Unicamp – Universidade Estadual de Campinas

FT – Faculdade de Tecnologia

Concurso Livre-Docente

Prof. Dr. André F. de Angelis

Prova Didática

Plano de Aula e Códigos-Fonte

Limeira/SP

Abril - 2022

Sumário

1. Plano de Aula

Aula preparada para o concurso Livre-Docente da Faculdade de Tecnologia, disciplina SI100 – Algoritmos e Programação de Computadores I.

1.1. Tema e Roteiro

Tema da aula (tópico 5 do edital):

Ponteiros e alocação dinâmica de memória

Roteiro da Aula:

- **Ponteiros**: conceito, declaração, operadores associados
- **Alocação** dinâmica de memória: requisição, uso e liberação (ciclo de vida)
- **Ponteiros** para funções: conceitos e aplicação (*toy application*)
- **Estrutura autorreferentes**: conceitos e aplicação (pilha de inteiros de escopo local)

1.2. Escolha do tema

A disciplina SI100 – Algoritmos e Programação de Computadores I é oferecida no 1º. semestre e representa para muitos alunos o contato inicial com a programação de computadores. Portanto, trabalha com conceitos importantes, mas por vezes básicos, especialmente nos tópicos introdutórios. Assim, alguns desses tópicos podem ser desfavoráveis à apresentação em uma prova didática, por oferecerem menores oportunidades de avaliação.

Na minha visão, ponteiros e alocação dinâmica de memória são essenciais à programação. Estes conceitos transcendem os limites da linguagem C e do paradigma estruturado. No entanto, apesar de sua importância, os alunos geralmente reportam dificuldades no tópico e são, por vezes, refratários ao tema.

Ponteiros são um dos recursos mais poderosos da linguagem C e estão entre os mais difíceis de se dominar, segundo Deitel & Deitel (2016)[1]. Falhas no seu uso e gerenciamento causam erros intermitentes, difíceis de se depurar ou simplesmente catastróficos.

Assim, os programas precisam ser feitos com cuidado adicional com a qualidade do código e redobrada atenção à sua correção. Estas características implicam o projeto e a codificação de melhores programas e, portanto, favorecem em muito a proficiência dos alunos, elevando o seu nível técnico e a sua maturidade enquanto programadores. Consequentemente, obtém-se uma formação mais robusta e qualificada.

Como professor de Estrutura de Dados, necessito do uso intenso de ponteiros e alocação dinâmica de memória pelos alunos para implementação de estruturas tais como listas, pilhas, filas, árvores e grafos, o que reforça a importância do assunto.

Nas disciplinas de Programação Orienta a Objetos I e II, linguagens C++ e Java respectivamente, advogo a fundamental importância do emprego do polimorfismo. Este somente é alcançado

[1] C How to Program; Deitel & Deitel; Pearson; 8a. ed.; 2016; 973p.

com uso de ponteiros e alocação dinâmica, demonstrando a necessidade de domínio destes conceitos pelo aluno desde cedo.

Com efeito, acredito que é um tema desafiador para o docente e que a sua exploração é apropriada para o objetivo de avaliação da aula pela banca examinadora.

1.3. Contexto

Ponteiros e alocação dinâmica de memória são vistos ao final da disciplina SI100, geralmente a partir da 12ª. semana de aula (de 15 no total). O aluno cursa, em paralelo, TT106 - Organização e Arquitetura de Computadores. Logo, no momento de ter contato com o tema, o aluno já conhece os conceitos necessários de arquitetura (organização da memória, *heap*, pilha, etc.), os conceitos de programação (linguagens, lógica, fundamentos da programação estruturada, vetores, matrizes, *structs*, etc.) e tem experiência em desenvolvimento de código C. Em particular, a modularização de código é trabalhada durante o semestre todo, de sorte que o aluno está familiarizado com funções, prototipação de funções, passagem de parâmetros e divisão de código em arquivos específicos.

No semestre seguinte, a disciplina SI200 - Algoritmos e Programação de Computadores II continua na linguagem C, exercitando a prática de programação, mas adicionando como conteúdo novo apenas o conceito de recursão.

Portanto, a maior parte do conteúdo referente à linguagem C está à cargo da SI100 que deve cobrir, portanto, os principais conceitos envolvidos na programação, especialmente ponteiros e alocação dinâmica de memória.

