

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS  
ESCOLA POLITÉCNICA E DE ARTES  
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



**INTERPRETADOR DBASE EM C - EXEMPLOS**

LEONARDO DE MOURA ALVES

GOIÂNIA  
2024

LEONARDO DE MOURA ALVES

## **INTERPRETADOR DBASE EM C - EXEMPLOS**

Trabalho apresentado à disciplina CMP1076 –  
Compiladores, do curso de Bacharel em Ciência  
da Computação, da Escola Politécnica e de  
Artes da Pontifícia Universidade Católica de  
Goiás, como parte dos requisitos para  
aprovação nesta disciplina.

Orientador:

Prof. Me. Claudio Martins Garcia

GOIÂNIA

2024

## 1 INTRODUÇÃO

Foi realizado neste trabalho o desenvolvimento de um interpretador de DBASE em linguagem C, com a finalidade de ler e executar comandos.

Esse interpretador processa um arquivo de texto contendo instruções em DBASE e é capaz de interpretar e executar os comandos CREATE TABLE, INSERT INTO e SELECT \* FROM.

Além da execução dos comandos, também foi implementado uma verificação de sintaxe para garantir a integridade dos dados e prevenir possíveis erros na execução das operações.

## 2 CÓDIGO COMPLETO

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

// Definição dos tamanhos máximos de campos e registros
#define MAX_FIELDS 10
#define MAX_RECORDS 100
#define MAX_FIELD_LENGTH 50

// Estrutura para definir um campo de tabela (nome e tipo)
typedef struct {
    char field_name[MAX_FIELD_LENGTH]; // Nome do campo
    char field_type;                    // Tipo do campo (C ou N)
} Field;

// Estrutura para definir uma tabela com campos e registros
typedef struct {
    char table_name[MAX_FIELD_LENGTH]; // Nome da tabela
    Field fields[MAX_FIELDS];           // Array de campos da tabela
```

```

    char records[MAX_RECORDS][MAX_FIELDS][MAX_FIELD_LENGTH]; // Array de
registros
    int field_count;          // Número de campos na tabela
    int record_count;        // Número de registros na tabela
} Table;

Table tables[10];           // Array de tabelas
int table_count = 0;        // Contador de tabelas

void show_table(Table *table); // Função para exibir dados de uma tabela
int execute_command(const char *input); // Função para executar comandos
Table* get_table_by_name(const char *name); // Função para buscar uma tabela por
nome

// Função que exibe a tabela e seus registros
void show_table(Table *table) {
    // Imprime o nome de cada campo
    for (int i = 0; i < table->field_count; i++) {
        printf("%s\t", table->fields[i].field_name);
    }
    printf("\n");

    // Imprime cada registro
    for (int i = 0; i < table->record_count; i++) {
        for (int j = 0; j < table->field_count; j++) {
            printf("%s\t", table->records[i][j]);
        }
        printf("\n");
    }
}

// Função para buscar uma tabela pelo nome
Table* get_table_by_name(const char *name) {

```

```

for (int i = 0; i < table_count; i++) {
    if (strcmp(tables[i].table_name, name) == 0) {
        return &tables[i]; // Retorna o ponteiro para a tabela
    }
}
return NULL; // Retorna NULL se a tabela não for encontrada
}

```

// Definição de tipos de tokens para análise léxica

```

typedef enum {
    TOKEN_EOF,
    TOKEN_CREATE,
    TOKEN_TABLE,
    TOKEN_INSERT,
    TOKEN_INT0,
    TOKEN_SELECT,
    TOKEN_FROM,
    TOKEN_STRING,
    TOKEN_NUMBER,
    TOKEN_COMMA,
    TOKEN_LPAREN, // (
    TOKEN_RPAREN, // )
    TOKEN_SEMICOLON,
    TOKEN_ASTERISK // *
} TokenType;

```

// Estrutura para definir um token (tipo e valor)

```

typedef struct {
    TokenType type;
    char value[MAX_FIELD_LENGTH];
} Token;

```

// Função que extrai o próximo token da entrada

```

Token get_token(const char **input) {
    Token token;
    token.type = TOKEN_EOF;
    token.value[0] = '\0';

    // Ignora espaços, tabulações e novas linhas
    while (**input == ' ' || **input == '\n' || **input == '\t' || **input == '\r') {
        (*input)++;
    }

