i] >= '0' && str[i] <= '9')
       |                ~~~^~~
codigo.c:5:10: note: 'str' declared here
     5 |     char str[425];
       |         ^~~
```



VALGRIND

- Mas alguns erros podem ser sutis:

```
int main()
{
    char str[425] = "ABC1ABC1";
    int contador = 0;

    for (int i = 0; str[i] != '\0'; ++i)
    {
        if (str[i] >= '0' && str[i] <= '9')
            contador++;
    }
    printf("%d\n", contador);
    return 0;
}
```



VALGRIND

- Pode-se executar um código no valgrind, passando o executável como argumento:
- `valgrind [argumentos valgrind] programa [argumentos programa]`



VALGRIND

- Ao executar esse código, o valgrind reporta alguns erros:

```
[user@default APC]$ valgrind ./cod
==161688== Memcheck, a memory error detector
==161688== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==161688== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==161688== Command: ./cod
==161688==
==161688== Conditional jump or move depends on uninitialised value(s)
==161688==    at 0x4011B7: main (2.c:8)
==161688==
==161688== Conditional jump or move depends on uninitialised value(s)
==161688==    at 0x40118D: main (2.c:10)
```



VALGRIND

- Alguma ideia da origem do problema?
- Sem ideias? Precisa de um gdb para ajudar?

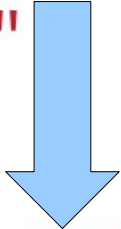


VALGRIND

- Faltou um '\'

```
int main()
{
    char str[425] = "ABC1ABC1"
    int contador = 0;

    for (int i = 0; str[i] != '0'; ++i)
    {
        if (str[i] >= '0' && str[i] <= '9')
            contador++;
    }
    printf("%d\n", contador);
    return 0;
}
```





VALGRIND

- Apesar de não apontar a linha que causou o problema, existe uma indicação que há algo de errado no programa e o programador pode buscar o problema



CONCLUSÃO

- O uso de flags apropriadas para a compilação, auxilia na detecção de erros no programa
- O gdb é um depurador que auxilia a verificar o valor de variáveis, mudanças de valores e execução do programa passo-a-passo



CONCLUSÃO

- O valgrind auxilia na verificação de alguns acessos inválidos a memória
- Mas ele não auxilia em todos os acessos inválidos
- Ele se torna ainda mais importante ao se utilizar com alocação dinâmica de memória!



DÚVIDAS?

- daniel.sundfeld@unb.br