1.4. Objetivo e habilidades pretendidas

O objetivo desta aula é permitir ao aluno compreender as ferramentas básicas de apontamento e alocação dinâmica de memória.

Ao final da aula, pretende-se que o aluno tenha adquirido as seguintes habilidades:

- Conhecer e usar ponteiros para alocação dinâmica de memória;
- Conhecer e usar ponteiros para fazer chamadas de funções;
- Conhecer e usar ponteiros para a construção de estruturas de dados.

1.5. Metodologia e recursos

A metodologia da aula preparada para prova didática é a aula expositiva, com a apresentação de conceitos teóricos e discussão de exemplos de código C preparados pelo professor. Slides eletrônicos projetados em tela são o recurso de apoio visual, enquanto os códigos-fonte completos são disponibilizados aos alunos em repositório públicos. Ao final da aula são propostos exercícios de fixação, de cunho eminentemente prático, que solicitam o desenvolvimento de código. Em regime regular de curso, esses exercícios são avaliados pelo professor a partir de uma lista de critérios previamente fornecida aos alunos e que inclui nos pontos de correção as funcionalidades esperadas e aspectos de clareza e qualidade técnica.

A bibliografia indicada encontra-se disponível fisicamente na biblioteca da faculdade.

O ambiente de desenvolvimento e teste dos programas C é o seguinte:

- Fedora Linux 35;
- IDE Eclipse 2021-12;
- Compilador gcc/g++ 11.2.1 20220127 (Red Hat 11.2.1-9).

O compilador fornece suporte ao padrão C 2018 (ISSO C18). Os códigos fazem uso exclusivo de funções e recursos da linguagem e das bibliotecas-padrão, não sendo dependentes de arquivos, bibliotecas ou recursos de terceiros.

1.6. Nota sobre os códigos-fonte C

Os exemplos de código C apresentados na aula são de dois tipos, segundo o objetivo de cada slide. O primeiro tipo é constituído de fragmentos de código isolados, destinados a ilustrar e enfatizar um ponto muito específico da discussão. São trechos com poucas linhas de código, contendo simplificações, cortes, omissão de tratamento de erros e outras manipulações, inclusive a apresentação de código incorreto. São usados como recurso didático na aula e não podem ser compilados. Na maioria das vezes, os identificadores de variáveis e funções estão com seus nomes em Português, facilitando ao aluno a compreensão do conceito em tela.

O segundo tipo é constituído de código C plenamente operacional, extraído de programas completos desenvolvidos pelo professor. São trechos um pouco mais extensos, por vezes divididos em slides distintos, usados para ilustrar conceitos de forma integrada, funcionalidades completas ou fornecer uma visão de contexto para o código. Devido à exclusão de elementos distrativos, tais como diretivas de pre-processador (especialmente *includes*) ou supressão de detalhes, nem sempre podem ser levados diretamente do slide para o compilador. No entanto, os códigos completos estão à disposição dos alunos que são, inclusive, estimulados a consultá-los como material de estudo e base de exercícios de fixação. Usualmente, os identificadores de variáveis e funções estão com seus nomes em Inglês, permitindo emprego de termos tradicionais (ex.: *push*, *pop*) e evitando uma combinação confusa de linguagens que possa prejudicar clareza e autodocumentação.

2. Códigos-fonte completos

Os códigos-fonte C aqui listados estão publicamente disponíveis em:

- *https://github.com/CafeForte/C-Code.git*

2.1. Exemplo 1: Ponteiros para Funções

Implementação de uma *toy application* (menu de operações)

