

**Pró-Reitoria Acadêmica
Escola de Educação, Tecnologia e Comunicação
Curso de Bacharelado em Engenharia de Software**

Trabalho de Algoritmos e Programação Estruturada

**LIMITES NUMÉRICOS PARA VARIÁVEIS EM LINGUAGEM C
ATRAVÉS DA DECODIFICAÇÃO DE STRINGS HEXADECIMAIS**

Autores: Gabriel Alberto Duarte de Andrade
Juliano Arthur da Silva Braga
Natanael Ferreira Neves
Thiago Antônio Vieira
Wagner Serpa Porto

Orientador: Prof. Me. Jefferson Salomão Rodrigues

Gabriel Alberto Duarte de Andrade

Juliano Arthur da Silva Braga

Natanael Ferreira Neves

Thiago Antônio Vieira

Wagner Serpa Porto

**LIMITES NUMÉRICOS PARA VARIÁVEIS EM LINGUAGEM C ATRAVÉS DA
DECODIFICAÇÃO DE STRINGS HEXADECIMAIS**

Documento apresentado ao Curso de graduação de Bacharelado em Engenharia de Software da Universidade Católica de Brasília, como requisito parcial para obtenção da aprovação na disciplina de Algoritmos e Programação Estruturada.

Orientador: Prof. Me. Jefferson Salomão Rodrigues

**Brasília
2024**

RESUMO

As variáveis na linguagem C são elementos indispensáveis, pois servem como espaços de armazenamento que permitem a manipulação de valores ou dados, facilitando assim a organização de informações de diferentes tipos, como inteiros, caracteres e pontos flutuantes, sendo assim como há diversos tipos de dados, há também diversos tipos de variáveis, onde que um dos erros mais comuns na programação em linguagem C, é o estouro de “*buffer*” ou estouro de memória, é quando este problema envolve cálculos matemáticos esse problema é conhecido como “*overflow*”, no qual estes problemas ocorrem quando as manipulações de dados, transferências, valores ou operações matemáticas, excedem a capacidade máxima de armazenamento permitida pela variável podendo gerar um valor equivocado trazendo erros ao código ou programa, devido a isso conhecer as variáveis e seu limites é fator fundamental para todo profissional que trabalhe com programação em linguagem C, onde que neste presente trabalho será demonstrado o conceito de limites de variáveis e também de mecanismos de repetição através da decodificação de strings em formato hexadecimal.

Palavras-chave: Código, Variáveis, Armazenamento, Dados, *Overflow*.

ABSTRACT

Variables in the C language are essential elements, as they serve as storage spaces that allow the manipulation of values or data, thereby facilitating the organization of information of different types, such as integers, characters, and floating-point numbers. Just as there are various data types, there are also different types of variables. One of the most common errors in C programming is buffer overflow, which occurs when issues related to mathematical calculations arise; this problem is known as "overflow." These issues occur when data manipulations, transfers, values, or mathematical operations exceed the maximum storage capacity allowed by the variable, potentially generating incorrect values and causing errors in the code or program. Therefore, understanding variables and their limits is a fundamental factor for every professional working with C programming. This paper will demonstrate the concept of variable limits as well as mechanisms of repetition through the decoding of strings in hexadecimal format.

Keywords: Code, Variables, Storage, Data, Overflow.

LISTA DE FIGURAS

Figura 1. Limites em constantes de inteiro.....	13
Figura 2 - Sufixo Long Long Int e Unsigned Long Long Int.....	15
Figura 3 - Termo especificador Long Long	15
Figura 4 - Termo especificador Unsigned Long Long Int.....	16
Figura 5 - Termo especificador não compatível	16
Figura 6 - Declaração de tipo incompatível.	16
Figura 7 - Código para impressão da tabela ASCII em linguagem C.	18
Figura 8 - Strings fornecidas para elaboração do código de decodificação.	20
Figura 9 - Equação padrão para validação dos caracteres das strings.	20
Figura 10 – Strings fornecidas para elaboração do código de decodificação com os valores da variáveis x e b, destacados.....	21
Figura 11 - Localização do fim da mensagem.	21
Figura 12 - Código em linguagem C, para separação em duplas de dois algarismos ou letras das strings fornecidas anteriormente.....	22
Figura 13 - Tabela de verificação do 1º caso de teste e da equação de validação e da mensagem fornecida pela 1º string.	23
Figura 14 - Link do repositório na plataforma de hospedagem GitHub.....	24
Figura 15 - Localização dos códigos na plataforma GitHub.....	24
Figura 16 - Visão geral do código “funçãoondas”.	25
Figura 17. Bibliotecas incluídas para construir o código “funçãoondas”.	25
Figura 18 - Código “funçãoondas” ajustado.	26
Figura 19 - Código “funçãoondas”, equação detalhada	27
Figura 20- Código “funçãoondas”, detalhe das funções “pow” e “round”.	27
Figura 21- Código “funçãoondas”, detalhe do resultado da equação.....	28
Figura 22- Código “funçãoondas”.....	28
Figura 23- Visão geral do código “decodificador”.	29
Figura 24- Detalhe da função “printf” presente no código “decodificador”	29
Figura 25– Detalhe das funções “strtol” e “strcmp” presente no código “decodificador”.....	30
Figura 26 - Detalhe da função “strlen” presente no código “decodificador”.	30
Figura 27 - Detalhe da inclusão das bibliotecas e do código “funçãoondas” presente no código “decodificador”..	30
Figura 28 - Detalhe decodificação da string presente no código “decodificador”.....	31
Figura 29 - Detalhe da inclusão do código “decodificador”, no código “executar”..	31
Figura 30 - Visão geral do código “executar”.....	32
Figura 31 - Executando o código..	32
Figura 32 - Executando o código inserindo a quantidade de testes.....	33
Figura 33 - Executando o código inserindo a variável “b” e a string fornecida.....	34
Figura 34 - Executando o código exibição da string decodificada.....	34
Figura 35 - Executando o código exibição da string decodificada.....	34
Figura 36 - Executando o código exibição da string decodificada.....	34

SUMÁRIO

RESUMO	3
ABSTRACT	4
LISTA DE FIGURAS	5
1 INTRODUÇÃO.....	7
2 COMPILAÇÃO	8
3 BIBLIOTECAS.....	9
4 VARIÁVEIS.....	10
4.1 VARIÁVEL INT.....	12
4.2 VARIÁVEL CHAR	13
4.3 VARIÁVEIS LONG LONG INT E UNSIGNED LONG LONG INT.....	13
4.4 VARIÁVEL SHORT INT	17
4.5 VARIÁVEL LONG INT	17
5 HEXADECIMAIS.....	17
6 TABELA ACSII.....	18
7 OBJETIVOS	19
7.1 GERAL	19
7.2 ESPECÍFICOS.....	19
8 MATERIAIS E MÉTODOS	20
9 RESULTADOS E DISCUSSÃO	25
9.1 CÓDIGO “FUNÇÃOONDAS”.....	25
9.2 CÓDIGO “DECODIFICADOR”	29
9.3 CÓDIGO “EXECUTAR”	31
10 CONCLUSÃO.....	35
REFERÊNCIAS	36
APÊNDICE A – CODIGO FUNÇÃO ONDAS.....	38
APÊNDICE B – CODIGO DECODIFICADOR.....	39
APÊNDICE C – CODIGO EXECUTA.....	40

1 INTRODUÇÃO

Nesta secção introdutória será apresentado um breve histórico, algumas características, particularidades e conceitos da linguagem C, as quais são importantes para o entendimento do conceito de variáveis que será apresentado ao logo deste presente trabalho, sendo assim a linguagem C é uma das linguagens de programação mais famosas ou se não a mais famosa entre os programadores, pois em grande parte das vezes é a primeira linguagem onde se tem o primeiro contato com a programação, onde que a mesma foi criada em 1972 nos *Bell Telephone Laboratories*, localizado em Murray Hill nos Estados Unidos, pelo cientista da computação Dennis MacAlistair Ritchie (1941 – 2011), tendo a finalidade de permitir uma escrita operacional que seria utilizada na codificação da segunda versão do sistema operacional UNIX, sendo assim utilizando uma linguagem com um nível relativamente alto para evitar o uso da linguagem Assembly que fora utilizada para desenvolver a primeira versão do mesmo sistema operacional para o computador DEC PDP-11.

Algumas bibliografias consideram a linguagem C como de alto nível devido as suas variáveis, loops e funções, mas, no entanto, outras bibliografias a considera como uma linguagem de baixo nível devido a sua sintaxe ser próxima ao hardware do computador onde que as instruções realizadas com essa linguagem são executadas pelo mesmo de maneira mais fácil em relação a outras linguagens de programação devido a mesma ser projetada com o intuito de se desenvolver sistemas, ao contrário do Python que é uma linguagem de alto nível que possui uma sintaxe abstrata e de mais fácil entendimento para o ser humano (BRUNNER, T., & PORKOLÁB, Z., 2017).

O ponto forte da linguagem C, de acordo com Backes, (2023), é que a mesma se trata de uma linguagem estruturalmente simples e de grande portabilidade tendo várias bibliotecas para diversos usos, onde que esse conceito esse que será apresentado nos próximos tópicos. Para a linguagem C são poucas as arquiteturas de computadores para as quais não exista um compilador, onde que o compilador da linguagem C permite gerar códigos mais enxutos e velozes do que muitas outras linguagens, permitindo assim que os programadores desenvolvam códigos otimizados para o desempenho dos computadores, pois a sua sintaxe próxima ao nível de máquina não gera a mesma sobrecarga no computador como outras linguagens de alto nível permitindo assim uma execução mais rápida do mesmo, pois a mesma fornece um baixo nível de acesso a memória, permitindo o acesso e a programação direta do microprocessador, sendo assim é possível desenvolver programas com características de linguagens de alto nível e ao

mesmo tempo trabalhar em um nível muito próximo da linguagem de máquina gerando produtos que possuem um desempenho mais eficiente (RICARTE, 2001).

