Tutorial: Compilação e Depuração com GCC/GDB e VS Code

Índice

Introdução

Este tutorial ensina como compilar e debugar programas em C usando duas abordagens:

1. **Linha de comando** com GCC e GDB
2. **Interface gráfica** com VS Code

Usaremos como exemplo o programa `divide_outro.c` que verifica se um número divide outro.

Parte 1: Compilação com GCC

1.1 Código de Exemplo

Primeiro, vamos trabalhar com o seguinte código (`divide_outro.c`):

```c
#include<stdio.h>

int main() {
    int a, b, resto, divide;

    // assumir que nenhum número divide o outro
    divide = 0; // FALSO

    scanf("%d %d", &a, &b);

    if (a >= b) {
        resto = a % b;
        if (resto == 0) {
            divide = 1; // VERDADEIRO
        }
    }

    if (b >= a) {
        resto = b % a;
        if (resto == 0) {
            divide = 1; // VERDADEIRO
```

```
            }
        }

        if (!divide) {
            printf("Nenhum dos números divide o outro.");
        } else {
            printf("Um dos números divide o outro.");
        }

        return 0;
    }
```

1.2 Comandos de Compilação

Compilação Simples

```
gcc divide_outro.c -o divide_outro
```

Compilação Recomendada (com avisos)

```
gcc -Wall divide_outro.c -o divide_outro
```

Compilação para Debug

```
gcc -g -Wall divide_outro.c -o divide_outro
```

1.3 Execução

```
./divide_outro
```

Teste com entrada: `12 4` **Resultado esperado:** "Um dos números divide o outro."

1.4 Flags Importantes do GCC

Flag	Descrição
`-Wall`	Habilita avisos do compilador
`-g`	Inclui informações de debug
`-o nome`	Define nome do executável
`-std=c99`	Usa padrão C99

Flag	Descrição
`-O2`	Otimização nível 2

Parte 2: Depuração com GDB (Linha de Comando)

2.1 Preparação

```
# Compilar com informações de debug
gcc -g -Wall divide_outro.c -o divide_outro

# Iniciar GDB
gdb ./divide_outro
```

2.2 Comandos Básicos do GDB

Comando	Abreviação	Descrição
`run`	`r`	Executar o programa
`break linha`	`b linha`	Definir breakpoint
`break main`	`b main`	Breakpoint na função main
`step`	`s`	Executar próxima linha (entra em funções)
`next`	`n`	Executar próxima linha (não entra em funções)
`continue`	`c`	Continuar execução
`print variavel`	`p variavel`	Mostrar valor de variável
`info locals`		Mostrar todas as variáveis locais
`list`	`l`	Mostrar código fonte
`quit`	`q`	Sair do GDB

2.3 Sessão Prática de Debug

```
$ gdb ./divide_outro
(gdb) break main
Breakpoint 1 at 0x1149: file divide_outro.c, line 4.

(gdb) run
Starting program: ./divide_outro
Breakpoint 1, main () at divide_outro.c:4
4           int a, b, resto, divide;

(gdb) list
1       #include<stdio.h>
2
```

```
3       int main() {
4           int a, b, resto, divide;
5
6           // assumir que nenhum número divide o outro
7           divide = 0; // FALSO
8
9           scanf("%d %d", &a, &b);

(gdb) break 11
Breakpoint 2 at 0x1166: file divide_outro.c, line 11.

(gdb) continue
Continuing.
12 4
Breakpoint 2, main () at divide_outro.c:11
11          if (a >= b) {

(gdb) print a
$1 = 12

(gdb) print b
$2 = 4

(gdb) step
12              resto = a % b;

(gdb) step
13              if (resto == 0) {

(gdb) print resto
$3 = 0

(gdb) continue
Continuing.
Um dos números divide o outro.[Inferior 1 (process 1234) exited normally]
```

2.4 Técnicas de Debug Avançadas

Breakpoints Condicionais

```
(gdb) break 13 if resto == 0
```

Watch Points (monitorar mudanças em variáveis)

```
(gdb) watch divide
```

Examinar memória

```
(gdb) x/4i main    # Ver 4 instruções assembly da função main
```

Parte 3: Depuração no VS Code

3.1 Configuração Inicial

3.1.1 Extensões Necessárias

1. **C/C++** (Microsoft) - Essencial
2. **C/C++ Extension Pack** (Microsoft) - Recomendado