2.1.1 Arquivo C_FuncionPointer.c (arquivo único)

```
/*
 * C_FuncionPointer.c
 *
 *  Created on: Mar 2, 2022
```

```
 *      Author: Andre Franceschi de Angelis
 */

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

#define PROGRAM   "C Function Pointer Demo"
#define BOLD      "##########################"
#define SEPARATOR "----------"
#define DEFAULT_MENU 1

int makeChoice(void);

int main(int argc, char *argv[])
        {
        void processTask(void);

        puts(PROGRAM ": Entering main function.");
        printf("Running: %s\n", argv[0]);
        puts(SEPARATOR);

        processTask();

        puts(SEPARATOR);
        puts(PROGRAM ": Leaving main function.");
        printf("Finishing: %s\n", argv[0]);
        return (0);
        }

void processTask()
        {
        int fixedMenuEnglish(void);
        int fixedMenuPortuguese(void);

        int actionOne(int);
        int actionTwo(int);
        int actionThree(int);

        void printMessage(void);

        // registering functions
        atexit(printMessage);

        int (*functions[3])(int);
        functions[0] = actionOne;
        functions[1] = actionTwo;
        functions[2] = actionThree;

        int (*menu)(void) = NULL;

        if (DEFAULT_MENU)
                menu = fixedMenuEnglish;
        else
                menu = fixedMenuPortuguese;

        // operations
        int choice = -1;

        while (choice)
                {
                choice = menu();

                switch (choice)
                        {
                        case 1:
```

```
                                case 2:
                                case 3:
                                        printf("Return received: %i\n\n", (functions[choice - 1])(choice));
                                        break;
                                case 4:
                                        exit(EXIT_SUCCESS);
                                        break;
                                case 5:
                                        exit(EXIT_FAILURE);
                                        break;
                                case 6:
                                        abort();
                                        break;
                                }
                        }
                }

int fixedMenuEnglish(void)
                {
                puts("--------------------------");
                puts("Menu");
                puts("--------------------------");
                puts("1. Action One");
                puts("2. Action Two");
                puts("3. Action Three");
                puts("4. Normal exit (success)");
                puts("5. Normal exit (failure)");
                puts("6. Abnormal exit (abort)");
                puts("0. Finish \'main\'");
                puts("--------------------------");
                return (makeChoice());
                }

int fixedMenuPortuguese(void)
                {
                puts("--------------------------");
                puts("Menu");
                puts("--------------------------");
                puts("1. Acao Um");
                puts("2. Acao Dois");
                puts("3. Acao Tres");
                puts("4. Saida normal (sucesso)");
                puts("5. Saida normal (falha)");
                puts("6. Saida anormal (aborto)");
                puts("0. Final de \'main\'");
                puts("--------------------------");
                return (makeChoice());
                }

int makeChoice()
                {
                int answer = -1;
                while ((answer < 0) || (answer > 6))
                        {
                        puts("Make your choice: ");
                        scanf("%1d", &answer);
                        }
                return answer;
                }

int actionOne(int value)
                {
                puts("Action One completed!");
                return (10 * value);
                }
```

```
int actionTwo(int value)
        {
        puts("Action Two completed!");
        return (20 * value);
        }

int actionThree(int value)
        {
        puts("Action Three completed!");
        return (30 * value);
        }

void printMessage()
        {
        puts("\n" BOLD);
        puts("# The Message is: goodbye #");
        puts(BOLD "\n");
        }
```

2.2. Exemplo 2: Estruturas autorreferentes

Implementação de uma pilha dinâmica de escopo local

2.2.1 Arquivo C_LocalStack.h

```
/*
 * C_StackPointer.h
 *
 *  Created on: Mar 2, 2022
  *      Author: Andre Franceschi de Angelis
*/

#ifndef C_StackPointer_H_
#define C_StackPointer_H_

struct data
        {
        int value;
        struct data *next;
        };

typedef struct data ** StackPointer;

void push(StackPointer, int);
int pop(StackPointer);
int isEmpty(StackPointer);
void clear(StackPointer);
void print(StackPointer);
size_t size(StackPointer);

#endif /* C_StackPointer_H_ */

// end of file
```

Obs: arquivo foi renomeado para C_LocalStack.h para upload no repositório, mas o conteúdo interno foi preservado com o nome original.