Devido a linguagem C ter sido desenvolvida para incentivar a programação em diversas plataformas com o intuito de se desenvolver sistemas e até mesmo outras linguagens de programação, os códigos concebidos em C podem ser compilados para uma grande variedade de plataformas e sistemas operacionais com algumas pequenas alterações em seu código-fonte (BACKES 2024).

Quando se fala em código-fonte de um programa é a relação entre um conjunto de símbolos e palavras no qual estão descritas as instruções do que o programa deve fazer ao ser executado, onde que o código-fonte é normalmente escrito de uma forma que facilite a compreensão pelos seres humanos (MANZANO, 2002). Onde para que o código fonte tenha algum significado para o computador, o mesmo precisa ser traduzido para o código de máquina, onde que esse processo de tradução é denominado de compilação no qual é o assunto do próximo tópico apresentando neste presente trabalho.

2 COMPILAÇÃO

A compilação em programação estruturada é realizada pelos compiladores que são componentes que traduzem o código-fonte que foi escrito em uma linguagem de programação para o código de máquina, onde que muitos acreditam que a compilação através dos compiladores é apenas uma tradução, mas a mesma se trata de um conjunto de etapas que devem ser seguidas.

Sendo assim a compilação tem o objetivo de facilitar a execução e a otimização do desempenho do programa ou código por parte do computador onde que também desempenha um papel importante na detecção de erros no código e auxilia na implementação de boas práticas de programação (FREITAS, 2019).

Este processo de tradução permite o desenvolvimento de programas independentemente da máquina que está sendo utilizada pois o mesmo código pode ser compilado em diversos computadores ou máquinas, onde que outra tarefa importante que o compilador executa é a identificação dos erros sintáticos e semânticos na estrutura do código no ciclo de desenvolvimento, com o intuito de reduzir os riscos que podem trazer falhas a execução inicial do programa, sendo assim mantendo a integridade e a eficiência do código quando executado pelo computador (ANDRADE, 2017).

O compilador além de impor a adesão aos padrões de codificação e às práticas recomendadas, os mesmos suportam a capacidade de manutenção e escalabilidade das bases de código, onde que a união entre os princípios de programação estruturada e as técnicas de compilação permite o desenvolvimento sistemático e eficiente de softwares e programas, fazendo com que os compiladores sejam elementos essenciais na otimização do desempenho e na proteção da qualidade do código (BACKERS, 2024).

Dentre os compiladores para a linguagem C destaca a coleção de compiladores “GNU Compiler Collection” ou GNU, conhecidos também como GCC, onde se trata de uma ferramenta especializada no desenvolvimento de software, disponibilizada no regime de software livre, para diversos sistemas operacionais, se tratando de um ambiente de programação que contém *front-ends* para diversas linguagens para computadores, sendo um dos poucos compiladores que respeitam os padrões de normatização da “*International Organization for Standardization*”, ISO, principalmente para as linguagens C e C++, sendo assim devido a essa peculiaridade é o compilador de linguagem C e C++ mais utilizado e respeitado para essas linguagens (MANZANNO 2002; FREITAS 2019).

3 BIBLIOTECAS

Como foi visto no tópico anterior, os compiladores são elementos essenciais na otimização do desempenho e na proteção da qualidade do código, mas para que isso ocorra na linguagem C temos os elementos essenciais para se programar na mesma, no qual esses elementos são denominados de biblioteca.

O conceito inicial de biblioteca na linguagem C, é de conjuntos de funções e rotinas pré-definidas que auxiliam o desenvolvimento de softwares ou programas a partir da permissão de reutilização de código, onde consequentemente promovem a padronização e a eficiência da programação sendo um elemento essencial para a construção de aplicações robustas e eficientes (AISAWA, 2020).

Em um contexto geral as bibliotecas presentes na linguagem C são coleções de funções e rotinas predefinidas que são semelhantes a blocos de construção, onde permitem que os programadores simplifiquem seu fluxo de trabalho reutilizando o código, sendo assim uma prática que reduz significativamente o tempo de desenvolvimento de programas e softwares (DE OLIVEIRA; MOLGORA 2014).

Como já foi dito anteriormente que as bibliotecas servem como coleções de funções e rotinas predefinidas que abrangem operações complexas em trechos de código que são mais simples, enxutos e reutilizáveis, fazendo com que se acelere significativamente o processo de codificação, fazendo com que a tarefa de compilação que é uma atividade que requer uma certa intensidade poder computacional, possa ser aprimorada e ser eficientemente tratada através de bibliotecas especializadas para qual programa está sendo desenvolvido (LIDANI; BORBA, 1999), no qual um exemplo claro é a biblioteca “math.h”, onde que a mesma fornece um conjunto de funções matemáticas que simplificam a implementação de cálculos numéricos complexos por parte do programador, fornecendo consistência e confiabilidade geral dos cálculos realizados pelo computador, exigindo menos do mesmo e facilitando o processo de compilação por parte do compilador (DE OLIVEIRA; MOLGORA 2014).

A vantagem disso não é apenas a economia de tempo de desenvolvimento, mas também a padronização em diferentes projetos, fazendo com que seja preservada a padronização da codificação que os programadores estão utilizando durante o projetos, como também a liberdade de se realizar tarefas complexas, sendo assim, as bibliotecas são ferramentas essenciais para a construção de softwares e programas de alta qualidade, no qual dentro da linguagem C, as bibliotecas são pilares essenciais para se iniciar qualquer algoritmo, sendo um elemento fundamental para o desenvolvimento de software (AISAWA, 2024).

4 VARIÁVEIS

Neste tópico discutiremos o que são variáveis na linguagem C, sendo que neste tópico será apresentado a importância do uso e os limites das variáveis, pois a partir desta compreensão se terá entendimento do objetivo proposto neste presente trabalho.

Como foi visto anteriormente a linguagem C possui bibliotecas que são conjuntos de funções e rotinas pré-definidas, mas para que essas funções tenham alguma funcionalidade, as mesmas devem possuir valores ou dados atribuídos, sendo assim necessitando de variáveis para tal atribuição (SCHILDT, 1997). As variáveis na linguagem C são elementos indispensáveis, pois servem como espaços de armazenamento que permitem a manipulação de valores ou dados, facilitando assim a organização de informações de diferentes tipos, como inteiros, caracteres e pontos flutuantes, sendo assim como a diversos tipos de dados há também diversos tipos de variáveis que permitem que o programador manipule esses valores ou dados com precisão (COELHO, 2022).

Embora o conceito de variáveis de uma forma inicial seja aparentemente simples, quando se é realizada uma análise mais profunda, percebe-se as complexas maneiras pelas quais as variáveis contribuem para práticas de programação e desenvolvimento de software, sendo assim, entender o uso e os limites das variáveis em C é fundamental para o uso e desenvolvimento de técnicas de programação que gerem produtos com desempenho satisfatórios (SCHILDT, 1997).

Um dos erros mais comuns na programação em C, é o estouro de “*buffer*” ou estouro de memória, é quando este problema envolve cálculos matemáticos esse problema é conhecido como “*overflow*”, no qual estes problemas ocorrem quando as manipulações de dados, transferências, valores ou operações matemáticas, excedem a capacidade máxima de armazenamento permitida pela variável gerando um valor diferente, podendo levar à substituição de locais de memória próximos, no qual essa vulnerabilidade é particularmente perigosa devido ao seu potencial de exploração quando o programa apresenta erro, permitindo que agentes mal-intencionados executem código de maneira arbitrária para causar falhas no programa, comprometendo a integridade do mesmo, onde que este assunto é de grande relevância para ambientes de computação, onde os recursos são finitos e precisamente pré-definidos (DIETZ, W. et al, 2015).

Esses problemas estão associados há uma falta de gerenciamento de dados ou valores e boas práticas de gerenciamento de memória como também a falta de conhecimento por parte do programador das variáveis disponíveis para a linguagem, enfatizando a necessidade de os desenvolvedores conhecerem os tipos de variáveis existentes e saber diferenciá-las, como também ter conhecimento sobre seus limites, associando ferramentas e técnicas para reduzir possíveis erros e vulnerabilidades no código (SHAW; DOGETT; HAFIZ 2014).

De acordo com Zhang et al (2021), os pontos fracos do código são identificados em apenas 2% das respostas apresentadas para as linguagens C e C++ em plataformas como o site “*Stack Overflow*”, refletindo problemas na estrutura das práticas de programação quando realizada de maneira compartilhada.

Com base na necessidade mencionada anteriormente de se gerenciar meticulosamente a memória para que o programa de respostas confiáveis e seja imune a erros, entender os limites das variáveis em C torna-se algo imprescindível devido ao sua influência direta na otimização do uso da memória e na melhoria do desempenho do programa, onde que o conceito de limites

de variáveis abrange o intervalo e o tamanho do armazenamento de vários tipos de dados na linguagem C, sendo fator determinante para a eficiência com que os dados podem ser armazenados e manipulados em um programa. Por exemplo, utilizar uma variável do tipo “char” para armazenar um dado que poderia ser armazenado em uma variável do tipo “int” seria suficiente, mas, no entanto, seria um desperdício de espaço de memória, o que geraria uma sobrecarga de processamento por parte do computador (OUALLINE, 1997).