3.1.2 Estrutura de Pastas

```
projeto/
├── .vscode/
│   ├── tasks.json      # Configuração de compilação
│   └── launch.json     # Configuração de debug
└── divide_outro.c      # Código fonte
```

3.2 Configuração de Compilação (tasks.json)

Crie o arquivo `.vscode/tasks.json`:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C/C++: gcc build active file",
            "command": "/usr/bin/gcc",
            "args": [
                "-fdiagnostics-color=always",
                "-g",
                "-Wall",
                "${file}",
                "-o",
                "${fileDirname}/${fileBasenameNoExtension}"
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": ["$gcc"],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "detail": "Compilador GCC com flags de debug"
```

```
        }
    ]
}
```

3.3 Configuração de Debug (launch.json)

Crie o arquivo `.vscode/launch.json`:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "C/C++: gcc build and debug active file",
            "type": "cppdbg",
            "request": "launch",
            "program": "${fileDirname}/${fileBasenameNoExtension}",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "C/C++: gcc build active file",
            "miDebuggerPath": "/usr/bin/gdb"
        }
    ]
}
```

3.4 Processo de Debug no VS Code

3.4.1 Definindo Breakpoints

1. Abra o arquivo `divide_outro.c`
2. Clique na margem esquerda (ao lado dos números das linhas)
3. Um ponto vermelho aparecerá indicando o breakpoint

3.4.2 Iniciando o Debug

1. Pressione **F5** ou
2. Vá em **Run → Start Debugging** ou
3. Use o ícone de "play" no painel Debug

3.4.3 Controles Durante o Debug

Atalho	Botão	Ação
F5	▶	Continue
F10	⤴	Step Over
F11	⬇️	Step Into
Shift+F11	⬆️	Step Out
Shift+F5	⏹	Stop

3.4.4 Painéis Importantes

🔍 **Variables**: Mostra valores de todas as variáveis no escopo atual 👁️ **Watch**: Permite monitorar expressões específicas 📞 **Call Stack**: Mostra a pilha de chamadas de funções 🔗 **Breakpoints**: Lista todos os breakpoints definidos

3.5 ⚠ ATENÇÃO: Entrada de Dados no VS Code

Problema comum: Onde digitar quando o programa pede entrada via `scanf`?

✅ **Solução**: Digite no painel **TERMINAL** (não no Debug Console)

Passo a passo:

1. Programa executa até o `scanf`
2. Execução pausa aguardando entrada
3. **Clique na aba "TERMINAL"** na parte inferior da tela
4. Digite os valores (exemplo: `12 4`)
5. Pressione Enter
6. Programa continua a execução

3.6 Exemplo Prático: Debug no VS Code

1. **Defina breakpoints** nas linhas:
    - Linha 9 (antes do scanf)
    - Linha 11 (primeiro if)
    - Linha 18 (segundo if)
2. **Inicie o debug** (F5)
3. **Quando parar na linha 9**:
    - Observe no painel Variables que as variáveis ainda não têm valores definidos
    - Pressione F10 para ir para o scanf
4. **No scanf**:
    - Vá para a aba TERMINAL

- Digite: `12 4`
- Pressione Enter

5. **Continue o debug**:
   - Observe como os valores de `a` e `b` aparecem no painel Variables
   - Use F10 para executar linha por linha
   - Observe como `resto` e `divide` mudam de valor

Exercícios Práticos

Exercício 1: Debug Básico

1. Compile e execute o programa `divide_outro.c`
2. Teste com as entradas:
   - `12 4` (4 divide 12)
   - `7 3` (nenhum divide o outro)
   - `15 15` (números iguais)
3. Use tanto GDB quanto VS Code para verificar os valores das variáveis

Exercício 2: Encontrando Bugs

Considere este código com bug:

```c
#include<stdio.h>

int main() {
    int n, fatorial;
    fatorial = 1;

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

    for (int i = 1; i < n; i++) {  // BUG: deveria ser i <= n
        fatorial *= i;
    }

    printf("Fatorial de %d é %d\n", n, fatorial);
    return 0;
}
```

Tarefa: Use debug para encontrar e corrigir o erro.

