Descompilando

Entendendo o código da máquina

Anthony Carneiro da Silva

22 de agosto de 2026

Sumário

Capítulo 1

Índice

1.1 Sobre o autor

Meu nome é Anthony Carneiro da Silva, tenho 22 anos e sou de Sobral/CE. Desde cedo, sempre gostei de usar o computador para jogar videogame e acessar a internet, ainda cedo tive contato com o mundo da programação, mas assim como muitos na época, desde muito novo eu tinha uma pré-concepção de que programação ou desenvolvimento de software fosse algo que só gênios conseguiam fazer.

Aos 17 anos enquanto estava preso a um trabalho que eu já não me sentia mais interessado em fazer, e que sinceramente estava-me desgastando, acabei por novamente dar de cara com o mundo da programação ao ver uma notícia de que Harvard estava oferecendo um curso completamente online e gratuito chamado 'CS50', vendo aquilo acabei por dar uma oportunidade a mim mesmo e experimentar.

Por fim, depois de quase 5 anos, posso dizer que começar a estudar programação foi provavelmente uma das melhores escolhas que fiz na vida, além de claro, ter sido extremamente divertido e desafiador. Hoje em dia, não só tenho certeza que quero trabalhar na área mas também que desejo ser plenamente capaz de repassar tudo que aprendi pessoas novas na área.

Portanto, espero que esse e-book possa ser o meu primeiro grande passo para divulgar o meu conhecimento, que apesar de não muito, pode ajudar quem está começando.

1.2 Sobre o livro

O intuito desse e-book é ajudar iniciantes na linguagem C, que se sentem perdidos e não sabem por onde começar. Quando comecei a estudar sobre programação percebi de cara que maior parte do conteúdo bom estava disponível apenas em inglês, o que não foi um problema para mim que tive uma educação que me auxiliou e permitiu absorver esse tipo de conteúdo, porém, reconheço que nem todos possuíram a mesma oportunidade de aprender inglês ao decorrer da vida.

Aqui você será guiado desde a como preparar a sua máquina para começar a programar, a até algumas dicas e truques mais avançados sobre como a linguagem e o compilador se comportam. Portanto, espero que esse material seja suficiente para preencher todas as lacunas de conhecimento necessárias para poder começar a programar em C sem muitos problemas.

Capítulo 2

Introdução a linguagem C

2.1 História e contexto

A linguagem C foi desenvolvida no início da década de 1970 por Dennis Ritchie com a finalidade de desenvolver sistemas operacionais, drivers e outros softwares de baixo nível. Apesar de ter mais de 50 anos, continua sendo amplamente utilizada em diversos tipos de hardware, desde supercomputadores até microcontroladores e sistemas embarcados.

Com compiladores disponíveis em praticamente todos os sistemas modernos, destacou-se pela portabilidade e foi amplamente empregada na construção de utilitários e aplicações complexas que exigiam maior velocidade e eficiência. Em 1978, foi publicado o livro "The C Programming Language", escrito pelo coautor da linguagem C, que por muito tempo serviu como referência padrão para a linguagem.

Por ser uma linguagem imperativa, com suporte à programação estruturada, amplo escopo lexical e acesso eficiente à memória do computador em baixo nível, ela é frequentemente a escolha ideal para projetos em sistemas com grandes restrições de hardware.

2.2 Benefícios e aplicações práticas

Devido ao seu alto desempenho e à sua natureza como linguagem compilada de baixo nível, a linguagem C apresenta tanto vantagens quanto limitações. Entre suas principais vantagens, destacam-se:

- Alta portabilidade entre diferentes arquiteturas e sistemas
- Alto desempenho comparado a outras linguagens
- Controle direto e eficiente sobre o uso de memória
- Acesso direto a recursos do sistema e ao hardware

Entretanto, essas vantagens vêm acompanhadas de um custo. Essa ideia pode ser substanciada por uma frase que gosto bastante:

> "No começo todos queremos resultados, mas no fim, tudo que queremos é controle"[1]

No cenário atual da indústria de software, frequentemente orientado por prazos curtos e rápida entrega de funcionalidades, observa-se a crescente adoção de linguagens de alto nível. Essas linguagens permitem o desenvolvimento mais ágil, muitas vezes à custa de desempenho, previsibilidade e controle fino sobre os recursos do sistema.

[1] Eskil Steenberg

Nesse contexto, a linguagem C se destaca por sua simplicidade estrutural e pelo baixo nível de abstração. O que acaba transferindo ao programador a responsabilidade por diversos aspectos que, em linguagens mais modernas, são automatizados ou abstraídos por bibliotecas e frameworks, como gerenciamento de memória, manipulação de estruturas de dados e controle explícito de recursos.