2.2.2 Arquivo C_LocalStack.c

```
/*
 * C_LocalStack.c
 *
 *  Created on: Mar 2, 2022
  *      Author: Andre Franceschi de Angelis
*/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "C_LocalStack.h"

void push(StackPointer stack, int newValue)
		{
		struct data *newData = (struct data*) malloc(sizeof(struct data));

		if (newData)
			{
			newData->value = newValue;
			newData->next = *stack;
			*stack = newData;
			}
		else
			{
			puts("Stack overflow.\n");
			abort();
			}
		}

int pop(StackPointer stack)
		{
		struct data *oldData = NULL;
		int oldValue;

		if (!isEmpty(stack))
			{
			oldValue = (*stack)->value;
			oldData = (*stack);
			(*stack) = (*stack)->next;
			free(oldData);
			}
		else
			{
			puts("Stack underflow.\n");
			abort();
			}
		return (oldValue);
		}

int isEmpty(StackPointer stack)
		{
		return (*stack == NULL);
		}

void clear(StackPointer stack)
		{
		struct data *oldData = NULL;

		while (*stack != NULL)
			{
```

```
                        oldData = *stack;
                        *stack = (*stack)->next;
                        free(oldData);
                        }
                }

void print(StackPointer stack)
                {
                struct data *dataPtr = *stack;
                unsigned count = 0;

                printf("(Top) ");
                while (dataPtr != NULL)
                        {
                        printf("%.2i -> %.5i; ", ++count, dataPtr->value);
                        dataPtr = dataPtr->next;
                        }
                printf(" (Bottom)");
                        }

size_t size(StackPointer stack)
                {
                struct data *dataPtr = *stack;
                size_t count = 0;

                while (dataPtr != NULL)
                        {
                        count++;
                        dataPtr = dataPtr->next;
                        }
                return (count);
                }

// end of file
```

