Tutorial Ultra-rápido: desenvolvendo programas C em Unix

Jerônimo C. Pellegrini

1 Introdução

Este é um breve guia para programação em C em Unix. O texto está longe de ser completo (na verdade nem tem estrutura definida ainda), mas já deve ser útil. Este texto não cobre o Eclipse, que também pode ser usado para programar em ambiente Unix.

Minha escolha pessoal: vim ou Emacs apenas como editor, e todas as outras ferramentas (gcc, gdb, vlagrind, make) na linha de comando.

Minha recomendação: aprenda a usar tudo "dentro do Emacs", tanto quanto possível (compilação, depuração, etc). O que não for possível, use na linha de comando. Aprenda Emacs Lisp; apesar de só funcionar para programas "dentro de Emacs", é uma linguagem muito elegante e muitíssimo útil para quem quer continuar programando em Unix (porque você configurará seu Emacs de diversas maneiras em elisp até conseguir seu "ambiente ideal").

2 Editores de texto

2.1 vi

O vi é um editor pequeno, para quem não gosta de programas pesados. Há duas versões interessantes:

- vim (versão mais usável e configurável do vi), O vim é muito configurável, oferece bom suporte para edição de programas em várias linguagens, mas não é um "ambiente integrado" (ele é um programa leve, e faz apenas uma tarefa, como é tradicional em ambientes Unix);
- gvim (versão do vim com interface gráfica).

O vim é um editor muito bom, mas tem uma curva de aprendizado um pouco íngreme. Inicialmente é importante saber algumas coisas:

O vi opera em dois *modos*: o modo de comando e o modo de edição. Quando é invocado a partir da linha de comando, o editor está em modo de comando, *e portanto não está pronto para editar texto!* Ele está esperando um comando.

Para passar do modo de comando para o modo de edição, digite a tecla **i** ou a tecla **insert**. Para voltar ao modo de comando, tecle **esc**.

Veja a tabela de referência rápida do vim: `http://tnerual.eriogerg.free.fr/vim.html`

Há também um tutorial em Português: `http://br-linux.org/artigos/vim_intro.htm`

2.2 Emacs

O Emacs é um editor altamente configurável.

Sem nenhuma configuração adicional, você pode chamar o Emacs e abrir um arquivo C. Ele automaticamente entrará no modo de edição C, com *syntax highlighting* e várias outras facilidades.

Uma boa referencia para uso do Emacs para programação é `http://www.cs.bu.edu/teaching/tool/emacs/programming/`

Você pode querer usar também o ECB (Emacs Code Browser): `http://ecb.sourceforge.net`

O Emacs pode ser programado para fazer qualquer coisa. A linguagem de configuração do Emacs, chamada "Emacs Lisp", ou "elisp", é elegante e poderosa. Em Emacs Lisp foram escritos calendários, leitores de email, jogos, ambientes integrados de desenvolvimento e diversos outros programas.

Você pode aprender elisp aqui:

- `http://www.rattlesnake.com/emacs-lisp-intro.html`
- `http://www.delorie.com/gnu/docs/emacs-lisp-intro/emacs-lisp-intro_toc.html`

3 Compilando

3.1 GCC

Um compilador muito usado em plataformas Unix é o GCC.

Para compilar o arquivo `programa.c`, gerando o executável `programa`, use:

```
gcc -o programa programa.c
```

Para compilar três arquivos fonte em um programa:

```
gcc programa parte1.c parte2.c parte3.c
```

Para que o compilador otimize o código gerado:

```
gcc -O3 prog prog.c
```

Há três níveis de otimização atualmente no GCC: O0 (nada), O1, O2 e O3 (o último é o que permite mais otimizações).

Para pedir ao compilaor que inclua símbolos 'uteis para depuração:

```
gcc -g prog prog.c
```

Desta forma o *debugger* (gdb) poderá identificar melhor os problemas no código.

3.2 Makefiles

Um *Makefile* é um arquivo com informações sobre os arquivos de programa C que fazem parte do seu projeto. O programa `make` compila programas de acordo com as especificações em Makefiles.

4 Depurando

4.1 GDB

Para depurar um programa com o gdb, use a flag `-g` na compilação:

```
gcc -g prog prog.c
```

É bom também não permitir que o compilador otimize o código gerado, para facilitar a identificação das linhas no código fonte.

Este texto descreve três maneiras de usar o gdb:

- Usando um front-end gráfico
- A partir do Emacs
- Diretamente

4.1.1 Front-ends para o GDB

Há alguns *front-ends* gráficos para o GDB. Dois deles são o ddd e o xxgdb.

4.1.2 GDB a partir do Emacs

O Emacs pode funcionar como front-end para o GDB.

4.1.3 Uso direto do gdb

```
programa a b c
... <erro>
```

Invocamos o gdb, dando a ele o nome do programa que será depurado:

```
gdb programa
...
(gdb) run a b c
...                    (o programa é executado)
```

No exemplo acima, a, b e c são parâmetros de linha de comando do programa sendo depurado, que passamos ao comando `run` do gdb.