Apesar dessa exigência adicional, o domínio da linguagem C proporciona uma compreensão profunda dos fundamentos da computação, incluindo organização de memória, fluxo de execução e interação com o hardware. Esse conhecimento serve como base sólida para o aprendizado de outras linguagens e paradigmas.

Além disso, a linguagem C continua sendo amplamente utilizada em áreas onde desempenho, previsibilidade e controle são essenciais, como:

- Desenvolvimento de sistemas operacionais
- Programação com sistemas embarcados e microcontroladores
- Implementação de drivers de dispositivos
- Construção de compiladores e outras linguagens de programação
- Criação de utilitários e ferramentas do sistema

Capítulo 3

Ambiente de desenvolvimento

3.1 Configuração do ambiente

A configuração do ambiente de desenvolvimento para programar em C é crucial para iniciar projetos com eficiência. Abaixo, são fornecidos passos para configurar o ambiente em sistemas Linux e Windows.

Linux

Para configurar o ambiente de desenvolvimento C em sistemas Linux, siga estes passos:

1. **Instalação do compilador C (GCC):** Em grande parte das distribuições Linux, o GCC já está disponível por padrão. Caso não esteja, ele pode ser facilmente instalado utilizando o gerenciador de pacotes da sua distribuição (como apt para Ubuntu/Debian ou yum para Fedora).

   Seguem abaixo os comandos para instalar o GCC nas distribuições Linux mais famosas:

```
# Debian
sudo apt update
sudo apt install build-essential
# Fedora
sudo dnf update
sudo dnf install gcc
# Arch Linux
sudo pacman -Syu
sudo pacman -S gcc
# openSUSE
sudo zypper refresh
sudo zypper install gcc
# Alpine Linux
sudo apk update
sudo apk add build-base

# Para checar se o GCC foi instalado
# corretamente use o comando abaixo
gcc --version
```

Windows

Para configurar o ambiente de desenvolvimento C em sistemas Windows, siga estes passos:

1. **Instalação do compilador C (GCC):** No Windows, a utilização de ferramentas como o GCC é um pouco mais complexa, pois é necessária a instalação do MinGW

ou MSYS2. Essas ferramentas fornecem um ambiente similar ao Linux para compilar código C.

Para instalar o MinGW, siga os passos abaixo:

(a) Acesse o site oficial do MinGW: `https://osdn.net/projects/mingw/`

(b) Baixe o instalador adequado para o seu sistema (32 ou 64 bits).

(c) Execute o instalador baixado (`mingw-w64-install.exe`).

(d) Durante a instalação, selecione os pacotes desejados. O pacote essencial para o GCC é o `mingw32-gcc` para sistemas 32 bits ou `mingw64-gcc` para sistemas 64 bits.

(e) Siga as instruções do instalador para concluir a instalação.

Após a instalação, é necessário configurar o ambiente para usar o GCC via linha de comando:

(a) Adicione o diretório bin do MinGW ao caminho do sistema:

- Para Windows 10, pesquise "Editar variáveis de ambiente do sistema"na barra de pesquisa e clique em "Variáveis de ambiente". Em "Variáveis do sistema", selecione a variável PATH e clique em "Editar...". Adicione o caminho para o diretório bin do MinGW (exemplo: `C:\MinGW\bin`) e clique em OK.
- Para versões anteriores do Windows, o processo é similar, mas pode variar um pouco, caso você utilize uma versão mais antiga do sistema talvez seja necessário que você faça uma pesquisa a parte para finalizar o processo corretamente.

(b) Verifique se a instalação foi bem-sucedida abra o Prompt de Comando (cmd) e insira o seguinte comando:

```
gcc --version
```

2. **Editor de texto ou IDE:** Instale um editor de texto ou uma IDE (Ambiente de Desenvolvimento Integrado) para escrever e compilar código C. Alguns exemplos populares são o Visual Studio Code, Atom, Sublime Text, ou IDEs como o Code::Blocks ou o Eclipse com plugins para C/C++.

> Um outro editor de texto que é bastante utilizado em ambientes Linux, que possui também, grande vantagens e desvantagens, é o Nvim. Mas no momento, não irei recomendá-lo de usá-lo como editor primário dado a sua dificuldade de adaptação por novos usuários. Porém, aos bem aventurados e dispostos, nada lhe impede de ir atrás de entender o Nvim por conta própria ;)

Se você seguiu os passos acima com sucesso, seu ambiente de desenvolvimento C está configurado e pronto para uso. Agora é o momento de escrever seu primeiro programa!

