DEEC
DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES
TÉCNICO LISBOA

Licenciatura em Engenharia Electrotécnica e de Computadores (LEEC)

Algoritmos e Estruturas de Dados
Aula prática #01

Conteúdo

1 Objectivos

Pretende-se com esta aula familiarizar o aluno com metodologias de trabalho no desenvolvimento de código. São abordados os seguintes tópicos: desenvolvimento, compilação e depuração (debug) de programas em C usando a linha de comando (gcc e gdb). No final da aula os alunos deverão:

- Conhecer um conjunto mínimo de comandos UNIX[1] – listar conteúdo da directoria, mudar de directoria, apagar ficheiros, mover ficheiros, criar e apagar directorias.
- Saber como procurar ajuda na utilização de comandos UNIX.
- Utilizar um editor de texto e conhecer um conjunto mínimo de operações de edição.
- Reconhecer a utilidade do editor no desenvolvimento de código, com especial ênfase na indentação correcta, verificação de chavetas e parêntesis, etc..
- Saber como compilar um programa usando o comando gcc – com opções de debug, definindo o nome do executável, etc..
- Perceber a metodologia básica de depuração de erros de compilação, interpretando as mensagens de erro produzidas na compilação e procurando as suas causas.
- Conhecer o ambiente de depuração em runtime gdb[2], e um conjunto mínimo de comandos – listar linhas de código, criar e remover breakpoints, em linha e em função; executar passo a passo, com e sem entrada nas funções; examinar conteúdo de variáveis; usar o menu de ajuda do gdb.

2 Plano de aula

Para atingir os objectivos anteriormente listados propõe-se o seguinte plano de aula.

1. Ilustrar a utilização de comandos UNIX e do comando man.
2. Ilustrar um episódio de edição de código com o emacs[3] ou vim[4].
3. O programa seguinte calcula a raiz quadrada de um número real. Este programa tem um conjunto variado de erros sintáticos e de lógica. Devem identificar os problemas e propor soluções.

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char ** argv)
{
  int i, size, dots=0;
  int x;

  printf("I am a mathematical genius and can calculate the square root of any
    number!!!\n\n");

  if (argc < 2 || argc > 20)
    printf("A single number is needed as an argument!\n");
    exit(EXIT_FAILURE);

  if (size == strlen(argv[1]) > 1){
    i = 0;
    if (argv[1][0] == '-' || argv[1][0] == '+') i=1;
    for(; i<size; i++)
    if (argv[1][i] < '0' || argv[1][i] > '9') {
     if (argv[1][i] == '.' && dots == 0) {
       dots = 1;
       continue;
     }
     printf("Not a valid number! Pay attention. I'm fragile\n");
     exit(EXIT_FAILURE);
   }
  }
  else {
    if (argv[1][0] < '0' || argv[1][0] > '9') {
      printf("Not a valid number! Pay attention. I'm fragile\n");
     exit(EXIT_FAILURE);
   }
  }

  x=atof(argv[1]);

  if (x < 0) {
    printf("Today, I'm not in the mood for negative numbers! Try tomorrow that I
     may accommodate your request...\n");
    exit(EXIT_FAILURE);
  }
  else
    printf("Square root of %.2lf is %.4lf\n", x, sqrt(x));

  exit(EXIT_SUCCESS);
}
```

O programa acima encontra-se disponível no ficheiro `aula1a.c`. Devem compilar este programa com gcc, usando o comando `gcc -Wall -o c1a class1a.c -lm` e registar as linhas onde surgem mensagens de erro (para facilitar, talvez seja boa ideia ter duas sessões abertas: uma para a compilação e outra para a edição).

Para remover os erros de lógica, devem compilar adcionando a flag `-g` e utilizar o gdb, nomeadamente o seguinte conjunto de comandos que este depurador de código proporciona: `help`, `list`, `break`, `next`, `step`, `continue`, `delete`, `print`, `backtrace`, etc. Um pequeno guia do gdb encontra-se em anexo.

4. Considere um programa que se encontra estruturado em três ficheiros:

   (a) `Complex.h/Complex.c`: Declaração e definição de funções para processamento de números complexos bem com a estrutura de dados `Complex`

   (b) `aula1b.c` Programa que calcula $\|\mathbf{z}\|$, $\Re(\mathbf{z^2})$, $\Im(\mathbf{z^2})$, $\Re(\mathbf{z^{-1}})$, $\Im(\mathbf{z^{-1}})$ and $\theta(\mathbf{z})$.

   Tal como na questão anterior, identifique os erros sintáticos e de lógica recorrendo ao gcc/gdb.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Complex.h"