    // Verifica palavras-chave e símbolos e atribui o tipo de token
    if (strncmp(*input, "CREATE", 6) == 0) {
        (*input) += 6;
        token.type = TOKEN_CREATE;
    } else if (strncmp(*input, "TABLE", 5) == 0) {
        (*input) += 5;
        token.type = TOKEN_TABLE;
    } else if (strncmp(*input, "INSERT", 6) == 0) {
        (*input) += 6;
        token.type = TOKEN_INSERT;
    } else if (strncmp(*input, "INTO", 4) == 0) {
        (*input) += 4;
        token.type = TOKEN_INTO;
    } else if (strncmp(*input, "SELECT", 6) == 0) {
        (*input) += 6;
        token.type = TOKEN_SELECT;
    } else if (strncmp(*input, "FROM", 4) == 0) {
        (*input) += 4;
        token.type = TOKEN_FROM;
    } else if (**input == ',') {
        (*input)++;
        token.type = TOKEN_COMMA;
    } else if (**input == '(') {

```

```

    (*input)++;
    token.type = TOKEN_LPAREN;
} else if (**input == ')') {
    (*input)++;
    token.type = TOKEN_RPAREN;
} else if (**input == ';') {
    (*input)++;
    token.type = TOKEN_SEMICOLON;
} else if (**input == '*') {
    (*input)++;
    token.type = TOKEN_ASTERISK;
} else if (isdigit(**input)) {
    int i = 0;
    while (isdigit(**input)) {
        token.value[i++] = *(*input)++;
    }
    token.value[i] = '\0';
    token.type = TOKEN_NUMBER;
} else if (isalpha(**input)) {
    int i = 0;
    while (isalnum(**input) || **input == '_') {
        token.value[i++] = *(*input)++;
    }
    token.value[i] = '\0';
    token.type = TOKEN_STRING;
}
return token;
}

```

```

// Função para validar o tipo de dado de um valor inserido
int validate_field_value(const char *value, char field_type) {
    if (field_type == 'N') { // Campo numérico
        for (int i = 0; value[i] != '\0'; i++) {

```

```

        if (!isdigit(value[i])) {
            return 0; // valor inválido
        }
    }
} else if (field_type == 'C') { // Campo de texto
    for (int i = 0; value[i] != '\0'; i++) {
        if (!isalpha(value[i])) { // Aceita apenas letras
            return 0; // valor inválido
        }
    }
}
return 1; // valor válido
}

```

// Função para processar o comando CREATE TABLE

```

int parse_create_table(const char **input) {
    Token token = get_token(input);
    if (token.type == TOKEN_STRING) {
        Table *table = &tables[table_count++];
        strcpy(table->table_name, token.value);
        table->field_count = 0;
        token = get_token(input);
        if (token.type == TOKEN_LPAREN) {
            int field_index = 0;
            do {
                token = get_token(input);
                if (token.type == TOKEN_STRING) {
                    strcpy(table->fields[field_index].field_name, token.value);
                    token = get_token(input);
                    if (strcmp(token.value, "C") == 0) {
                        table->fields[field_index].field_type = 'C';
                    } else if (strcmp(token.value, "N") == 0) {
                        table->fields[field_index].field_type = 'N';
                    }
                }
            } while (token.type != TOKEN_COMMA);
        }
    }
}

```



```

    } else {
        printf("Erro de sintaxe: Tipo de campo inválido.\n");
        return 1;
    }
    field_index++;
    table->field_count++;
}
token = get_token(input);
} while (token.type == TOKEN_COMMA);
if (token.type != TOKEN_RPAREN) {
    printf("Erro de sintaxe: Esperado ').\n");
    return 1;
}
} else {
    printf("Erro de sintaxe: Esperado '('\n");
    return 1;
}
} else {
    printf("Erro de sintaxe: Nome da tabela esperado.\n");
    return 1;
}
return 0;
}

```

// Função para processar o comando INSERT INTO, incluindo validação de tipo

```

int parse_insert_into(const char **input) {
    Token token = get_token(input);
    if (token.type == TOKEN_STRING) {
        Table *table = get_table_by_name(token.value);
        if (!table) {
            printf("Erro: Tabela '%s' não encontrada.\n", token.value);
            return 1;
        }
    }
}

```