2.2.3 Arquivo C_LocalStack_Start.c

```
/*
 * C_LocalStack_Start.c
 *
 *  Created on: Mar 2, 2022
  *      Author: Andre Franceschi de Angelis
*/

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

#include "C_LocalStack.h"

#define PROGRAM "C Local Scope Stack Reviewed"
#define SEPARATOR "----------"
#define UNDERFLOW false
#define STACKS 2048
#define LIMIT 1000
#define SAMPLES 50

void processTask(void);
void badProcessTask(void);
void showStackStatus(StackPointer, char*);
void printStackValues(StackPointer);
void loadStack(StackPointer, int);
void unloadStack(StackPointer);
```

```
void badUnloadStack(StackPointer);
void singleStackTest(void);
void stackArrayTest(void);

int main(int argc, char *argv[])
        {
        puts(PROGRAM ": Entering main function.");
        printf("Running: %s\n", argv[0]);
        puts(SEPARATOR);

        processTask();
        puts(SEPARATOR);
        badProcessTask();

        puts(SEPARATOR);
        puts(PROGRAM ": Leaving main function.");
        printf("Finishing: %s\n", argv[0]);
        return (0);
        }

void singleStackTest()
        {
        puts("\n");
        puts("**********************");
        puts("- SINGLE STACK TEST -");
        puts("**********************");

        struct data **singleStack = (struct data**) malloc(sizeof(struct data*));
        *singleStack = NULL;

        showStackStatus(singleStack, "SingleStack");
        puts("Loading   stack");
        loadStack(singleStack, SAMPLES);
        showStackStatus(singleStack, "SingleStack");
        puts("Clearing  stacki");
        clear(singleStack);
        printf("%s\n", "done!");
        showStackStatus(singleStack, "SingleStack");
        puts("Loading   stack");
        loadStack(singleStack, SAMPLES);
        showStackStatus(singleStack, "SingleStack");
        puts("Unloading stack");
        unloadStack(singleStack);
        showStackStatus(singleStack, "SingleStack");
        free(singleStack);
        singleStack = NULL;
        }

void stackArrayTest()
        {
        puts("\n");
        puts("**********************");
        puts("- STACK ARRAY TEST  -");
        puts("**********************");

        StackPointer stacksDemo[STACKS];
        char name[80];
        printf("On processTask - creating %i stacks\n\n", STACKS);
        for (int repeat = 0; repeat < STACKS; repeat++)
                {
                stacksDemo[repeat] = (StackPointer) malloc(sizeof(struct data*));
                *stacksDemo[repeat] = NULL;
                }
        for (int repeat = 0; repeat < STACKS; repeat++)
                {
```

```
                    sprintf(name, "Stack [%02i]", repeat);
                    printf("On processTask - testing %s\n", name);
                    puts(SEPARATOR);
                    showStackStatus(stacksDemo[repeat], name);
                    printf("Loading   Stack[%02i] ....: ", repeat);
                    loadStack(stacksDemo[repeat], SAMPLES);
                    showStackStatus(stacksDemo[repeat], name);
                    printf("Clearing  Stack[%02i] ....: ", repeat);
                    clear(stacksDemo[repeat]);
                    printf("%s\n", "done!");
                    showStackStatus(stacksDemo[repeat], name);
                    printf("Loading   Stack[%02i] ....: ", repeat);
                    loadStack(stacksDemo[repeat], SAMPLES);
                    showStackStatus(stacksDemo[repeat], name);
                    printf("Unloading Stack[%02i] ....: ", repeat);
                    unloadStack(stacksDemo[repeat]);
                    showStackStatus(stacksDemo[repeat], name);
                    printf("Loading   Stack[%02i] ....: ", repeat);
                    loadStack(stacksDemo[repeat], SAMPLES);
                    showStackStatus(stacksDemo[repeat], name);
                    }
            puts(SEPARATOR);
            printf("%s\n", "On processTask - listinga all local stacks");
            for (int repeat = 0; repeat < STACKS; repeat++)
                    {
                    sprintf(name, "Stack [%02i]", repeat);
                    showStackStatus(stacksDemo[repeat], name);
                    }
            puts(SEPARATOR);
            printf("%s\n", "On processTask - freeing all local stacks");
            for (int repeat = 0; repeat < STACKS; repeat++)
                    {
                    sprintf(name, "Stack [%02i]", repeat);
                    clear(stacksDemo[repeat]);
                    showStackStatus(stacksDemo[repeat], name);
                    free(stacksDemo[repeat]);
                    stacksDemo[repeat] = NULL;
                    }
            }

void processTask()
            {
            singleStackTest();

            stackArrayTest();
            }

void badProcessTask()
            {
            if (UNDERFLOW)
                    {
                    puts(SEPARATOR);
                    printf("%s\n", "BAD OPERATION SAMPLE - DO NOT DO THIS AT HOME\n");

                    char *name = "Other Stack";
                    StackPointer otherStack = (StackPointer) malloc(sizeof(StackPointer));
                    *otherStack = NULL;

                    showStackStatus(otherStack, name);

                    printf("Loading   %s ...: ", name);
                    loadStack(otherStack, SAMPLES);
                    showStackStatus(otherStack, name);

                    printf("Loading   %s ...: ", name);
```

```
                loadStack(otherStack, SAMPLES);
                showStackStatus(otherStack, name);

                printf("Unloading %s ...: ", name);
                badUnloadStack(otherStack);

                free(otherStack);
                *otherStack = NULL;
                }
        }

void showStackStatus(StackPointer stack, char *name)
        {
        printf("Checking  %s ...: The stack is%s empty. It has %lu elements.\t", name, (isEmpty(stack) ? "" : " not"),
size(stack));
        printStackValues(stack);
        }

void printStackValues(StackPointer stack)
        {
        printf("%s", ">> ");
        print(stack);
        printf("%s", " <<\n");
        }

void loadStack(StackPointer stack, int quantd)
        {
        printf("%s", "Pushing values: ");
        for (int count = 0; count < quantd; count++)
                {
                int newValue = rand() % LIMIT;
                printf("%.5i; ", newValue);
                push(stack, newValue);
                }
        puts("");
        }

void unloadStack(StackPointer stack)
        {
        printf("%s", "Poping values:");
        while (!isEmpty(stack))
                {
                printf("%.5i; ", pop(stack));
                }
        puts("");
        }

void badUnloadStack(StackPointer stack)
        {
        printf("%s", "Poping values crazily");
        while (true)
                {
                printf("%i; ", pop(stack));
                }
        puts("\n");
        }

// end of file
```

3. Exercícios propostos

- Baixe, examine, compile e rode os códigos de exemplo do professor, verificando cuidadosamente as saídas das execuções manuais (menu) e automatizadas (pilha), inclusive nas condições de saída forçada (*abort*) e erro (*overflow* e *underflow*).

- Escreva um programa C que receba, ordene e imprima um *array* de números reais de dupla precisão (*double*). Use o fato de que nomes de *arrays* são ponteiros para fazer a passagem dos dados por referência para uma função ordenadora (*bubblesort* ?). Passe, também como parâmetro, uma função comparadora do tipo apropriado, usando o conhecimento de ponteiros para funções. Ela determinará se elementos devem ser trocados.

- Implemente uma **fila** (estrutura *first in, first out*) usando alocação dinâmica de estruturas autorreferentes. Modifique o programa do professor para executar testes automatizados da sua implementação.

Limeira/SP
2022