3.2 Primeiro programa

Após configurar e instalar todas as ferramentas necessárias, é comum começar com um simples "Olá, mundo!".

O programa que iremos fazer exibe o texto "Olá, mundo!" na tela e embora seja simples, é um ponto de partida essencial para começar a entender não somente como a linguagem C funciona, mas qualquer outra linguagem de programação.

Para criar o seu primeiro programa, siga estes passos:

1. Crie um diretório para guardar o seu projeto.
2. Crie um arquivo chamado "main.c" onde será escrito o código.
3. Abra o arquivo "main.c" no seu editor de código e escreva o seguinte:

```c
#include <stdio.h>

int main(void) {
    printf("Olá, mundo!\n");
    return 0;
}
```

4. Salve o arquivo de texto e compile o código com o GCC usando o seguinte comando no seu terminal:

```
gcc main.c
```

5. Agora, execute o programa compilado com o seguinte comando:

```
# Para sistemas Linux
./a.out
# Para sistemas Windows
./a.exe
```

Toda linha de código que comece com \\vai ser entendido como um comentário, e deve ser ignorado pelo compilador. Geralmente, comentários são utilizados pelo programador para descrever aspectos não óbvios, ou mais complexos do programa, tanto para si mesmo como para outras pessoas que possam vir a acessar o código.

Após executar o comando acima, você já deve ver o texto "Olá, mundo!" na sua tela. Mas antes de prosseguir, você provavelmente está se perguntando o que exatamente cada

uma das linhas de código que você escreveu fazem e como exatamente elas fazem esse texto aparecer na sua tela. Por isso, vou explicar passo a passo o que significa cada linha do código acima.

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

Na linha número 1, temos o `include`, uma diretiva que inclui bibliotecas externas no nosso programa. Aqui, estamos incluindo a biblioteca "stdio.h". O conceito de bibliotecas ainda vai ser melhor explicado futuramente, mas você pode entender como se cada biblioteca fosse um tipo de extensão com funções extras.

```
1    int main(void) {
```

Na linha número 2, temos o início de uma função. Nessa linha, há o uso de quatro palavras/caracteres importantes:

- **"int":** Define o que a função deve retornar ao ser finalizada, neste caso, um número inteiro.
- **"main":** É o nome padrão escolhido para identificar o bloco de código a ser executado inicialmente. Apesar de que, em contextos como o de softwares embarcados é possível utilizar outros nomes, **mas isso não será relevante por enquanto!**
- **"(void)":** Define quais parâmetros/argumentos são necessários para chamar a função. Neste caso, a palavra **void** indica que a função não precisa de nenhum valor para ser chamada.
- **"{":** Marca o início de um bloco de código de uma função.

```
1    printf("Olá, mundo!\n");
```

Na linha número 3, encontramos a função `printf`, responsável por exibir informações no terminal. Esta diretiva na verdade não existe como uma funcionalidade padrão na linguagem C, e está sendo declarada através da biblioteca "stdio.h", mencionada na linha 1.

Neste caso a informação que está sendo passada para a função é o texto "Olá, mundo!\n"que sempre deve ser colocado entre os parênteses e as aspas, você pode ter percebido também a presença do caracter "\n" que não aparece quando a mensagem é exibida. Esse caracter é um caracter especial e tem como função fazer com que naquele espaço da mensagem o terminal desça uma linha abaixo antes de continuar.

```
1    return 0;
```

Na linha número 4, encontramos o comando `return`. Em uma função, essa diretiva indica o valor que a função deve retornar quando terminar sua execução e normalmente, mas nem sempre, é acompanhada de um valor que está escrito como um sufixo da diretiva.

No caso da função `main`, que é a função principal em um programa C, o valor 0 geralmente é usado como uma boa prática para indicar que o programa foi finalizado sem erros.

```
1     }
```

A linha número 5 é a chave de fechamento "}", que indica o fim do bloco de código da função `main`. Tudo o que está contido entre as chaves forma o corpo da função, o compilador (GCC) utiliza dessas chaves de abertura e fechamento como uma forma de entender quando um código é pertencente a uma função.