De acordo com Kernighan, Ritchie (2002), dominar os limites das variáveis presentes na linguagem C não é apenas uma necessidade técnica, mas uma habilidade fundamental que todo programador que trabalha e desenvolve nesta linguagem deve ter o conhecimento, pois as variáveis influenciam diretamente na eficiência e a confiabilidade do software, onde que a prevenção de erros de estouro de memória é são primordiais, onde que seguindo com essa afirmação nos próximos tópicos que seguem serão apresentados alguns tipos de variáveis que foram utilizadas para a elaboração do código neste presente trabalho.

4.1 VARIÁVEL INT

Variável int em C de acordo com o Microsoft Ignite (2024), o ‘int’ é usado para armazenar números inteiros. Ele representa valores numéricos em diversas faixas, dependendo do sistema e do compilador. O “int” é utilizado para armazenar números inteiros em um programa C, sendo amplamente usado em operações matemáticas, contagens, *loopings* e representações de números inteiros onde não há necessidade de parte fracionária. A variável “int” ocupa normalmente 4 bytes de memória em sistemas de 32 ou 64 bits e pode armazenar tanto números com sinal quanto sem sinal, dependendo da arquitetura da máquina e do compilador. Quando uma variável “int” é declarada, o sistema aloca um bloco de memória de 4 bytes e armazena o valor da variável em forma binária. Em sistemas de 32 bits, um “*signed int*” tem um limite de -2.147.483.648 a 2.147.483.647, enquanto um “*unsigned int*” pode armazenar valores de 0 a 4.294.967.295, utilizando todos os 32 bits para representar valores positivos.

As implicações do uso de ‘int’ incluem a possibilidade de *overflow* ou *underflow* ao exceder os limites permitidos. Por exemplo, tentar armazenar 2.147.483.648 em um signed int resulta em *overflow*, podendo causar comportamento inesperado. Em sistemas com signed int, tentar armazenar um número muito negativo resulta em *underflow*. Para mais informações sobre o funcionamento dos tipos de dados (MICROSOFT, 2024).

4.2 VARIÁVEL CHAR

Variável char em C de acordo com Kernighan, Ritchie (1998), foi projetada para armazenar caracteres, como letras e símbolos, mas tecnicamente é um tipo numérico de 1 byte que armazena o código numérico associado ao caractere (geralmente o valor ASCII). O `char` é normalmente utilizado para armazenar um único caractere (como 'A', 'B', '1', '#') ou para criar strings, além de poder armazenar números pequenos em situações em que a economia de memória é importante. O “char” ocupa 1 byte de memória, o que corresponde a 8 bits. Se a variável for *signed*, o primeiro bit é usado como o bit de sinal (0 para positivo, 1 para negativo), enquanto se for *unsigned*, todos os 8 bits são usados para armazenar o valor numérico.

Um *signed char* pode armazenar valores de -128 a 127, enquanto um *unsigned char* pode armazenar valores de 0 a 255. Como o char é muito limitado em termos de faixa de valores, ele é adequado apenas para números pequenos ou caracteres. Tentativas de armazenar valores fora dos limites resultarão em *overflow* ou *underflow*, causando comportamentos indesejados (KERNIGHAN; RITCHIE, 1998).

4.3 VARIÁVEIS LONG LONG INT E UNSIGNED LONG LONG INT

De acordo com Microsoft, (2022) através da sua documentação oficial, a variável *Long Long int*, possuindo 8 bytes (64 bits) é o dobro do *Long int*. Quando precedido do termo “*unsigned*” que é um modificador para determinar que o tipo numérico inteiro é sem sinal, ou seja, passaremos a ter somente valores positivos nele. Assim sendo, ele permite o dobro dos valores (18446744073709551615).

Se usar apenas “*long long*”, sem especificar mais nada, o “Signed int” é assumido. Tornando-se assim Signed Long Long Int, ou somente *Long Long Int*, onde que a figura 1 que se encontra abaixo apresenta os limites em constantes de inteiro para variável *long long int*.

Figura 1 - Limites em constantes de inteiro

LLONG_MIN	Valor mínimo para uma variável do tipo <i>long long</i>	-9223372036854775808
LLONG_MAX	Valor máximo para uma variável do tipo <i>long long</i>	9223372036854775807
ULLONG_MAX	“Valor máximo para uma variável do tipo <i>unsigned long long</i> ”	18446744073709551615 (0xffffffffffffffff)

Fonte: Limites de inteiro Microsoft Learn 2024

Ainda sobre o quadro acima limites em constantes de inteiro, em “*long long*”, lê-se nove trilhões, duzentos e vinte e três trilhões, trezentos e setenta e dois bilhões, trinta e seis milhões, oitocentos e cinquenta e quatro mil, setecentos e setenta e cinco mil, oitocentos e sete, positivo ou negativo. E em “*unsigned*”, lê-se, dezoito quintilhões, quatrocentos e quarenta e seis quatrilhões, setecentos e quarenta e quatro trilhões, setenta e três bilhões, setecentos e nove milhões, quinhentos e cinquenta e um mil, seiscentos e quinze.

Tendo por base isso, operações com a variável “*long long*” e “*unsigned long long*” se tornam demasiadamente extravagantes e desnecessárias no cotidiano visto que, trabalhar com números longos pode significar trabalhar com o dobro de processamento para cálculos (MICROSOFT, 2022).

O uso mais frequente dessas variáveis pode ser encontrado na contagem de eventos em sistemas operacionais, bases de dados e simulações, esse número pode ser usado para contar o número de ocorrências de um evento, também em espaços de endereçamento em sistemas de memória, onde que esse número pode representar o endereço máximo de uma célula de memória, também em criptografia em alguns algoritmos criptográficos, onde que esse número pode ser utilizado como um limite superior para chaves de criptografia (DE AZEVEDO; KONZEN; SAUTER, 2020).

E a seguir, temos um exemplo abaixo que demonstra quão desgastante pode se tornar operações com cálculos que utilizam tais tipos de números.

- Exemplo:

Tente executar um “laço for” simples, que imprima na tela os números de 1 a 1 bilhão.

```
for (i = 0; i <= 1000000000; i++){ printf("%d\n", i);}
```

Imagine o dobro do primeiro valor

```
for (i = 0; i <= 2000000000; i++){ printf("%d\n", i);}
```

Imagine o triplo do primeiro valor

```
for (i = 0; i <= 3000000000; i++){ printf("%d\n", i);}
```

Nas figuras que se seguem (2 a 6), será demonstrado modelos de sintaxe e declarações para os tipos “*Long Long Int*” e “*Unsigned Long Long Int*”.

Figura 2 - Sufixo *Long Long Int* e *Unsigned Long Long Int*

```
//long long equivale a signed long long int ou simplesmente long long int

//Limite Máximo Positivo do Long Long
long long ll_max = 9223372036854775807LL;
signed long long sll_max = ll_max;
signed long long int sllint_max = ll_max;

//Limite Mínimo Negativo do Long Long
long long ll_min = -9223372036854775808LL;
signed long long sll_min = ll_min;
signed long long int sllint_min = ll_min;
```

Fonte: Autores

Na figura 3 que está abaixo, o sufixo LL é necessário para especificar que o número é longo, onde que para apresentar um “*long long*”, o termo especificador deverá ser LLD. Onde esse tipo é especialmente útil para cálculos que envolvem grandes intervalos, como em algoritmos que processam grandes conjuntos de dados ou trabalham com valores financeiros (MICROSOFT INGINITE, 2024).

Figura 3 - Termo especificador *Long Long*

```
//
printf("ll_max -> %lld\n", ll_max);
printf("sll_max -> %lld\n", sll_max);
printf("sllint_max -> %lld\n", sllint_max);

printf("ll_min -> %lld\n", ll_min);
printf("sll_min -> %lld\n", sll_min);
printf("sllint_min -> %lld\n", sllint_min);
```

Fonte: Autores

Para uso do “*Unsigned Long Long Int*”, a sintaxe, segue a mesma mecânica, todavia, o termo especificador torna-se “*LLU*”, conforme figura 4 que se encontra abaixo.

Figura 4 - Termo especificador *Unsigned Long Long* Int

```

unsigned long long int ullint_max = 18446744073709551615LL;
long long int llint_max = 18446744073709551615LL;
long long ll_max = 18446744073709551615LL;

printf("%llu\n", ullint_max);
printf("%llu\n", llint_max);
printf("%llu\n", ll_max);

```

Fonte: Autores

E por fim, nas figuras a seguir 5 e 6 que estão abaixo, demonstra-se através de comentário em linha, possíveis erros apresentados, quando há declaração errada.

Figura 5 - Termo especificador não compatível.

```

long long ll_max = 9223372036854775807LL; //declaração correta
long long ll_min = -9223372036854775808LL;

printf("ll_max -> %d\n", ll_max); //Extrapolando o máximo de INT retorna -1
printf("ll_min -> %d\n", ll_min); //Extrapolando o mínimo de INT retorna 0

```

Fonte: Autores

Figura 6 - Declaração de tipo incompatível

```

long long ll_max = 9223372036854775807; //declaração sem sufixo,
funciona mas não recomendado;

int ll_min = -9223372036854775808LL; // Torna o resultado como NULL

long int ullint_max = 18446744073709551615LL; //Apresentar o valor
máximo do tipo int (4294967295)

printf("ll_max -> %lld\n", ll_max); //Extrapolando o máximo de INT retorna -
1

printf("ll_min -> %lld\n", ll_min); //Extrapolando o mínimo de INT retorna 0

```

Fonte: Autores.

Esse erro de declaração pode trazer problemas na execução do código, como também fazer que a compilação seja bloqueada pelo sistema operacional.

4.4 VARIÁVEL SHORT INT

Variável *short int* em C é de maneira similar a variável *int*, sendo capaz de armazenar números inteiros, com a diferença que *short int* possui uma capacidade de armazenamento consideravelmente menor, sendo capaz de armazenar valores de -32.768 a 32.767 com o uso de 16 bits. (ALMEIDA, 2001).