Exercício 3: Comparação de Abordagens

1. Debug o programa do Exercício 2 usando GDB
2. Debug o mesmo programa usando VS Code
3. Liste 3 vantagens de cada abordagem

Troubleshooting

Problemas Comuns e Soluções

1. "gdb: command not found"

```
# Ubuntu/Debian
sudo apt install gdb

# macOS
xcode-select --install

# Windows (MSYS2)
pacman -S gdb
```

2. VS Code não encontra o debugger

- Verifique se a extensão C/C++ está instalada
- Confirme o caminho do GDB em `launch.json`
- No Linux: `/usr/bin/gdb`
- No macOS: `/usr/bin/gdb` ou `/opt/homebrew/bin/gdb`
- No Windows: caminho do MinGW ou MSYS2

3. Programa não para nos breakpoints

- Certifique-se de compilar com `-g`
- Verifique se o breakpoint está em uma linha executável
- Não defina breakpoints em comentários ou linhas vazias

4. VS Code: "Terminal will be reused by tasks"

- Normal - significa que a compilação foi bem-sucedida
- Se houver erros, eles aparecerão no painel "Problems"

5. Entrada não funciona no VS Code

- **Verifique se está digitando na aba TERMINAL**
- Se estiver usando externalConsole: true, uma janela separada abrirá

Dicas de Boas Práticas

1. **Sempre compile com `-g` para debug**
2. **Use `-Wall` para ver avisos do compilador**
3. **Defina breakpoints em pontos estratégicos**
4. **Monitore variáveis importantes no painel Watch**
5. **Use Step Over (F10) na maioria das situações**
6. **Use Step Into (F11) apenas quando quiser entrar em funções**

Recursos Adicionais

Documentação

- GDB Manual
- VS Code C++ Documentation
- GCC Manual

Comandos de Referência Rápida

GDB Quick Reference

```
# Controle de execução
(gdb) run [args]              # Executar programa
(gdb) continue                # Continuar execução
(gdb) step                    # Próxima linha (entra em funções)
(gdb) next                    # Próxima linha (não entra em funções)
(gdb) finish                  # Executar até sair da função atual

# Breakpoints
(gdb) break main              # Breakpoint na função main
(gdb) break arquivo.c:10      # Breakpoint na linha 10
(gdb) break funcao            # Breakpoint em função
(gdb) info breakpoints        # Listar breakpoints
(gdb) delete 1                # Remover breakpoint 1

# Inspeção
(gdb) print variavel          # Valor da variável
(gdb) print &variavel         # Endereço da variável
(gdb) print *pointer          # Valor apontado pelo pointer
(gdb) info locals             # Todas as variáveis locais
(gdb) list                    # Mostrar código fonte
(gdb) backtrace               # Stack trace
```

VS Code Atalhos

```
F5          - Start/Continue debugging
F9          - Toggle breakpoint
F10         - Step over
F11         - Step into
Shift+F11   - Step out
Shift+F5    - Stop debugging
Ctrl+Shift+Y - Show debug console
Ctrl+`       - Show terminal
```

Exemplos Avançados

Debug de Arrays

```c
#include<stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int soma = 0;

    for (int i = 0; i < 5; i++) {
        soma += arr[i];
        printf("arr[%d] = %d, soma atual = %d\n", i, arr[i], soma);
    }

    printf("Soma total: %d\n", soma);
    return 0;
}
```

Comandos GDB para arrays:

```
(gdb) print arr              # Mostra endereço base
(gdb) print arr[0]           # Primeiro elemento
(gdb) print *arr@5           # Todos os 5 elementos
(gdb) x/5i arr               # Ver 5 elementos em hexadecimal
```

Debug de Ponteiros

```c
#include<stdio.h>
#include<stdlib.h>

int main() {
    int *ptr = malloc(sizeof(int) * 5);

    for (int i = 0; i < 5; i++) {
        ptr[i] = i * 2;
    }

    printf("Valores: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", ptr[i]);
    }

    free(ptr);
    return 0;
}
```

Comandos GDB para ponteiros:

```
(gdb) print ptr              # Endereço do ponteiro
(gdb) print *ptr             # Primeiro valor
(gdb) print ptr[2]           # Terceiro elemento
(gdb) x/5w ptr               # Ver 5 words na memória
```