Caso tenha parecido um pouco confuso a primeira vista, acredito que com um pouco de prática você vai começar a escrever esse tipo de código sem sequer pensar direito sobre isso. E no geral, essa estrutura tende a se repetir bastante, então caso não tenha entendido algo muito bem vale a pena voltar e reler.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Modifique a mensagem exibida na tela pela função `printf`
2. Tente alterar a posição do caracter especial "\n" e veja o que acontece
3. Mudar o nome da função para qualquer outra coisa e veja qual o erro retornado
4. Remova a linha de código que inclui a biblioteca "stdio.h" e veja qual o erro retornado

Capítulo 4

Conceitos básicos de programação

4.1 Operadores e variáveis

4.1.1 Operadores

Um operador é um símbolo que indica a realização de uma operação entre valores de um programa. Na linguagem C é possível encontrar alguns diferentes tipos de operadores, dentre eles:

Aritméticos

Os operadores aritméticos são utilizados para representar operações matemáticas entre valores. Segue abaixo uma lista de todos os operadores aritméticos:

Operador	Descrição	Exemplo	Resultado
+	soma	10 + 20	30
-	subtração	20 - 10	10
*	multiplicação	10 * 2	20
/	divisão	20 / 10	2
%	módulo	21 % 10	1
++	incremento	++10 é o mesmo que 10 + 1	11
–	decremento	–10 é o mesmo que 10 - 1	9

O operador de módulo realiza uma divisão entre os dois valores e retorna o resto da operação.

Relacionais

Os operadores relacionais possuem a finalidade de fazer comparações entre valores, como por exemplo, verificar se um número é maior que outro. O resultado de uma comparação sempre vai ser `verdadeiro` ou `falso`.

Segue abaixo uma lista dos comparadores relacionais:

Operador	Descrição	Exemplo	Resultado
==	igual	10 == 10	verdadeiro
!=	diferente	10 != 10	falso
>	maior que	10 > 10	falso
>=	maior ou igual	10 >= 10	verdadeiro
<	menor que	10 < 10	falso
<=	menor ou igual	10 <= 10	verdadeiro

Lógicos

Os operadores lógicos são usados em conjunto com os operadores relacionais. Para compreendê-los corretamente, é importante entender que na linguagem C os valores de `verdadeiro` e `falso` podem ser representados pelos números 1 e 0, respectivamente.

Ao utilizar os operadores lógicos, podemos criar verificações de valores mais complexas. Abaixo está a lista dos operadores existentes e como usá-los:

Operador	Descrição	Exemplo	Resultado
&&	AND lógico	1 && 1	1
||	OR lógico	1 || 0	1
!	NOT lógico	!1	0

Caso tenha ficado um pouco confuso como cada operador atua exatamente, vou tentar detalhar de maneira mais sucinta e com mais exemplos de como cada operador deve agir em cada caso.

1. **AND lógico:** Retorna verdadeiro apenas se ambos os valores forem verdadeiros.

Exemplo	Resultado
0 && 0	0
0 && 1	0
1 && 0	0
1 && 1	1

2. **OR lógico:** Retorna verdadeiro se pelo menos um dos valores for verdadeiro.

Exemplo	Resultado
0 || 0	0
0 || 1	1
1 || 0	1
1 || 1	1

3. **NOT lógico:** Inverte o valor lógico; verdadeiro se torna falso e vice-versa.

Exemplo	Resultado
!0	1
!1	0

Atribuição

Os operadores de atribuição tem a função de atribuir valores a variáveis. Por exemplo, utilizando um operador de atribuição poderíamos dizer que A = 10 então toda vez que utilizássemos o valor A no código de um programa, ele seria interpretado como o número 10.

Operador	Descrição	Exemplo	Equivale
=	Atribuição de valor a variável	A = 10	A = 10
+=	Atribuição com soma	A += 10	A = A + 10
-=	Atribuição com subtração	A -= 10	A = A - 10
*=	Atribuição com multiplicação	A *= 10	A = A * 10
/=	Atribuição com divisão	A /= 10	A = A / 10
%=	Atribuição com divisão	A %= 10	A = A % 10

4.1.2 Variáveis e tipos de dados

Variáveis

Variáveis podem ser entendidas como objetos com nomes personalizados, capazes de armazenar valores que serão utilizados pelo programa durante sua execução. A declaração de uma variável geralmente segue o seguinte padrão:

```
<tipo da variável> <nome da variável>;
```

Os nomes das variáveis podem conter letras e números, além de poderem ser iniciados com um sublinhado. No entanto, algumas regras devem ser seguidas para declarar uma variável sem problemas.