Por ter uma capacidade de armazenamento reduzida a variável *short int* requer um uso de memória menor do que *int*, utilizando apenas 16 bits ao invés de 32. Isso a torna uma melhor alternativa quando se faz necessário o uso de uma capacidade de armazenamento muito grande e se busca uma maior economia de memória e um melhor desempenho. (ALMEIDA, 2001).

4.5 VARIÁVEL LONG INT

Ao contrário da variável *short int* a variável *long int* possui uma capacidade de armazenamento consideravelmente maior que *int*, sendo capaz de armazenar valores de aprox. -9×10 elevado à 18 a aprox. 9×10 elevado à 18, porém faz se necessário o uso de 64 bits. (ALMEIDA, 2001).

Por requerer o dobro da quantidade de memória que *int*, a variável *long int* é utilizada em casos em que se faça necessário o uso de uma grande quantidade de armazenamento de dados, sendo útil para evitar *overflow*. (ALMEIDA, 2001).

5 HEXADECIMAIS

Os números hexadecimais são uma base numérica que utiliza 16 símbolos para representar valores, abrangendo 0-9 e A-F, e são amplamente utilizados em programação e computação, pois oferecem uma forma compacta de representar dados binários, facilitando a leitura e a manipulação. Em notação hexadecimal, um número é precedido por "0x" ou "0X", como em 0x1A, que representa o valor decimal 26.

Eles são utilizados em várias áreas, como representação de cores no desenvolvimento web (ex.: #FF5733), endereçamento de memória em linguagens de baixo nível como *Assembly*, representação de dados binários, onde cada dígito hexadecimal representa 4 bits, e na programação de sistemas operacionais e drivers. A conversão entre sistemas numéricos é essencial, por exemplo, ao converter decimal para hexadecimal, como no caso de 255, que se divide por 16, resultando em 0xFF. As operações aritméticas com números hexadecimais são semelhantes às do sistema decimal, como demonstrado no código onde 0xA e 0x5 resultam em

0xF, e operações bit a bit, como AND, são realizadas para manipular bits individuais (KERNIGHAN; RITCHIE, 1988; ISO/IEC, 2011; PEEK; THORPE, 2005).

Os limites para números hexadecimais variam conforme a implementação da linguagem, como um `unsigned int` de 32 bits que pode representar valores de 0 até 0xFFFFFFFF (4.294.967.295 em decimal). No entanto, a mistura de notações pode causar ambiguidade, e erros de conversão podem ocorrer se não forem tratados adequadamente. Embora hexadecimais sejam mais compactos que binários, podem ser menos intuitivos para programadores menos experientes. Em conclusão, os números hexadecimais são fundamentais na programação, permitindo uma representação clara e compacta de dados binários, e compreender sua conversão, uso e limitações é crucial para o desenvolvimento eficaz de software (LINUX FOUNDATION; LAFORE, 2002).

6 TABELA ACSII

A tabela ASCII (*American Standard Code for Information Interchange*) foi criada em 1963 com o intuito de padronizar os caracteres propostos pelo ANSI (*American National Standards Institute*), onde que a mesma é utilizada para a organização dos caracteres nos softwares e computadores (BAHIANA, 2009).

A tabela ASCII possui 128 símbolos, cada um ocupando espaço de 1 byte, o que confere portabilidade e a capacidade de criar arquivos menores. Apesar disso, a tabela ASCII não possui diversos caracteres de outras línguas como o português, faltando as letras acentuadas e a cedilha, sendo necessário o uso da tabela ASCII estendida para representar estes símbolos. A tabela ASCII estendida adiciona mais 128 caracteres, sendo esses caracteres de outras línguas, caracteres acentuados e símbolos gráficos, sendo assim totalizando 256 caracteres no total, onde que a figura 7 que está abaixo demonstra o código para impressão dos caracteres em linguagem C (BAHIANA, 2009).

Figura 7 – Código para impressão da tabela ASCII em linguagem C

```
#include <stdio.h>

int main() {
    printf("Tabela ASCII:\n");
    printf("Codigo\tCaractere\n");
    printf("-----\n");

    for (int i = 0; i < 128; i++) {
        printf("%d\t%c\n", i, (char)i);
    }

    return 0;
}
```

Fonte: Autores

7 OBJETIVOS

7.1 GERAL

O objetivo geral desta presente pesquisa é de conceber um código em linguagem C, para se realizar a decodificação de quatro “strings” propostas em sala de aula, sendo assim cada das strings apresenta uma mensagem em formato hexadecimal que deve ser decodificada uma mensagem em escrita alfabética utilizando a tabela ACSII estendida, com o objetivo de fixar o conhecimento sobre limites numéricos das variáveis presentes na linguagem programação C.

7.2 ESPECÍFICOS

A presente pesquisa possui o objetivo específico de se aplicar o conhecimento adquirido em sala relacionado a linguagem de programação C, sendo sete tópicos em específico, onde que cada um representa um assunto dentro desta área do conhecimento, no qual os mesmo estão apresentados logo abaixo.

- Aritmética
- Operadores matemáticos
- Tipos de dados
- Entrada e saída de dados
- Organização e modularidade do código
- Funções, vetores e matrizes
- Strings em linguagem de programação C.

O objetivo específico alinhado ao objetivo geral, tem como objetivo gerar um código que atenda a todos os tópicos que estão listados acima.

8 MATERIAIS E MÉTODOS

Para a realização do código que será apresentado nesta presente pesquisa, foi apresentada um história hipotética juntamente com o fornecimento de quatro “strings” em formato hexadecimal, onde cada uma das strings apresenta uma mensagem como pode ser visto na figura 8 que está abaixo.

Figura 8 - Strings fornecidas para elaboração do código de decodificação

Entrada	Saída
1 0 566F6388732073C66F2076656_ E6365646F867265732C20766F_ 63C3887320636FBE6E7365677_ 5656D2E002DC6C921B7B87FCF	Vocês são vencedores, vocês conseguem.
1 3 5465636E6F336C6f676961206_ 46120496E666f726D6187C66F_ 2E003333333333333333333333_ 33333333333333333333333333	Tecnologia da Informação.
2 0 566F6388732073C66F2076656_ E6365646F867265732C00566F_ 6388732073C66F2076656E636_ 5646F867265732C00332C2C2C 3 566F638873C320636F6E73656_ 775656D2E002DC6C921B7B87F_ CF566F638873C320636F6E736_ 56775656D2E002DC6C921B7B8	Vocês são vencedores, Vocês conseguem.

Fonte: Professor Orientador Me. Jefferson Salomão Rodrigues

Juntamente com o fornecimento dessas strings, também foi fornecido uma equação padrão para validação do caracteres das strings, com os valores de constantes pré-definidos como pode ser visto na figura 9 que está abaixo.

Figura 9 - Equação padrão para validação dos caracteres das strings

$$f(x, b) = a_0 + (a_1 + b)x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

Em que,

- $a_0 = 186,752$
- $a_1 = -148,235$
- $a_2 = 34,5049$
- $a_3 = -3,5091$
- $a_4 = 0,183166$
- $a_5 = -0,00513554$
- $a_6 = 0,0000735464$
- $a_7 = -4,22038 \times 10^{-7}$

Fonte: Professor Orientador Me. Jefferson Salomão Rodrigues

Como pode ser visto na figura 11 que está acima o caractere nulo está identificado pelo quadrado laranja indicando o fim da mensagem, sendo assim diante de todas essas informações obtidas foi realizado o procedimento de verificação da equação de validação é também das mensagens em hexadecimal.

Para verificação das mensagens em hexadecimal, primeiramente realizou-se um código em C no software Visual Studio Code, com o objetivo de se ler cada string fornecida e separá-las em duplas de dois algarismos ou pares de algarismos e letras, como pode ser visto na figura 12 que está abaixo.

Figura 12 - Código em linguagem C, para separação em duplas de dois algarismos ou letras das strings fornecidas anteriormente.

```

C pares.c > main()
1  #include <stdio.h>
2  #include <string.h>
3
4  #define MAX_LENGTH 1024
5
6  void separarEmPares(const char* hexString) {
7      int length = strlen(hexString);
8
9
10     if (length % 2 != 0) {
11         printf("A string hexadecimal deve ter um comprimento par.\n");
12         return;
13     }
14
15     printf("Pares de dois dígitos:\n");
16
17     for (int i = 0; i < length; i += 2) {
18         printf("%c%c\n", hexString[i], hexString[i + 1]);
19     }
20 }
21
22
23 int main() {
24     char hexString[MAX_LENGTH];
25
26     printf("Digite a string hexadecimal : \n", MAX_LENGTH - 1);
27
28     if (fgets(hexString, sizeof(hexString), stdin) != NULL) {
29         hexString[strcspn(hexString, "\n")] = '\0';
30         separarEmPares(hexString);
31     } else {
32         printf("Digite um valor válido.\n");
33     }
34 }
35
36 return 0;
37 }

```

Fonte: Autores

Com a utilização do código em C que está apresentado pela figura 12 que está acima, foi possível a divisão das strings em duplas, permitindo assim a verificação das informações

fornecidas antes de se realizar produção do código, sendo assim foi produzida uma planilha no software Microsoft Excel, como pode ser visto na figura 13 que está abaixo.

Figura 13 - Tabela de verificação do 1º caso de teste e da equação de validação e da mensagem fornecida pela 1º string.