```

```c
int main(int argc, char ** argv)
{
  int operation;
  double a, b;
  printf("I am an expert in complex numbers algebra!\n\n");

  if (argc < 4) {
    printf("Please provide a and b for a+ib being the complex you want to generate, followed by a digit between 0 and 5\n");
    exit(EXIT_FAILURE);
  }
  if (argc > 4) {
    printf("You've entered too much data.\n");
    exit(EXIT_FAILURE);
  }

  a = atof(argv[1]);
  b = atof(argv[2]);
  operation = atoi(argv[3]);
  Complex * t, * u;

  switch (operation) {
  case 0:
    t = ComplexInit(a, b);
    printf("The norm of your complex is %0.3lf\n", ComplexNorm(t));
    ComplexFree(t);
    break;
  case 1:
    t = ComplexInit(a, b);
    u = ComplexMultiply(t, a);
    printf("The real part of the square of your complex is %0.3lf\n", ComplexReal(u));
    ComplexFree(t);
    ComplexFree(u);
    break;
  case 2:
    t = ComplexInit(a, b, u);
    u = ComplexMultiply(t, t);
    printf("The imaginary part of the square of your complex is %0.3lf\n", ComplexImaginary(u));
    ComplexFree(t);
    ComplexFree(u);
    break;
  case 3:
    t = ComplexInit(a, b);
    u = ComplexInverse(t);
    printf("The real part of the inverse of your complex is %0.3lf\n", u->Real);
    ComplexFree(t);
    ComplexFree(u);
    break;
  case 4:
    t = ComplexInit(a);
    u = ComplexInverse(t);
    printf("The imaginary part of the inverse of your complex is %0.3lf\n", ComplexImaginary(u));
    ComplexFree(t);
    ComplexFree(u);
    break;
  case 5:
    t=ComplexInit(a, b);
    printf("The phase of your complex id %0.3lf\n", Complexphase(t));
    ComplexFree(t);
    break;
  default:
    printf("No such option\n");
    exit(EXIT_FAILURE);
    break;
  }
}
```

5. Compile e execute o seguinte programa, disponível no ficheiro `aula1c.c` e que como pode ver gera um `segmentation fault`. Com o gdb coloque um *breakpoint* na função `ll_equal` e corra o programa. Avançe linha a linha, todas as instruções da função e examine os valores das variáveis, especialmente aos ponteiros `a` e `b`. Encontre o erro e corrija-o !

```
#include <stdio.h>

typedef struct node {
  int val;
  struct node* next;
} node;

int ll_equal(node* a, node* b) {
  while (a != NULL) {
    if (a->val != b->val)
      return 0;
    a = a->next;
    b = b->next;
  }
  /* lists are equal if a and b are both null */
  return a == b;
}

int main(int argc, char** argv) {
  int i;
  node nodes[10];

  for (i=0; i<10; i++) {
    nodes[i].val = 0;
    nodes[i].next = NULL;
  }

  nodes[0].next = &nodes[1];
  nodes[1].next = &nodes[2];
  nodes[2].next = &nodes[3];

  printf("Equal test 1 result = %d\n", ll_equal(&nodes[0], &nodes[0]));
  printf("Equal test 2 result = %d\n", ll_equal(&nodes[0], &nodes[2]));

  return 0;
}
```

Referências

[1] Linux Command Cheat Sheet, https://www.guru99.com/linux-commands-cheat-sheet.html

[2] GDB Quick Reference, https://users.ece.utexas.edu/ adnan/gdb-refcard.pdf

[3] GNU Emacs Reference Card, https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf

[4] VIM Cheat Sheet, https://vim.rtorr.com

Annex - Short GDB Guide

Introduction

This document serves as a guide on how to use several gdb commands. The commands listed in this document are the ones we believe you will use the most frequently. There are many other gdb commands not included in this guide which you can read about online or by typing `help` inside gdb.

Notation

The notation we will use for commands in this document is `[c]ommand (args)`, where "command" is the command name and the abbreviated version of commands is listed in brackets (in this case "c").

Getting Started

Consider the program basic.c, which you want to run in gdb. First you must compile basic.c into an executable. To run it in gdb, use the following command in your terminal:

```
gdb basic
```

Now you should go inside gdb! You can use the command `layout src` to view the source file while debugging.