Na linguagem C, algumas palavras já são reservadas para representar diretivas e, portanto, não podem ser usadas como nomes de variáveis. Abaixo, segue uma tabela das palavras reservadas:

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Você não precisa necessariamente memorizar todas essas diretivas neste momento; isso é algo que você absorverá e memorizará com bastante prática e um pouco de tempo.

Tipos de dados

Um aspecto que ainda não abordamos é a 'tipagem' de variáveis. Embora cada variável tenha um nome definido, o computador não sabe exatamente o que há dentro de cada uma delas. Portanto, cabe ao usuário fazer essa definição.

A princípio, a linguagem C fornece quatro tipos básicos: **char**, **int**, **float** e **double**. Além disso, há modificadores como **signed**, **unsigned**, **short** e **long**, que podem ser usados em conjunto com cada tipo.

Novamente, não é necessário memorizar tudo isso agora, mas por curiosidade, na próxima página terá uma lista descrevendo os tipos e modificadores, juntamente com algumas informações adicionais:

Tipo	**Descrição**	**Tamanho (bits)**	**Formatador**	**Alcance**
char	Representa um byte de dados. Pode ser utilizado para caracteres ou valores numéricos pequenos. Pode ser *signed* ou *unsigned*.	8	%c (ou %d)	-128 a 127 (signed) ou 0 a 255 (unsigned)
int	Tipo inteiro padrão, utilizado para operações aritméticas gerais.	32	%d (ou %i)	aproximadamente -2^{31} a $2^{31}-1$
float	Representa números de ponto flutuante de precisão simples.	32	%f	$\approx 1.17\times10^{-38}$ a 3.40×10^{38}
double	Representa números de ponto flutuante de dupla precisão.	64	%lf	$\approx 2.22 \times 10^{-308}$ a 1.79×10^{308}

Além dos tipos de dados descritos acima, segue abaixo, uma descrição mais detalhada do que cada um dos modificadores de tipo fazem:

Modificador	**Descrição**
signed	Indica que o tipo suporta valores negativos (com sinal). Geralmente é o padrão para tipos inteiros.
unsigned	Indica que o tipo armazena apenas valores não negativos, dobrando o limite superior positivo.
short	Reduz o tamanho de um inteiro (ex: `short int`). Garante no mínimo 16 bits.
long	Aumenta o tamanho de um inteiro (ex: `long int`). Garante no mínimo 32 bits.
long long	Inteiro de maior capacidade, garantindo no mínimo 64 bits.

Agora, saindo um pouco da teoria e partindo para a prática, vamos aplicar os conceitos aprendidos em um programa simples.

```c
#include <stdio.h>

int main(void) {
    // Declara variáveis para guardar valores
    // necessários para realizar os cálculos
    float x;
    float y;
    float area;

    // Pede ao usuário por valores de largura e altura
    printf("Digite a altura do retângulo: ");
    scanf("%f", &x);

    printf("Digite a comprimento do retângulo: ");
    scanf("%f", &y);

    // Calcula a área do triângulo
    area = x * y;

    // Escreve o resultado no terminal
    printf("A área do retângulo é: %f\n", area);

    return 0;
}
```

O código acima funciona da seguinte forma: ao ser iniciado, o usuário deve digitar no terminal os valores da altura e comprimento de um retângulo imaginário. Em seguida, o programa calcula automaticamente a área do retângulo e exibe o resultado.

```c
    float x;
    float y;
    float area;
```

No trecho abaixo, declaramos três variáveis diferentes: `x` para a altura, `y` para o comprimento e `area` para o resultado do cálculo. Todas as três variáveis são do tipo **float**, pois nada impede que as medidas do retângulo sejam números fracionados.

```c
    printf("Digite a altura do retângulo: ");
    scanf("%f", &x);

    printf("Digite a comprimento do retângulo: ");
    scanf("%f", &y);
```

Aqui utilizamos a função `printf` para exibir texto na tela, assim como já feito antes, e em seguida, usamos uma nova função chamada `scanf` para receber informações do usuário pelo terminal.

Dentro de scanf, passamos dois parâmetros: o primeiro (`%f`) é o formatador, que especifica o tipo de valor esperado para ser recebido, neste caso, uma **float**. O segundo (`&x`) é onde queremos armazenar o valor recebido; a variável vem com o sufixo & porque estamos passando o endereço da variável.