Constante B	Pares Ordenados	Hexadecimal Binário	Posição do Caracore	Constantes das Equações	alg-1/b-1	alg-2	alg-3	alg-4	alg-5	alg-6	alg-7	Solução da Combinação	Letras
0	00	00	1	A0	10750	10750	10750	10750	10750	10750	10750	10750	0
	01	01	2	A1	10751	10751	10751	10751	10751	10751	10751	10751	1
	02	02	3	A2	10752	10752	10752	10752	10752	10752	10752	10752	2
	03	03	4	A3	10753	10753	10753	10753	10753	10753	10753	10753	3
	04	04	5	A4	10754	10754	10754	10754	10754	10754	10754	10754	4
	05	05	6	A5	10755	10755	10755	10755	10755	10755	10755	10755	5
	06	06	7	A6	10756	10756	10756	10756	10756	10756	10756	10756	6
	07	07	8	A7	10757	10757	10757	10757	10757	10757	10757	10757	7
	08	08	9	A8	10758	10758	10758	10758	10758	10758	10758	10758	8
	09	09	10	A9	10759	10759	10759	10759	10759	10759	10759	10759	9
	0A	0A	11	AA	10760	10760	10760	10760	10760	10760	10760	10760	A
	0B	0B	12	AB	10761	10761	10761	10761	10761	10761	10761	10761	B
	0C	0C	13	AC	10762	10762	10762	10762	10762	10762	10762	10762	C
	0D	0D	14	AD	10763	10763	10763	10763	10763	10763	10763	10763	D
	0E	0E	15	AE	10764	10764	10764	10764	10764	10764	10764	10764	E
	0F	0F	16	AF	10765	10765	10765	10765	10765	10765	10765	10765	F
	10	10	17	B0	10766	10766	10766	10766	10766	10766	10766	10766	0
	11	11	18	B1	10767	10767	10767	10767	10767	10767	10767	10767	1
	12	12	19	B2	10768	10768	10768	10768	10768	10768	10768	10768	2
	13	13	20	B3	10769	10769	10769	10769	10769	10769	10769	10769	3
	14	14	21	B4	10770	10770	10770	10770	10770	10770	10770	10770	4
	15	15	22	B5	10771	10771	10771	10771	10771	10771	10771	10771	5
	16	16	23	B6	10772	10772	10772	10772	10772	10772	10772	10772	6
	17	17	24	B7	10773	10773	10773	10773	10773	10773	10773	10773	7
	18	18	25	B8	10774	10774	10774	10774	10774	10774	10774	10774	8
	19	19	26	B9	10775	10775	10775	10775	10775	10775	10775	10775	9
	1A	1A	27	BA	10776	10776	10776	10776	10776	10776	10776	10776	A
	1B	1B	28	BB	10777	10777	10777	10777	10777	10777	10777	10777	B
	1C	1C	29	BC	10778	10778	10778	10778	10778	10778	10778	10778	C
	1D	1D	30	BD	10779	10779	10779	10779	10779	10779	10779	10779	D
	1E	1E	31	BE	10780	10780	10780	10780	10780	10780	10780	10780	E
	1F	1F	32	BF	10781	10781	10781	10781	10781	10781	10781	10781	F
	20	20	33	C0	10782	10782	10782	10782	10782	10782	10782	10782	0
	21	21	34	C1	10783	10783	10783	10783	10783	10783	10783	10783	1
	22	22	35	C2	10784	10784	10784	10784	10784	10784	10784	10784	2
	23	23	36	C3	10785	10785	10785	10785	10785	10785	10785	10785	3
	24	24	37	C4	10786	10786	10786	10786	10786	10786	10786	10786	4
	25	25	38	C5	10787	10787	10787	10787	10787	10787	10787	10787	5
	26	26	39	C6	10788	10788	10788	10788	10788	10788	10788	10788	6
	27	27	40	C7	10789	10789	10789	10789	10789	10789	10789	10789	7
	28	28	41	C8	10790	10790	10790	10790	10790	10790	10790	10790	8
	29	29	42	C9	10791	10791	10791	10791	10791	10791	10791	10791	9
	2A	2A	43	CA	10792	10792	10792	10792	10792	10792	10792	10792	A
	2B	2B	44	CB	10793	10793	10793	10793	10793	10793	10793	10793	B
	2C	2C	45	CC	10794	10794	10794	10794	10794	10794	10794	10794	C
	2D	2D	46	CD	10795	10795	10795	10795	10795	10795	10795	10795	D
	2E	2E	47	CE	10796	10796	10796	10796	10796	10796	10796	10796	E
	2F	2F	48	CF	10797	10797	10797	10797	10797	10797	10797	10797	F
	30	30	49	D0	10798	10798	10798	10798	10798	10798	10798	10798	0
	31	31	50	D1	10799	10799	10799	10799	10799	10799	10799	10799	1
	32	32	51	D2	10800	10800	10800	10800	10800	10800	10800	10800	2
	33	33	52	D3	10801	10801	10801	10801	10801	10801	10801	10801	3
	34	34	53	D4	10802	10802	10802	10802	10802	10802	10802	10802	4
	35	35	54	D5	10803	10803	10803	10803	10803	10803	10803	10803	5
	36	36	55	D6	10804	10804	10804	10804	10804	10804	10804	10804	6
	37	37	56	D7	10805	10805	10805	10805	10805	10805	10805	10805	7
	38	38	57	D8	10806	10806	10806	10806	10806	10806	10806	10806	8
	39	39	58	D9	10807	10807	10807	10807	10807	10807	10807	10807	9
	3A	3A	59	DA	10808	10808	10808	10808	10808	10808	10808	10808	A
	3B	3B	60	DB	10809	10809	10809	10809	10809	10809	10809	10809	B
	3C	3C	61	DC	10810	10810	10810	10810	10810	10810	10810	10810	C
	3D	3D	62	DD	10811	10811	10811	10811	10811	10811	10811	10811	D
	3E	3E	63	DE	10812	10812	10812	10812	10812	10812	10812	10812	E
	3F	3F	64	DF	10813	10813	10813	10813	10813	10813	10813	10813	F
	40	40	65	E0	10814	10814	10814	10814	10814	10814	10814	10814	0
	41	41	66	E1	10815	10815	10815	10815	10815	10815	10815	10815	1
	42	42	67	E2	10816	10816	10816	10816	10816	10816	10816	10816	2
	43	43	68	E3	10817	10817	10817	10817	10817	10817	10817	10817	3
	44	44	69	E4	10818	10818	10818	10818	10818	10818	10818	10818	4
	45	45	70	E5	10819	10819	10819	10819	10819	10819	10819	10819	5
	46	46	71	E6	10820	10820	10820	10820	10820	10820	10820	10820	6
	47	47	72	E7	10821	10821	10821	10821	10821	10821	10821	10821	7
	48	48	73	E8	10822	10822	10822	10822	10822	10822	10822	10822	8
	49	49	74	E9	10823	10823	10823	10823	10823	10823	10823	10823	9
	4A	4A	75	EA	10824	10824	10824	10824	10824	10824	10824	10824	A
	4B	4B	76	EB	10825	10825	10825	10825	10825	10825	10825	10825	B
	4C	4C	77	EC	10826	10826	10826	10826	10826	10826	10826	10826	C
	4D	4D	78	ED	10827	10827	10827	10827	10827	10827	10827	10827	D
	4E	4E	79	EE	10828	10828	10828	10828	10828	10828	10828	10828	E
	4F	4F	80	EF	10829	10829	10829	10829	10829	10829	10829	10829	F
	50	50	81	F0	10830	10830	10830	10830	10830	10830	10830	10830	0
	51	51	82	F1	10831	10831	10831	10831	10831	10831	10831	10831	1
	52	52	83	F2	10832	10832	10832	10832	10832	10832	10832	10832	2
	53	53	84	F3	10833	10833	10833	10833	10833	10833	10833	10833	3
	54	54	85	F4	10834	10834	10834	10834	10834	10834	10834	10834	4
	55	55	86	F5	10835	10835	10835	10835	10835	10835	10835	10835	5
	56	56	87	F6	10836	10836	10836	10836	10836	10836	10836	10836	6
	57	57	88	F7	10837	10837	10837	10837	10837	10837	10837	10837	7
	58	58	89	F8	10838	10838	10838	10838	10838	10838	10838	10838	8
	59	59	90	F9	10839	10839	10839	10839	10839	10839	10839	10839	9
	5A	5A	91	FA	10840	10840	10840	10840	10840	10840	10840	10840	A
	5B	5B	92	FB	10841	10841	10841	10841	10841	10841	10841	10841	B
	5C	5C	93	FC	10842	10842	10842	10842	10842	10842	10842	10842	C
	5D	5D	94	FD	10843	10843	10843	10843	10843	10843	10843	10843	D
	5E	5E	95	FE	10844	10844	10844	10844	10844	10844	10844	10844	E
	5F	5F	96	FF	10845	10845	10845	10845	10845	10845	10845	10845	F
	60	60	97	00	10846	10846	10846	10846	10846	10846	10846	10846	0
	61	61	98	01	10847	10847	10847	10847	10847	10847	10847	10847	1
	62	62	99	02									

A versão final dos códigos está disponibilizada na plataforma de hospedagem GitHub em um repositório público onde o link de acesso a este repositório está apresentado pela figura 14 que está abaixo.


Figura 14 - Link do repositório na plataforma de hospedagem GitHub

<https://github.com/wagnerserpaporto/Trabalho-de-Algoritmo.git>

Fonte: <https://github.com/wagnerserpaporto/Trabalho-de-Algoritmo.git>

Os códigos apresentados neste presente trabalho estão dentro do repositório acessado pelo link apresentado pela figura 14 que está acima onde que na plataforma GitHub os mesmo estão identificados pelo “commit” ou comentário “Versão Final” como pode ser visto na figura 15 que está abaixo.