```

token = get_token(input);
if (token.type == TOKEN_LPAREN) {
    int record_index = table->record_count;
    int field_index = 0;
    do {
        token = get_token(input);
        if (token.type == TOKEN_STRING || token.type == TOKEN_NUMBER) {
            // Valida o valor conforme o tipo do campo
            if (field_index < table->field_count &&
                !validate_field_value(token.value, table->fields[field_index].field_type))
        {
            printf("Erro de sintaxe: Tipo de valor inválido para o campo '%s'.\n",
                table->fields[field_index].field_name);
            return 1;
        }
        if (field_index < table->field_count) {
            strcpy(table->records[record_index][field_index], token.value);
            field_index++;
        }
    }
    token = get_token(input);
} while (token.type == TOKEN_COMMA);

table->record_count++;
printf("Registro inserido com sucesso! (Registro %d)\n", record_index + 1);
} else {
    printf("Erro de sintaxe: Esperado '('\n");
    return 1;
}
} else {
    printf("Erro de sintaxe: Nome da tabela esperado.\n");
    return 1;
}
}

```

```

    return 0;
}

// Função para processar o comando SELECT
int parse_select_from(const char **input) {
    Token token = get_token(input);
    if (token.type == TOKEN_ASTERISK) {
        token = get_token(input);
        if (token.type == TOKEN_FROM) {
            token = get_token(input);
            if (token.type == TOKEN_STRING) {
                Table *table = get_table_by_name(token.value);
                if (table) {
                    show_table(table);
                    token = get_token(input); // Verifica se há um ';' após SELECT
                    if (token.type != TOKEN_SEMICOLON && token.type != TOKEN_EOF) {
                        printf("Erro de sintaxe: Comando inesperado após SELECT.\n");
                        return 1;
                    }
                    return 0;
                } else {
                    printf("Erro: Tabela '%s' não encontrada.\n", token.value);
                    return 1;
                }
            } else {
                printf("Erro de sintaxe: Nome da tabela esperado após FROM.\n");
                return 1;
            }
        } else {
            printf("Erro de sintaxe: Esperado 'FROM'.\n");
            return 1;
        }
    } else {

```

```

        printf("Erro de sintaxe: Esperado '*'.\n");
        return 1;
    }
}

// Função que executa o comando baseado na entrada
int execute_command(const char *input) {
    const char *ptr = input;
    Token token = get_token(&ptr);
    if (token.type == TOKEN_CREATE) {
        token = get_token(&ptr);
        if (token.type == TOKEN_TABLE) {
            if (parse_create_table(&ptr) != 0) return 1;
            printf("Tabela criada com sucesso!\n");
        }
    } else if (token.type == TOKEN_INSERT) {
        token = get_token(&ptr);
        if (token.type == TOKEN_INT0) {
            if (parse_insert_into(&ptr) != 0) return 1;
        }
    } else if (token.type == TOKEN_SELECT) {
        if (parse_select_from(&ptr) != 0) return 1;
    } else {
        printf("Erro de sintaxe: Comando desconhecido.\n");
        return 1;
    }
    return 0;
}

// Função principal que lê o arquivo e executa os comandos
int main() {
    table_count = 0;

```

```

char filename[100];

printf("\nInterpretador DBASE\n");
printf("CMP1076 - COMPILADORES\n");
printf("Aluno: Leonardo de Moura Alves\n\n");

printf("Informe o nome do arquivo .txt com os comandos: ");
scanf("%99s", filename);

FILE *file = fopen(filename, "r");
if (!file) {
    printf("Erro ao abrir o arquivo.\n");
    return 1;
}

char line[256];
while (fgets(line, sizeof(line), file)) {
    // Remove novas linhas e espaços ao final
    line[strcspn(line, "\r\n")] = 0;

    // Ignora linhas vazias
    if (strlen(line) > 0) {
        if (execute_command(line) != 0) {
            printf("Interrompendo execução devido a erro.\n");
            fclose(file);
            return 1;
        }
    }
}

fclose(file);