> O endereço de uma variável é um tópico que será abordado com maior foco no capítulo 5.1.1, porém, no momento você pode assumir que ele diz respeito a localização da variável na memória do computador.

```
1         area = x * y;
```

Após receber os valores de x e y usando a função `scanf`, multiplicamos as variáveis usando o operador de multiplicação e armazenamos o resultado em `area` usando o operador de atribuição (`=`).

```
1         printf("A área do retângulo é: %f\n", area);
2         return 0;
```

Finalmente, exibimos o valor na tela usando a função `printf` e retornamos o valor 0 para indicar que o programa foi encerrado com sucesso. É importante lembrar que, ao exibir uma variável do tipo **float**, é necessário utilizar o formatador `%f`.

Entendendo bem o código acima, é possível praticar os conceitos de operadores e variáveis sem muitas dificuldades.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Modifique a lógica do programa acima para calcular a área de um triângulo.
2. Crie uma calculadora simples capaz de receber 2 números e realizar alguma das 4 operações básicas.
3. Crie um conversor de unidade de metro para centímetros.

4.1.3 Conversão de tipos

Normalmente referido como “casting”, as conversões de tipos de dados podem ser feitas com as mais diferentes finalidades.

Conversões implícitas

Ao avaliar uma expressão, o compilador acaba por realizar um processo de conversão de dados que respeita uma regra chamada **promoção automática** onde todas as variáveis são convertidas para um mesmo tipo.

Essa regra funciona de forma que os tipos de menor tamanho são convertidos para os tipos de maior tamanho que existem dentro da expressão, essa hierarquia, por assim dizer, de tipos segue a seguinte ordem:

```
char < short < int < long < long long < float < double < long double
```

Para ilustrar melhor o processo de promoção automática vamos usar o código abaixo como exemplo:

```
1   int main(void) {
2       char x = 'A' ;
3       int j;
4       float y;
5       int a = 10;
6       float b = 2;
7
8       j = x - 65; // 'x' é convertido de char para int
9       y = a / b; // 'a' é convertido para float
10      return 0;
11  }
```

As conversões acima acabam acontecendo pois estamos realizando cálculos com valores de diferentes tipos, o que acaba forçando o compilador a fazer essa conversão para que os cálculos sejam realizados sem erros. Essa conversão se mostra obrigatória devido a forma que o computador guarda e interpreta esses valores na memória.

> Uma explicação mais detalhada sobre o assunto será dada em um capítulo posterior, o que você precisa saber no momento é que a fim de evitar erros o compilador prefere evitar realizar operações entre valores de diferentes tipos.

Conversões explícitas

As conversões explícitas são bem mais simples de entender e de se usar. Contudo, para usuários mais experientes podem ser uma ferramenta bem poderosa e complicada.

```
1  int main(void) {
2      float y = 10.01;
3      int x;
4
5      x = (int)y;
6      return 0;
7  }
```

Como o valor de x é de um tipo menor que y é necessário realizar essa conversão para int utilizando "(int)" antes do valor que será atribuído a x. Dessa forma também, já que x é incapaz de guardar valores quebrados o valor de 0.01 será perdido.

Portanto, sempre que for necessário realizar uma conversão de tipos em uma expressão basta seguir esse padrão:

```
(tipo a ser convertido) valor a ser convertido
```

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Tente compilar o código acima sem converter o valor de 'y'.

2. Declare duas variáveis do tipo 'float', realize um cálculo entre ambas, converta o resultado para 'int' e imprima o valor

3. Declare uma variável do tipo 'float', uma do tipo 'int' e realize um cálculo entre ambas que deve ter o resultado guardado em uma variável do tipo 'short int' para ver o que acontece

4.2 Funções e estrutura do código

4.2.1 Funções

Apesar de talvez não ter notado, desde o seu primeiro código você vem lidando com funções. Funções nada mais são do que blocos de códigos capazes de executar uma sequência específica de instruções utilizadas para realizar uma melhor modularização do código, ou seja, separar o código "em pedaços" com diferentes funções bem definidas.

As funções em C geralmente seguem o seguinte padrão de declaração:

```
1    tipo_do_retorno nome_da_função(argumentos)
2    {
3    // Código da função
4    }
```

O tipo do retorno identifica o tipo de valor que a função deve retornar, caso necessário. Este valor pode ser de qualquer tipo já apresentado até o momento, como inteiros, caracteres, ponto flutuante, entre outros.

Vamos considerar um exemplo simples para melhor entender a declaração de uma função:

```
1    #include <stdio.h>
2
3    void printHello(void) // Define uma nova função
4    {
5        printf("Hello!\n");
6    }
7
8    int main(void) // Define a função principal
9    {
10       printHello();
11       printHello();
12       printHello();
13       return 0;
14   }
```

Este código é um exemplo básico de utilização de função. Ele define uma função chamada `printHello`, responsável por imprimir o texto ”Hello!”três vezes consecutivas, sem necessidade de receber ou retornar qualquer valor.