Cenários Comuns de Debug

1. Segmentation Fault

```c
#include<stdio.h>

int main() {
    int *ptr = NULL;
    *ptr = 10;  // ERRO: Segmentation fault
    return 0;
}
```

Como debugar:

1. Compile com `-g`
2. Execute no GDB: `gdb ./programa`
3. `run` - programa vai crashar
4. `backtrace` - mostra onde ocorreu o erro
5. `list` - mostra o código problemático

2. Loop Infinito

```c
#include<stdio.h>

int main() {
    int i = 0;
    while (i < 10) {
        printf("i = %d\n", i);
        // BUG: esqueceu de incrementar i
    }
    return 0;
}
```

Como debugar:

1. Definir breakpoint dentro do loop
2. Verificar se a variável de controle está sendo alterada
3. Usar `continue` algumas vezes para verificar o padrão

3. Resultado Incorreto

```c
#include<stdio.h>

int main() {
    int a = 5, b = 2;
    float resultado = a / b;  // BUG: divisão inteira
    printf("5/2 = %.2f\n", resultado);
    return 0;
}
```

Como debugar:

1. Breakpoint antes da divisão
2. `print a` e `print b` para verificar valores
3. `step` para executar a divisão
4. `print resultado` - verá que o resultado é 2.00 em vez de 2.50
5. Descobrir que precisa fazer cast: `(float)a / b`

Configurações Avançadas do VS Code

Configuração com argumentos de linha de comando

```json
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug with args",
            "type": "cppdbg",
            "request": "launch",
            "program": "${fileDirname}/${fileBasenameNoExtension}",
            "args": ["arg1", "arg2"],
            "stopAtEntry": true,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb"
        }
    ]
}
```

Configuração para múltiplos arquivos

```json
{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C/C++: gcc build multiple files",
```

```
            "command": "/usr/bin/gcc",
            "args": [
                "-fdiagnostics-color=always",
                "-g",
                "-Wall",
                "*.c",
                "-o",
                "programa"
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": ["$gcc"],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        }
    ]
}
```

Checklist de Debug

Antes de Começar

- [ ] Código compila sem warnings (-Wall)
- [ ] Compilado com informações de debug (-g)
- [ ] Problema reproduzível
- [ ] Entrada de teste definida

Durante o Debug

- [ ] Breakpoints em pontos estratégicos
- [ ] Verificar valores de variáveis críticas
- [ ] Seguir fluxo de execução passo a passo
- [ ] Testar diferentes entradas

Após Encontrar o Bug

- [ ] Entender a causa raiz
- [ ] Implementar correção
- [ ] Testar com múltiplas entradas
- [ ] Verificar se não quebrou outras funcionalidades

Exercícios de Fixação

Exercício 4: Debug Colaborativo

Trabalhe em dupla para debugar este programa:

```c
#include<stdio.h>

int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}

int main() {
    int num;
    printf("Digite um número: ");
    scanf("%d", &num);

    printf("Fibonacci de %d é %d\n", num, fibonacci(num));
    return 0;
}
```

Tarefas:

1. Uma pessoa usa GDB, outra usa VS Code
2. Debug com entrada `n = 5`
3. Observem quantas vezes cada chamada recursiva é feita
4. Comparem as experiências de debug

Exercício 5: Projeto Integrado

Crie um programa que:

1. Leia um array de números do usuário
2. Ordene o array (bubble sort)
3. Procure um número específico (busca binária)

Requisitos de debug:

- Use breakpoints em cada função
- Monitore as variáveis durante a ordenação
- Verifique se a busca binária funciona corretamente

Conclusão

Este tutorial cobriu os aspectos essenciais da compilação e depuração em C usando GCC/GDB e VS Code. As principais lições são:

1. **Compilação adequada** é fundamental - sempre use `-g` para debug
2. **GDB** oferece controle total mas tem curva de aprendizado
3. **VS Code** fornece interface amigável mas requer configuração
4. **Debug sistemático** é mais eficiente que tentativa e erro
5. **Prática regular** desenvolve intuição para encontrar bugs

Próximos Passos

- Pratique com programas mais complexos
- Explore ferramentas como Valgrind para detecção de vazamentos de memória
- Aprenda sobre profiling para otimização de performance
- Configure ambientes de desenvolvimento para projetos maiores

Lembre-se: Debugging é uma habilidade que se desenvolve com prática. Não desanime se parecer complexo no início!