Figura 15 - Localização dos códigos na plataforma GitHub

 wagnerserpaporto	Instruções Revisado.md	459c678 · 45 minutes ago	🕒 4 Commits
📁 output	Versão Final	49 minutes ago	
📄 .gitattributes	Initial commit	last month	
📄 PontoExtra.c	Versão Final	49 minutes ago	
📄 README.md	Instruções Revisado.md	45 minutes ago	
📄 decodificador.c	Versão Final	49 minutes ago	
📄 executa.c	Versão Final	49 minutes ago	
📄 funcaoondas.c	Versão Final	49 minutes ago	

Fonte: <https://github.com/wagnerserpaporto/Trabalho-de-Algoritmo.git>

Como pode ser visto na figura 15 que está acima os códigos apresentados neste presente trabalho estão dentro do retângulo em vermelho, onde que os mesmos estão identificados pelo “commit” ou comentário “Versão Final” onde que neste mesmo repositório existe um arquivo “readme.md”, com a apresentação geral do repositório e demais informações, observando-se que para fins de organização existe um outro repositório na mesma plataforma que apresenta o histórico de versões e todos os arquivos utilizados para concepção do código, onde que seu link está disponibilizado no “readme” do repositório, sendo assim os códigos serão detalhados e discutidos a partir do próximo item se iniciado pelo código “funçãoondas.c”.

9 RESULTADOS E DISCUSSÃO

9.1 CÓDIGO “FUNÇÃOONDAS”

Para se iniciar a decodificação das strings foi concebido o código em linguagem C denominado “funçãoondas”, onde que o mesmo recebeu este nome devido ao contexto histórico hipotético fornecido indicar que as strings que devem ser decodificadas foram hipoteticamente obtidas através da interceptação de ondas de rádio, sendo assim o código concebido pode ser visto na figura 16 que está abaixo.

Figura 16 - Visão geral do código “funçãoondas”

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
int func_val(int X, int B) {
    double VALORES_AS[] = {186.752, -148.235, 34.5049, -3.5091, 0.183166, -0.00513554, 0.0000735464, -4.22038e-7};
    double RESULT = 0.0;
    int I;
    for (I = 0; I < 8; I++) {
        RESULT += VALORES_AS[I] * pow(X, I);
    }
    RESULT += B * X;
    int RESULTADO_FINAL = round(RESULT);
    //printf("%d", resultf); verifica??o do resultado final
    return RESULTADO_FINAL;
}
/*void main(){
    // for gerado apenas para confirma??o de valores segundo novo vetor
    int i;
    for (i = 1; i < 100; i++){
        int resultado = func_val(i, 0);
        if(resultado != 0){
            printf("%2d\t%d\n", i, resultado);
        }
    }
}*/
```

Fonte: Autores

Para se iniciar a realização do código apresentado pela figura 16 que está acima foram inseridas as seguintes bibliotecas, “stdio.h”, “stdlib.h”, “math.h”, “string.h”, como pode ser visto na figura 17 que está abaixo.

Figura 17 - Bibliotecas incluídas para construir o código “funçãoondas”

```
C funcaoondas.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <string.h>
```

Fonte: Autores

A biblioteca “stdio.h” é uma biblioteca de entrada e saída padrão onde que de acordo com PET-BCC (2024) é onde estão localizadas as funções referentes às operações que necessitam de mecanismos que operem em função da entrada e da saída padrão.

Sendo assim no código “funçãoondas” discutido nesse tópico o seu uso se faz desnecessário pois não há interação direta do usuário no código, ou seja, não existe uma função de “scanf” para o usuário inserir algum dado, como também não existe uma função “printf”, como pode ser visto na figura 16 que está acima, sendo assim fazendo se desnecessário o uso da biblioteca “stdio.h”.

Já biblioteca “stdlib.h” de acordo com Feofiloff (2013) é onde estão localizadas as funções correspondentes a manipulação e alocação de memória, como também as funções necessárias para converter números que estão representados em strings, sendo assim como pode ser visto no código apresentado pela figura 16 que está acima, o mesmo não contém nenhuma função específica como “malloc”, “free” ou “exit” e entre diversas outras, como também o mesmo não possui strings, fazendo assim desnecessária a utilização da biblioteca “stdlib.h” pois não há funções que justifique seu uso.

Como foi dito no parágrafo anterior o código “funçãoondas” apresentado pela figura 16, não contém strings sendo assim desnecessário o uso da biblioteca “string.h”, pois de acordo com PET-BCC (2024), esta biblioteca contém ferramentas para manipulação de strings, é como neste código não apresenta nenhuma string o seu uso se torna desnecessário, ficando o seguinte ajuste no código como pode ser visto na figura 18 que está abaixo.

Figura 18 - Código “funçãoondas” ajustado

```
C funcaoondas.c > ...
1  #include <math.h>
2  int func_val(int X, int B) {
3      double VALORES_AS[] = {186.752, -148.235, 34.
4      double RESULT = 0.0;
5      int I;
6      for (I = 0; I < 8; I++) {
7          RESULT += VALORES_AS[I] * pow(X, I);
8      }
9      RESULT += B * X;
10     int RESULTADO_FINAL = round(RESULT);
```

Fonte: Autores

Ao se comparar a figura 18 com as figuras 16 e 17 que estão acima pode se perceber que a única biblioteca que permaneceu depois do ajuste foi a biblioteca “math.h”, pois seu uso se faz necessário devido ao código ter o objetivo de realizar a verificação da equação fornecida anteriormente, apresentada no item 8, figura 9, como pode ser visto na figura 19 que está abaixo na qual apresenta a implementação do cálculo da mesma.

Figura 19 - Código “funçãoondas”, equação detalhada

```
#include <math.h>
int func_val(int X, int B) {
    double VALORES_AS[] = {186.752, -148.235, 34.5049, -3.5091, 0.183166, -0.00513554, 0.0000735464, -4.22038e-7};
    double RESULT = 0.0;
    int I;
    for (I = 0; I < 8; I++) {
        RESULT += VALORES_AS[I] * pow(X, I);
    }
    RESULT += B * X;
    int RESULTADO_FINAL = round(RESULT);
}
```

Fonte: Autores

Como pode ser visto na figura 19 que está acima temos uma equação do tipo exponencial, onde que para calcular os seus valores temos uma função “pow” e uma outra função “round” como pode ser visto na figura 20 que está abaixo.

Figura 20 - Código “funçãoondas”, detalhe das funções “pow” e “round”

```
for (I = 0; I < 8; I++) {
    RESULT += VALORES_AS[I] * pow(X, I);
}
RESULT += B * X;
int RESULTADO_FINAL = round(RESULT);
```

Fonte: Autores

O uso da biblioteca “math.h” se justifica através do uso das funções “pow” e “round” onde que a função “pow” pertencente a esta biblioteca está sendo utilizada para calcular as potências da equação e função “round” também pertencente a esta biblioteca sendo utilizada para arredondar os valores calculados pela função “pow”.

Como pode ser visto nas figuras 16, 18 e 19 que estão acima a equação está alocada dentro de uma variável do tipo “double” devido ao fato de que de acordo com Microsoft Ignite (2024) este tipo de variável em linguagem C é utilizada para armazenar números de ponto flutuante, com uma maior precisão do que a variável do tipo “float”, onde que este tipo de

variável geralmente ocupa 8 bytes de memória, podendo representar valores em uma faixa muito ampla atendendo a necessidade do código “funçãoondas” de realizar cálculos com resultados muito extensos, onde temos valores de potência maior que 5 como pode ser visto nas figuras 18, 19 e 20 que estão acima.

Observando essas mesmas figuras pode se notar que foi utilizada uma variável do tipo “int” para armazenar o resultado da final da equação a cada cálculo realizado pela função “for” como pode ser visto na figura 21 que está abaixo.

Figura 21 - Código “funçãoondas”, detalhe do resultado da equação

```

}
RESULT += B * X;
int RESULTADO_FINAL = round(RESULT);
//printf("%d", resultf); verifica?o do resultado final
return RESULTADO_FINAL;

```

Fonte: Autores.

A variável do tipo “int” foi utilizada nessa situação devido ao fato de que o valor calculado pela equação está sendo armazenado na variável “RESULT” e tendo seu valor arredondado para um valor inteiro na variável “RESULTADO_FINAL” como pode ser visto na figuras 16, 18, 19 e 21 que estão acima.

sendo assim realizada todas essas observações temos o seguinte código apresentado pela figura 22 que está abaixo para validar os caracteres das strings fornecidas anteriormente apresentadas pelas figuras 8,10 e 11, presentes no item 8, onde que a variável “VALORES_AS” representam os valores de “a” da equação fornecida e “x” e o valor da posição onde se inicia o caractere como pode ser visto abaixo.

Figura 22 - Código “funçãoondas”.