Um detalhe interessante sobre funções em C é que, pelo fato da linguagem não depender de indentação, é possível declarar funções em apenas uma linha, independentemente do seu tamanho.

Isso não significa necessariamente que você deve fazer isso em todos os casos, mas pode ser útil em algumas situações específicas. Se não ficou claro como isso é possível, aqui está um exemplo utilizando a mesma função `printHello`:

```
1    void printHello(void) { printf("Hello!\n"); }
```

Se você substituir o código anterior por essa versão de apenas uma linha, o código continuará sendo compilado normalmente.

Recebendo e retornando Valores

As funções podem retornar valores após a execução de suas operações. Isso permite que elas comuniquem resultados de volta para o código que as chamou. Vamos elaborar outro exemplo para demonstrar isso:

```
1   #include <stdio.h>
2
3   // Define uma nova função que retorna
4   // o valor da soma de dois valores
5   int soma(int a, int b)
6   {
7       return a + b;
8   }
9
10  int main(void)
11  {
12      int resultado = soma(3, 4);
13      printf("A soma é: %d\n", resultado);
14      return 0;
15  }
```

Neste exemplo, a função `soma` recebe dois argumentos inteiros e retorna a soma deles. No `main`, chamamos essa função com os argumentos 3 e 4, atribuindo o resultado à sua variável, que é então impressa na tela. É importante observar que os argumentos devem ser inseridos na mesma ordem em que foram declarados na função.

Os conceitos acima explicados são fundamentais para entender o funcionamento das funções em C e como elas podem ser utilizadas para tornar o código mais modular e eficiente. Ademais, o recomendado é que se utilize de bastante prática testes para ter certeza que você entendeu tudo que foi explicado.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Modifique os valores passados para a função soma para gerar diferentes resultados.
2. Tente alterar os valores passados para a função soma para caracteres ou símbolos (‘c’, ‘$’, etc.) e analise o comportamento do código.
3. Modifique a função de maneira que seja possível somar 3 valores ao invés de 2.
4. Elabore mais 3 funções que seja possível realizar operações de subtração, multiplicação, divisão e tente utilizá-las ao invés da função soma.

4.2.2 Incluindo bibliotecas e outros arquivos

Ao desenvolver aplicações de maior complexidade, é comum recorrer a bibliotecas que já contêm funções pré-desenvolvidas para simplificar o processo de desenvolvimento. Neste capítulo, vamos abordar os conceitos fundamentais necessários para trabalhar com a utilização e implementação de bibliotecas em um projeto.

Bibliotecas estáticas

As bibliotecas estáticas consistem em segmentos de código incorporados ao seu código durante o processo de compilação. Elas são acopladas diretamente ao seu programa, de modo que passam a fazer parte do executável produzido pelo compilador.

Em geral, as bibliotecas estáticas são as mais simples de usar, pois não exigem que o usuário instale ou gerencie quaisquer outros arquivos além dos do seu próprio código para compilar o código de forma independente.

Exemplos de bibliotecas estáticas incluem: `stdio.h, string.h e stdlib.h`. Todas essas bibliotecas são, na verdade, arquivos localizados em um diretório padrão do seu sistema, que são instalados automaticamente com o compilador `gcc`. Em sistemas Linux, esses arquivos podem ser encontrados no diretório: `/usr/include/`. Já em sistemas Windows, a localização desses arquivos dependerá do método que você usou para instalar o `gcc`, portanto, recomendo que você pesquise por conta própria na internet.