Podemos incluir *breakpoints* e observar variáveis:

```
(gdb) break 8
Breakpoint 1 at 0x400524: file erro.c, line 8.
(gdb) run
Starting program: /home/jeronimo/tmp/a.out

Breakpoint 1, main () at erro.c:8
8 vec = (int*) malloc (sizeof(int) * 10);
(gdb) print vec
$1 = (int *) 0x0
(gdb) next
10 for (i=0;i<20;i++)
(gdb) print i
$2 = 32767
11 vec[i] = 5;
(gdb) print i
$3 = 0
```

E se o programa estiver gerando erros durante a execução, podemos simplesmente executá-lo no gdb para tentar identificar o erro. Por exemplo:

```
#include <stdio.h>

int main() {
        int *vec;

        printf("%d\n",*vec);
        return 0;
}
```

Se tentarmos executar o programa, um erro será gerado:

```
./erro2
Segmentation fault
```

Podemos simplesmente pedir ao gdb que execute o programa:

```
gdb erro2
...
(gdb) run
Starting program: /home/jeronimo/tmp/erro2

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004d8 in main () at erro2.c:6
6 printf("%d\n",*vec);
```

O gdb nos diz que o erro ocorreu na linha 6, neste `printf`. No entanto, para erros de acesso à memória é melhor usar o valgrind.

4.2 Valgrind

O valgrind é programa muito útil que ajuda a localizar erros de acesso a memória (ponteiros perdidos, acesso além do limite de vetores, etc). Quando seu programa gerar "segmentation faults", **compile o programa com a flag -g** e rode-o no valgrind:

```
valgrind programa
```

Observe a saída.

O programa a seguir, por exemplo, contém dois erros de acesso à memória:

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

int main() {
        int i;
        int *vec;

        vec = (int*) malloc (sizeof(int) * 10);

        for (i=0;i<20;i++)
                vec[i] = 5;

        for (i=0;i<20;i++)
                printf("vec[%d] = %d\n",i,vec[i]);
}
```

Ao executá-lo no valgrind, somos informados dos erros:

```
==30975== Invalid write of size 4
==30975==    at 0x400548: main (erro.c:11)
==30975==  Address 0x5179058 is 0 bytes after a block of size 40 alloc'd
==30975==    at 0x4C20FEB: malloc (vg_replace_malloc.c:207)
==30975==    by 0x40052D: main (erro.c:8)
vec[0] = 5
...
vec[8] = 5
vec[9] = 5
==30975==
==30975== Invalid read of size 4
==30975==    at 0x40056E: main (erro.c:14)
==30975==  Address 0x5179058 is 0 bytes after a block of size 40 alloc'd
==30975==    at 0x4C20FEB: malloc (vg_replace_malloc.c:207)
==30975==    by 0x40052D: main (erro.c:8)
```

Notem que o valgrind detectou tanto o erro onde se escreve após o final do vetor como aquele em que se lê, informando a linha do programa em que cada erro ocorre. No primeiro caso, o erro está em `erro.c:11`, onde se escreve além do fim do vetor alocado na linha 8 (ele também diz onde o vetor foi alocado!)

```
==30568== LEAK SUMMARY:
==30568==    definitely lost: 40 bytes in 1 blocks.
==30568==      possibly lost: 0 bytes in 0 blocks.
==30568==    still reachable: 0 bytes in 0 blocks.
==30568==         suppressed: 0 bytes in 0 blocks.
```

Além disso, ele detectou o *memory leak*: o vetor de 20 inteiros (40 bytes) foi alocado e nunca liberado com `free`. Se rodarmos novamente o valgrind passando o parâmetro `--leak-check=full` ele nos dirá onde alocamos a memória que nunca foi liberada:

```
==30974== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==30974==    at 0x4C20FEB: malloc (vg_replace_malloc.c:207)
==30974==    by 0x40052D: main (erro.c:8)
```

5 Outras ferramentas

- **cscope** – permite buscar símbolos no código C. Bom se você precisa localizar variáveis, constantes etc. Dele você pode chamar o vi diretamente.
- **gprof** – um *profiler* identifica quanto cada parte do programa demora para executar. Ajuda a localizar gargalos.

6 Livros recomendados

Há dois livros importantes sobre a linguagem C: um tutorial e uma referência.

- Kernighan, B.; Ritchie, D. *C: a linguagem de programação.* (Tutorial)
- Harbison, S.; Steele, G. *C: manual de referência.* (Referência)

Estes são provavelmente os únicos livros de C que você poderá vir a precisar em toda a vida.

7 Extra

Aqui estão algumas ferramentas não diretamente relacionadas com programação, mas que poderão ser úteis:

- LaTeX: sistema de editoraçõa de textos onde há separação entre conteúdo e apresentação. O mais usado dentro do mundo das ciências exatas;
- GNU R: sistema para computação estatística;
- Graphviz: desenha grafos;
- Dia, ipe e xfig: eitores de diagramas;

- Octave: semelhante ao Matlab;
- GNUPlot: sistema para plotar gráficos.