```

#include <math.h>
int func_val(int X, int B) {
    double VALORES_AS[] = {186.752, -148.235, 34.5049, -3.5091, 0.183166, -0.00513554, 0.0000735464, -4.22038e-7};
    double RESULT = 0.0;
    int I;
    for (I = 0; I < 8; I++) {
        RESULT += VALORES_AS[I] * pow(X, I);
    }
    RESULT += B * X;
    int RESULTADO_FINAL = round(RESULT);
    //printf("%d", resultf); verifica?o do resultado final
    return RESULTADO_FINAL;
}

```

Fonte: Autores.

9.2 CÓDIGO “DECODIFICADOR”

Logo após de se construir o código de validação dos caracteres das strings fornecidas, o outro código que foi construído denomina-se “decodificador” o mesmo é apresentado na figura 23 que segue abaixo.

Figura 23 - Visão geral do código “decodificador”

```
#include <stdio.h>
#include <string.h>
#include "funcaoondas.c"
void separa_e_converte(char MENSAGEM_CIFRADA[], int B) {
    int LEN = strlen(MENSAGEM_CIFRADA);
    char PAR[3];
    char *endptr;
    char ENCERRAR[] = "00";
    int I, POS = 1; // pos = 1 para demais
    for (I = 0; I < LEN; I += 2) {
        PAR[0] = MENSAGEM_CIFRADA[I];
        PAR[1] = MENSAGEM_CIFRADA[I + 1];
        PAR[2] = '\0';
        //pos++;
        int CONVERTEH2D = strtol(PAR, &endptr, 16); //CONVERSÃO DE HEXADECIMAL PARA DECIMAL
        if (strcmp(PAR, ENCERRAR) == 0) {
            break;
        } else {
            if (func_val(POS, B) != 0){
                printf("%c", CONVERTEH2D);
                //printf("Pos: %2d -> X = %3d | \t%s -> \t%.3d -> \t%c\n", pos, func_val(pos, b), par, convertteh2d, convertteh2d);
            }
        }
        POS++;
    }
}
```

Fonte: Autores

Como pode ser observado na figura 23 que está acima o código “decodificador” faz uso das bibliotecas “stdio.h” e “string.h”, onde que anteriormente no item 9.1 foi discutido que o uso da biblioteca “stdio.h” se faz necessário quando há necessidade de ferramentas referentes às operações que necessitam de mecanismos que operem a função de entrada e saída padrão, sendo assim o seu uso é justificado pela função “printf” presente no código como pode ser visto na figura 24 que está abaixo.

Figura 24 - Detalhe da função “printf” presente no código “decodificador”

```
} else {
    if (func_val(POS, B) != 0){
        printf("%c", CONVERTEH2D);
        //printf("Pos: %2d -> X = %3d | \t%s -> \t%.3d -> \t%c\n", pos, func_val(pos, b), par, convertteh2d, convertteh2d);
    }
}
```

Fonte: Autores

Sendo assim no código “decodificador” observando a figura 23 que está acima o mesmo também faz o uso da biblioteca “string.h”, é como foi discutido anteriormente no item 9.1, a

UCB – Algoritmos e Programação Estruturada

mesma se trata de uma biblioteca que contém ferramentas para manipulação de strings, onde que seu uso é justificado pelo uso das funções “strtol” e “strcmp” pertencentes a essa biblioteca como pode ser visto na figura 25 que está abaixo.

Figura 25 - Detalhe das funções “strtol” e “strcmp” presente no código “decodificador”

```
int CONVERTEH2D = strtol(PAR, &endptr, 16); //CONVERSÃO DE HEXADECIMAL PARA DECIMAL
if (strcmp(PAR, ENCERRAR) == 0) {
```

Fonte: Autores

Como pode ser visto na figura 23 acima, onde é apresentado o código “decodificador” é possível observar uma outra função pertencente a biblioteca “string.h” que é a função “strlen”, onde que esta função está retornando o comprimento da string fornecida e sendo utilizada para separar a string em duplas ou pares como pode ser visto na figura 26 que está logo abaixo.

Figura 26 - Detalhe da função “strlen” presente no código “decodificador”

```
void separa_e_converte(char MENSAGEM_CIFRADA[], int B) {
    int LEN = strlen(MENSAGEM_CIFRADA);
    char PAR[3];
    char *endptr;
    char ENCERRAR[] = "00";
    int I, POS = 1; // pos = 1 para demais
    for (I = 0; I < LEN; I += 2) {
        PAR[0] = MENSAGEM_CIFRADA[I];
        PAR[1] = MENSAGEM_CIFRADA[I + 1];
        PAR[2] = '\0';
        //pos ++;
```

Fonte: Autores

Como pode ser visto na figura 26 temos uma função “for” após a função “strlen”, onde que a mesma está percorrendo a string e separando em pares onde que esses pares estão sendo validados pelo código “funçãoondas” apresentado no item 9.1 através da inclusão do código fonte como pode ser visto na figura 27 que está abaixo.

Figura 27 - Detalhe da inclusão das bibliotecas e do código “funçãoondas” presente no código “decodificador”

```
#include <stdio.h>
#include <string.h>
#include "funcaoondas.c"
```

Fonte: Autores

Após essa validação dos caracteres, os mesmos são convertidos de string hexadecimal para um número inteiro que possui valor representativo binário através de uma função “strtol”, onde que observando a figura 23 acima é possível comprovar que nesta secção do código e onde ocorre a decodificação da mensagem, onde que a função “strtol” converte os pares de caracteres hexadecimais em um valor decimal, no qual esses valores decimais representam os códigos ASCII dos caracteres correspondentes., onde que para comprovar tal afirmação será apresentada a figura 28 que está logo abaixo.

Figura 28 - Detalhe decodificação da string presente no código “decodificador”

```
for (I = 0; I < LEN; I += 2) {
    PAR[0] = MENSAGEM_CIFRADA[I];
    PAR[1] = MENSAGEM_CIFRADA[I + 1];
    PAR[2] = '\0';
    //pos ++;
    int CONVERTEH2D = strtol(PAR, &endptr, 16);
    if (strcmp(PAR, ENCERRAR) == 0) {
        break;
    } else {
        if (func_val(POS, B) != 0){
            printf("%c", CONVERTEH2D);
            //printf("Pos: %2d -> X = %3d | \t%s", POS, CONVERTEH2D, MENSAGEM_CIFRADA);
        }
    }
}
POS++;
```

Fonte: Autores

Após realizada essa etapa do código é necessário criar um outro código que imprima os resultados e as mensagens decodificadas, ou seja, um código de execução.

9.3 CÓDIGO “EXECUTAR”

Nesse tópico será apresentado o código de execução do demais códigos onde que no mesmo está inserido o código decodificador como pode ser visto na figura 29 abaixo.

Figura 29 - Detalhe da inclusão do código “decodificador”, no código “executar”.

```
#include <stdio.h>
#include <string.h>
#include "decodificador.c"
```

Fonte: Autores

Como pode ser observado na figura 29 que está acima o código “executar” faz uso das bibliotecas “string.h” e “stdio.h”, onde que anteriormente nos itens 9.1 e 9.2 foi discutido que o uso da biblioteca “stdio.h” se faz necessária quando há necessidade de ferramentas referentes às operações que necessitam de mecanismos que operem a função de entrada e saída padrão, sendo assim o seu uso é justificado pela função “printf” presente no código é a biblioteca “string.h” pela função “fgets” e “fflush” pode ser visto na figura 30 que está abaixo que apresenta o código “executar”.

Figura 30 - Visão geral do código “executar”.

```
#include <stdio.h>
#include <string.h>
#include "decodificador.c"
void main(){
    int TESTES;
    char M_CIFRADA[100];
    printf("Quantos testes: ");
    scanf("%d", &TESTES);
    fflush(stdin);

    int I, CHAVE;
    for (I = 0; I < TESTES; I++){
        printf("\nQual com a chave B: ");
        scanf("%d", &CHAVE);

        fflush(stdin);

        printf("Entre com a mensagem cifrada: ");
        fgets(M_CIFRADA, 100, stdin);
        fflush(stdin);