printf("\nFinalizado\n");

```

```
    return 0;  
}
```

### **3 COMO UTILIZAR O CÓDIGO**

Neste capítulo será demonstrado como utilizar o código como interpretador da linguagem DBASE.

Para utilizar o código, o primeiro passo é instalar um compilador da linguagem C, sendo o GCC uma opção popular e amplamente utilizada.

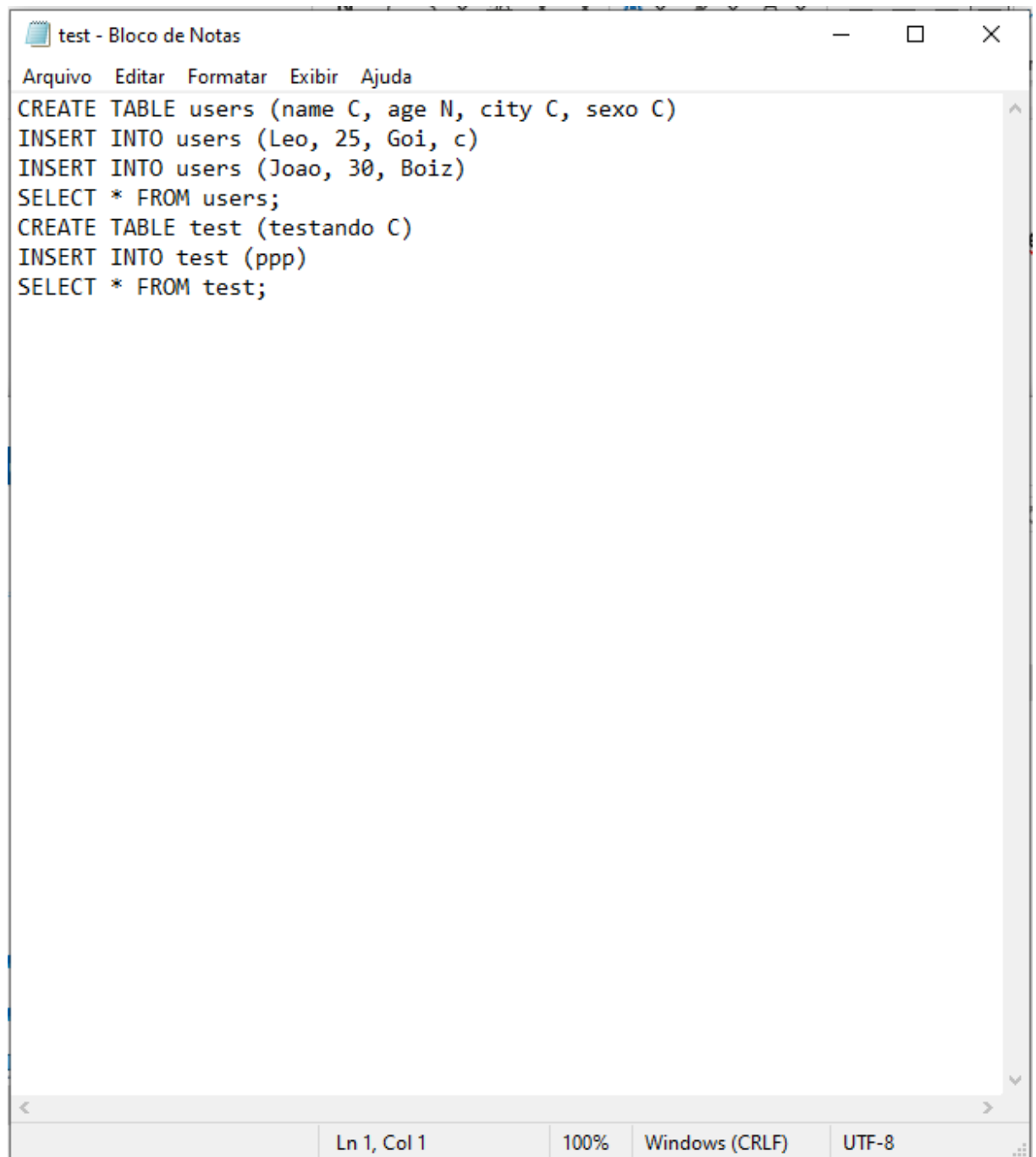
Após a instalação do GCC, você pode optar por utilizar uma IDE (Ambiente de Desenvolvimento Integrado) para facilitar o processo de codificação e execução do programa.

Neste caso, foi utilizado o uso do Visual Studio Code, que é leve e possui suporte a diversas extensões.

Para otimizar sua experiência, instale as extensões C/C++ e Code Runner, que permitirão compilar e executar seu código diretamente na IDE.

Primeiro é criar um documento TXT e colocar o código em DBASE.

Criamos o documento test.txt e colocamos os comandos.