Running and Stopping a Program

To start execution, you can use the command `run` followed by any arguments that might be required to run the program.

```
(gdb) [r]un args                (start running program)
```

The program will pause execution at the first breakpoint encountered. You could step through the lines in your program using `next` or `step` or you can continue running the program until the next breakpoint using `continue.`

```
(gdb) [c]ontinue                (continue to the next breakpoint or until the end if
                                there no more breakpoints)
(gdb) [n]ext                    (continue to next line, steps over function calls)
(gdb) [s]tep                    (continue to next line, steps into function calls)
```

You can stop your program execution using CTRL+C and you can quit gdb using the `quit` command.

```
(gdb) CTRL+C                    (stop program executing)
(gdb) [q]uit                    (quits GDB)
```

Listing

You can list the code which you are debugging. This command can be used at any time that you have stopped execution.

```
(gdb)  [l]ist                    (list code at the current position)
(gdb)  [l]ist 230                (list code at line 230 within file)
(gdb)  [l]ist helper.c:120       (list code at line 120 in file helper.c)
(gdb)  [l]ist CheckArgs          (list code of function CheckArgs)
```

Breakpoints

Breakpoints are used in gdb to stop a program whenever a certain point in the program (the breakpoint) is reached. Breakpoints are particularly useful for debugging.

You can set breakpoints at a function, at a specific line within the file you are currently running, or even at a line within another file.

```
(gdb) [b]reak main           (set breakpoint at function main)
(gdb) [b]reak 101            (set breakpoint at line 101 within file)
(gdb) [b]reak helper.c:101   (set breakpoint at file helper.c, line 101)
```

You can use the info command to get information about the breakpoints you have set and the breakpoint numbers they correspond to. You can also delete breakpoints with the delete command.

```
(gdb) [i]nfo breakpoints     (show all breakpoints)
(gdb) [d]elete 1             (delete breakpoint 1)
(gdb) [d]elete               (delete all breakpoints)
```

Viewing Data

The `print` command is useful to display variable names (from the current scope), memory addresses, registers, and constants.

```
(gdb) [p]rint <expression>     (prints the value that expression evaluates to)
(gdb) [p]rint/x <expression>  (same as above, but prints in hexadecimal)
```

<expression> can be a variable, a numerical expression or an expression returning an address whose content is printed

```
(gdb)  [p]rint x                 (print value of variable x)
(gdb)  [p]rint 2 * x             (print result of 2 * the value of x)
(gdb)  [p]rint *ptr              (print value pointed to by ptr)
(gdb)  [p]rint ptr->next->z      (print value of element z of “structure” pointed by
                                  ptr->next)
```

It is possible to check the value of all variables in a given frame at once

(gdb) info locals	(shows the values of all variables local to the current frame)

It is also possible to monitor the value of certain variables when execution is stopped in the current frame

(gdb) display x	(shows the value of variable x whenever execution stops)
(gdb) display <expression>	(shows the value of <expression> whenever execution stops)

It is also possible to setup a watchpoint for a variable or expression such that when that variable is modified (or read), execution stops at that point

(gdb) rwatch <expression>	(execution stops when the value of <expression> is read)
(gdb) watch <expression>	(execution stops when the value of <expression> is modified)

Not all implementations of gdb support setting watchpoints as it requires hardware support.

More Debugging

A frame contains the arguments given to the function, the function's local variables, and the address at which the function is executing. When your program is started, the stack has only one frame, that of the function main. Every time a function is called, a new frame is created. When the function returns, that frame is deleted. A backtrace is a summary of how your program got where it is. It prints a stack trace starting from the current frame of execution.

`(gdb) [b]ack[t]race`	(display stack backtrace)

The information command is useful to learn more about the frame. (Note: it can also be used to list information about several other things, such as functions, registers, etc.)

`(gdb) [i]nfo frame`	(provide information about current frame)

You can use the `frame` command to inspect a frame with a given number.

`(gdb) frame i`	(inspect frame i)

You can move and up and down the frame stack

(gdb) up	(if on frame i, move to frame i-1)
(gdb) down	(if on frame i, move to frame i+1, if it exists)

In each frame you can examine the variables from that frame.

Note: in certain severe conditions, a program may crash in such a way that the frame information is lost. In those cases it is not possible to recover information about the crash from examining the frames. Furthermore, the crash itself must be avoided to allow the debugging session to be useful.

FAQ

- **GDB gives me the error message: “not in executable format: File format not recognized”**
  You want to make sure that you have compiled your program into an executable, and then run gdb on that executable.

- **When should I set a breakpoint?**
  If you know that one particular function is causing a segmentation fault, then you can set a breakpoint on that particular function. If you don’t know where to start setting breakpoints, the best place to set it would be on main.

- **Why does layout src not work?**
  Make sure that you have run the program using the `r` command and that you have set breakpoints. If layout src does not work, it might mean that the breakpoints weren’t reached and you have exited or the program terminated due to a segmentation fault.

- **When I run gdb, my interface doesn’t look like it should, what should I do?**
  Type Ctr-L

- **Why does running the program just exit the program?**
  Make sure that you have set the breakpoint properly! Or the breakpoint that you set is reachable. If you’re running it and the program just exits normally, it could mean that you fixed the bug!

- **When should I use the command “bt”?**
  You might want to use `bt` to learn about where a segmentation fault was caused.