        separa_e_converte(M_CIFRADA, CHAVE);
    }
}
```

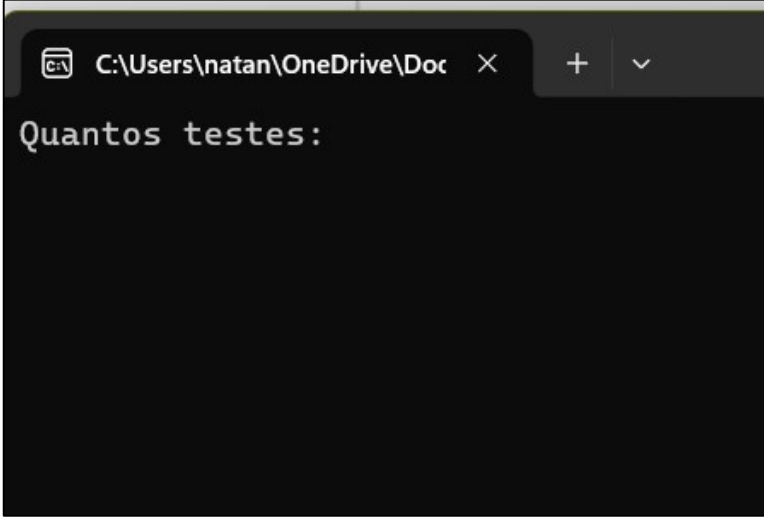
Fonte: Autores

Como pode ser observado na figura 30 que está acima o uso da função “fgets” tem o objetivo de se ler a string fornecida que será executada pelo código “decodificador”, inserido no mesmo, onde que de acordo com Feofiloff (2013) e maneira mais segura de se ler strings pois permite especificar o tamanho máximo da entrada evitando assim estourar os limites da variável.

A função “fflush” observada no código apresentado pela figura 30 que está acima tem o objetivo garantir que os dados armazenados pela variável sejam escritos imediatamente para o destino, como um arquivo ou a saída padrão, sendo assim evitando deixar resquícios de dados dentro da variável que possam trazer erros futuros nas próximas leituras, deixando o código limpo e eficiente.

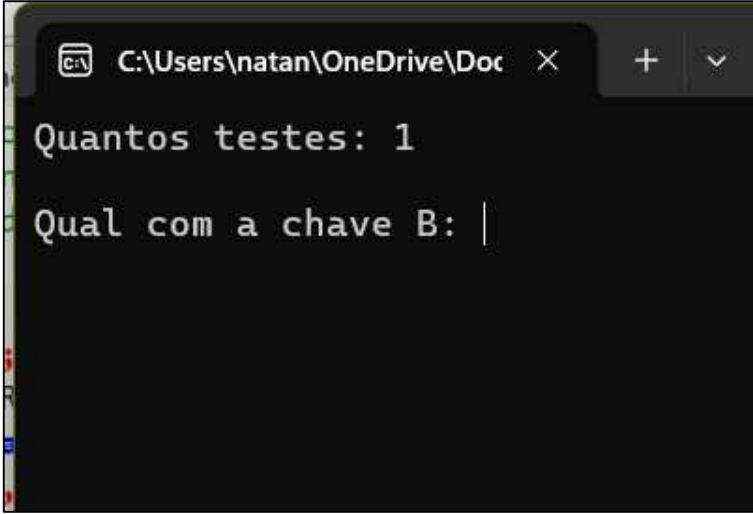
Como todos esses códigos apresentados temos a execução do código que será apresentado pelas figuras 31, 32, 33 e 34 que estão abaixo.

Figura 31 - Executando o código.

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\natan\OneDrive\Doc' and standard window controls. The command prompt displays the text 'Quantos testes:' in a monospaced font, waiting for input.

Fonte: Autores

Figura 32 - Executando o código inserindo a quantidade de testes.

A screenshot of the same Windows command prompt window. The text 'Quantos testes: 1' is now displayed, indicating that the user has entered '1'. Below this, a new prompt 'Qual com a chave B: |' is visible, with a cursor at the end of the line.

Fonte: Autores

10 CONCLUSÃO

Conclui-se que, entender os limites das variáveis na linguagem C é algo imprescindível devido ao sua influência direta na otimização do uso da memória e na melhoria do desempenho do programa ou código, onde que o desrespeito a esses limites gera códigos ou programas que geram respostas erradas no pior dos casos e no melhor dos casos nem mesmo executam, sendo assim os limites das variáveis em C como foi discutido ao longo deste presente trabalho, apresentam influência direta na eficiência e na confiabilidade do programa ou código, onde que a prevenção de erros de estouro de memória evita que os dados inseridos nas variáveis se sobreponham gerando um código ou programa limpo, estável e confiável.

REFERÊNCIAS

- AISAWA, William Akihiro Alves. Técnicas para identificação de funções de bibliotecas em binários vinculados estaticamente, Tese de Mestrado em Ciência da Computação, Universidade Federal de São Carlos, 2020. Disponível em: <https://repositorio.ufscar.br/handle/ufscar/13375>. Acesso em: 5 set 2024.
- ALMEIDA, L. **Linguagem C IF61A/IF71A -Computação 1**. [s.l: s.n.]. Disponível em: https://pessoal.dainf.ct.utfpr.edu.br/leoneloalmeida/cursos/if71a-s83-2016-01/linguagemC_1.pdf.
- ANDRADE, E. PROGRAMAÇÃO ESTRUTURADA. Disponível em: https://biblioteca.uniscd.edu.mz/bitstream/123456789/2446/1/CSI%202103_PT%20Programac%CC%A7a%CC%83o%20Estruturada.pdf. Acesso em: 5 set. 2024.
- BACKES, André. Linguagem C: Completa e Descomplicada. Rio de Janeiro: Grupo GEN, 2023. E-book. ISBN 9788595152090. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788595152090/>. Acesso em: 04 set. 2024.
- BAHIANA, M. **Introdução à Computação Científica em C**. [s.l: s.n.]. Disponível em: <https://www.if.ufrj.br/~sandra/MetComp/2009-1/doc/capitulo1.pdf>. Acesso em: 28 set. 2024.
- BRUNNER, T.; PORKOLÁB, Z. Programming Language History: Experiences based on the Evolution of C++. Proceedings of the 10th International Conference on Applied Informatics, 2018.
- COELHO, Jose. Introdução à programação: variáveis. Repositorioaberto.uab.pt, 2022. Disponível em: <http://hdl.handle.net/10400.2/12937>. Acesso em 6 set. 2024.
- DE AZEVEDO Fabio Souto; KONZEN, Pedro Henrique de Almeida; SAUTER, Esequia; Computação Científica em Linguagem C Um Livro Colaborativo, 2020. Disponível em:
- DE OLIVEIRA, Raiza Arteman; MOLGORA, Adriana Betânia de Paula. DESENVOLVIMENTO DE UM GERADOR DE GRÁFICOS DE FUNÇÕES. ANAIS DO ENIC, [S. l.], n. 6, 2015. Disponível em: <https://anaisonline.uems.br/index.php/enic/article/view/2344>. Acesso em: 5 set. 2024.
- DIETZ, W. et al. Understanding Integer Overflow in C/C++. ACM Transactions on Software Engineering and Methodology, v. 25, n. 1, p. 1–29, 2 dez. 2015. Disponível em: <https://dl.acm.org/doi/abs/10.1145/2743019>. Acesso em 6 set. 2024.
- FEOFILLOF, Paulo. **Algoritmos: Em Linguagem C**. Elsevier Brasil, 2013.
- FREITAS, Ricardo Luís. Compiladores. Pontifícia Universidade Católica de Campinas, 2019 Disponível em:

<<https://www.cesarkallas.net/arquivos/faculdade/compiladores/teoria/apostila-de-compiladores-ec.pdf>>. Acesso em: 5 set. 2024.

GUERREIRO, P. A Mesma Velha Questão: Como Ensinar Programação?. In Quinto Congresso Iberoamericano de Educación Superior, 1986.

<https://www.ufrgs.br/reatmat/ComputacaoCientifica/livro/main.html>. Acesso em: 12 set 2024.

ISO/IEC 9899, DOCUMENTAÇÃO OFICIAL DA LINGUAGEM C. Acessado em: 09 set. 2024.

KERNIGHAN, Brian W.; RITCHIE, Dennis M. The C programming language. 2002.

KERNIGHAN, Brian W; RITCHIE, Dennis M. The C Programming Language. 2. ed. Upper Saddle River: Prentice Hall, 1988.

LIDANI, Rafael; BORBA, Milton. Algoritmos Iterativos para Busca de Zeros de Funções. Departamento de Matemática, Universidade do Estado de Santa Catarina, 1999.

MANZANO, José Augusto Navarro G. Estudo Dirigido de Linguagem C. Rio de Janeiro: Grupo GEN, 2002. E-book. ISBN 9788536519128. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788536519128/>. Acesso em: 05 set. 2024.

MANZANO, José Augusto Navarro G. Estudo Dirigido de Linguagem C. Rio de Janeiro: Grupo GEN, 2002. E-book. ISBN 9788536519128. Disponível em: <https://integrada.minhabiblioteca.com.br/#/books/9788536519128/>. Acesso em: 05 set. 2024.

MICROSOFT. Documentação C/C++ (C Language), 2022. Disponível em: <https://learn.microsoft.com/pt-br/cpp/c-language/?view=msvc-170>. Acesso em: 09 set. 2024.

OUALLINE, Steve. Practical C programming. " O'Reilly Media, Inc.", 1997.

PET-BCC, String.h - Manual C. 2024. Disponível em: <<https://petbcc.ufscar.br/string/>>. Acesso em: 25 set 2024.

RICARTE, Ivan. Programação de Sistemas: Uma Introdução. Departamento de Engenharia de Computação e Automação Industrial. Universidade Estadual de Campinas. São Paulo, 2001.

SCHILDT, Herbert. C completo e total. Editora Pearson Universidades, ISBN 8534605955, 9788534605953, 1997 Disponível em: <https://books.google.com.br/books/about/C_completo_e_total.html?id=PbI0AAAACAAJ&redir_esc=y>. Acesso em: 6 set. 2024.

SHAW, Alex; DOGETT, Dusten; HAFIZ, Munawar. Automatically fixing c buffer overflows using program transformations. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 2014. p. 124-135.

ZHANG, Haoxiang et al. A study of c/c++ code weaknesses on stack overflow. IEEE Transactions on Software Engineering, v. 48, n. 7, p. 2359-2375, 2021.

APÊNDICE A – CODIGO FUNÇÃO ONDAS

```
#include <math.h>
int func_val(int X, int B) {
    double VALORES_AS[] = {186.752, -148.235, 34.5049, -3.5091, 0.183166, -
0.00513554, 0.0000735464, -4.22038e-7};
    double RESULT = 0.0;
    int I;
    for (I = 0; I < 8; I++) {
        RESULT += VALORES_AS[I] * pow(X, I);
    }
    RESULT += B * X;
    int RESULTADO_FINAL = round(RESULT);
    //printf("%d", resultf); verifica??o do resultado final
    return RESULTADO_FINAL;
}
/*void main(){
    // for gerado apenas para confirma??o de valores segundo novo vetor
    int i;
    for (i = 1; i < 100; i++){
        int resultado = func_val(i, 0);
        if(resultado != 0){
            printf("%2d\t%d\n", i, resultado);
        }
    }
}*/
```

Disponível em: <https://github.com/wagnerserpaporto/Trabalho-de-Algoritmo.git>

APÊNDICE C – CODIGO EXECUTA

```
#include <stdio.h>
#include <string.h>
#include "decodificador.c"

void main(){
    int TESTES;
    char M_CIFRADA[100];
    printf("Quantos testes: ");
    scanf("%d", &TESTES);
    fflush(stdin);

    int I, CHAVE;
    for (I = 0; I < TESTES; I++){
        printf("\nQual com a chave b: ");
        scanf("%d", &CHAVE);

        fflush(stdin);

        printf("Entre com a mensagem cifrada: ");
        fgets(M_CIFRADA, 100, stdin);
        fflush(stdin);

        separa_e_converte(M_CIFRADA, CHAVE);
    }
}
```

Disponível em: <https://github.com/wagnersepaporto/Trabalho-de-Algoritmo.git>

Link do repositório com o código revisado: <https://github.com/wagnersepaporto/Trabalho-de-Algoritmo.git>

Link do repositório com todos o histórico de elaboração do código e demais materiais de apoio
<https://github.com/wagnersepaporto/Trabalho-N1.git>