```
test - Bloco de Notas
Arquivo  Editar  Formatar  Exibir  Ajuda
CREATE TABLE users (name C, age N, city C, sexo C)
INSERT INTO users (Leo, 25, Goi, c)
INSERT INTO users (Joao, 30, Boiz)
SELECT * FROM users;
CREATE TABLE test (testando C)
INSERT INTO test (ppp)
SELECT * FROM test;
Ln 1, Col 1    100%    Windows (CRLF)    UTF-8
```

Lembrando que o código permite somente os comandos CREATE TABLE, INSERT INTO e SELECT \* FROM.

Após salva o documento TXT dentro da pasta que está o código, basta compilar.

The screenshot shows the Visual Studio Code editor with the 'engine.c' source file open. The code defines a simple database engine with fields and tables. The terminal window at the bottom shows the command prompt with the following text:

```
PS C:\Users\Leonardo\Desktop\InterpretadorDBASEemC> cd "c:\Users\Leonardo\Desktop\InterpretadorDBASEemC\Source\" ; if ($?) { gcc engine.c -o engine } ;  
if ($?) { .engine }
```

Below the command prompt, the program's header information is displayed:

```
Interpretador DBASE  
OMP1876 - COMPILADORES  
Aluno: Leonardo de Moura Alves
```

The prompt asks the user to enter the name of the .txt file to use with the commands:

```
Informe o nome do arquivo .txt com os comandos:
```

Conforme demonstrado no terminal, será apresentado o cabeçalho do trabalho e solicitado ao usuário que informe o nome do arquivo TXT, no caso será informado “test.txt”.

This screenshot is similar to the previous one, but the terminal window now shows the command 'test.txt' entered after the prompt:

```
Informe o nome do arquivo .txt com os comandos: test.txt
```

Logo após é demonstrado os comandos sendo executado e informado os resultados dos comandos.



```
Source > C engine.c > ...
10 typedef struct {
11     char field_name[MAX_FIELD_LENGTH]; // Nome do campo
12     char field_type; // Tipo do campo (C ou N)
13 } Field;
14
15 typedef struct {
16     char table_name[MAX_FIELD_LENGTH]; // Nome da tabela
17     Field fields[MAX_FIELDS]; // Nomes das colunas e tipos
18     char records[MAX_RECORDS][MAX_FIELDS][MAX_FIELD_LENGTH]; // Dados inseridos
19     int field_count; // Quantidade de colunas
20     int record_count; // Quantidade de registros
21 } Table;
22
23 Table tables[10]; // Array de tabelas
24 int table_count = 0; // Contador de tabelas
25
26 void show_table(Table *table);
27 int execute_command(const char *input);
28 Table* get_table_by_name(const char *name);
29
30 void show_table(Table *table) {
31     for (int i = 0; i < table->field_count; i++) {
32         printf("%s\t", table->fields[i].field_name);
33     }
34     printf("\n");
35 }
```

Informe o nome do arquivo .txt com os comandos: test.txt  
Tabela criada com sucesso!  
Registro inserido com sucesso! (Registro 2)  
name age city sexo  
Leo 25 Gol c  
Joao 30 Boiz  
Tabela criada com sucesso!  
Registro inserido com sucesso! (Registro 1)  
testando  
ppp  
Finalizado

No caso do exemplo, primeiro solicitamos a criação da tabela user, logo após a inserção de dois registros na tabela user e a impressão da tabela.

Em seguida criamos a tabela test e inserimos um elemento dentro dela e solicitamos a impressão, conforme demonstrado no exemplo.

Caso alteremos a primeira linha e erramos propositalmente a sintaxe, o código reconheceria e informaria que a sintaxe está errada e finalizaria o código.

```
PS C:\Users\Leonardo\Desktop\InterpretadorDBASEemC\Source>
PS C:\Users\Leonardo\Desktop\InterpretadorDBASEemC\Source> cd "C:\Users\Leonardo\Desktop\InterpretadorDBASEemC\Source\" ; if ($?) { gcc engine.c -o engi
ne } ; if ($?) { . ./engine }

Interpretador DBASE
CMP1076 - COMPILADORES
Aluno: Leonardo de Moura Alves

Informe o nome do arquivo .txt com os comandos: test.txt
Erro de sintaxe: Comando desconhecido.
Interrompendo execução devido a erro.
PS C:\Users\Leonardo\Desktop\InterpretadorDBASEemC\Source>
```

## **4 CONCLUSÃO**

O código construído é um interpretador básico para DBASE, permitindo criar tabelas, inserir registros e selecionar e imprimir dados. Cada componente é cuidadosamente projetado para lidar com as operações fundamentais do DBASE, enquanto valida entradas e gerência tabelas e registros em memória.