Aqui está um exemplo comum de uso de bibliotecas estáticas:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Olá, mundo!\n");
6      return 0;
7  }
```

Como você pode ver, já na primeira linha, estamos incluindo a biblioteca `stdio.h` no código para poder usar a função `printf`.

Utilizando bibliotecas dinâmicas

As bibliotecas dinâmicas, também conhecidas como bibliotecas compartilhadas, são um conjunto de funções e procedimentos carregados durante a execução, em vez de serem vinculados ao programa durante a compilação. Isso significa que o código contido nessas bibliotecas não é incorporado diretamente no executável final do programa.

A principal vantagem das bibliotecas dinâmicas é que elas permitem que vários programas compartilhem o mesmo código de biblioteca na memória, economizando espaço e recursos do sistema, pois aquele código não vai precisar estar incluso no executável de todos os

programas que a utilizam. Além disso, as bibliotecas dinâmicas permitem que os programas sejam atualizados ou modificados, sem a necessidade de recompilar ou relinkar os programas que as utilizam.

No entanto, o uso de bibliotecas dinâmicas também tem suas desvantagens. Por exemplo, se um programa depende de uma biblioteca dinâmica específica e essa biblioteca não está disponível no sistema onde o programa está sendo executado, o programa não será executado corretamente. Além disso, o uso de bibliotecas dinâmicas pode tornar o processo de compilação e linkagem mais complexo.

Em resumo, as bibliotecas dinâmicas são uma ferramenta poderosa que permite aos desenvolvedores compartilhar e reutilizar código de maneira eficiente. No entanto, eles também exigem um entendimento cuidadoso de como eles funcionam para evitar potenciais problemas.

Para exemplificar a utilização de bibliotecas externas, vamos considerar o seguinte código:

```c
#include <stdio.h>
#include <math.h>

int main(int argc, char **argv)
{
    // Armazena os valores
    int c1 = 3;
    int c2 = 4;

    // Realiza os cálculos
    float h = sqrt(pow(c1, 2) + pow(c2, 2));

    // Exibir resultados
    printf("Valor da hipotenusa: %.2f\n", h);
    return 0;
}
```

Este código calcula a hipotenusa de um triângulo com base nos valores fornecidos pelas variáveis `c1` e `c2`. No entanto, as funções `pow` e `sqrt`, responsáveis por calcular a potência e a raiz quadrada de um número, respectivamente, não estão definidas em nenhum lugar do código.

Ambas as funções estão incluídas na biblioteca `math.h`, que contém várias funções relacionadas a operações matemáticas mais complexas que não estão incluídas por padrão na linguagem C e que poderiam ser difíceis de implementar manualmente. Dito isso, se você tentar compilar o código acima usando o comando:

```
gcc -o main main.c
```

É provável que você veja um erro semelhante ao seguinte:

```
/usr/bin/ld: /tmp/ccF2H2AC.o: in function ‘main’:
main.c:(.text+0x3d): undefined reference to ‘pow’
/usr/bin/ld: main.c:(.text+0x66): undefined reference to ‘pow’
/usr/bin/ld: main.c:(.text+0x7e): undefined reference to ‘sqrt’
collect2: error: ld returned 1 exit status
```

Isso indica que as funções `pow e sqrt` não estão sendo encontradas pelo compilador.

Para corrigir o erro, você precisa informar ao compilador que pretende compilar o código usando uma biblioteca específica. Neste caso, como estamos incluindo a biblioteca `math`, devemos adicionar "-lm" ao comando de compilação, conforme mostrado abaixo:

```
gcc -o main main.c -lm
```

Dessa forma, o compilador reconhecerá a biblioteca e realizará a compilação corretamente.

Criando uma biblioteca

Um aspecto interessante das bibliotecas é que você pode criar as suas próprias para implementar funções com uma quantidade mínima de código.

Para demonstrar isso vamos supor uma situação bem simples. Vamos supor que você está incomodado de ter que adicionar o 'f' no final da chamada da função `printf` e que você também não quer ter que se lembrar de adicionar o `\n` ao final do texto impresso para ter que pular para a próxima linha e para resolver isso você resolveu implementar a função `print`.

```
1   // Arquivo minha_biblioteca.c
2   #include <stdio.h>
3
4   void print(char *text)
5   {
6       printf("%s\n", text);
7   }
8
9   // Arquivo minha_biblioteca.h
10
11  #ifndef MINHA_BIBLIOTECA_H
12  #define MINHA_BIBLIOTECA_H
13
14  void print(char *text);
15
16  #endif
17
18  // Arquivo main.c
19
20  #include "minha_biblioteca.h"
21
22  int main(void)
23  {
24      print("A função print está funcionando");
25      print("Esse texto deve ser impresso na linha abaixo");
26  }
```

Vamos supor que o código acima esteja dividido em três arquivos diferentes, localizados no mesmo diretório, com os respectivos nomes indicados nos comentários acima. Primeiramente, é importante notar a maneira como o arquivo `minha_biblioteca.h` foi incluído no código do arquivo "main.c". Como o nome da biblioteca foi colocado entre aspas, e não entre '<>', isso indicará ao compilador que estamos tentando incluir um arquivo localizado no mesmo diretório, e não no diretório padrão de bibliotecas.

Além disso, é importante notar que, em vez de incluir diretamente o arquivo `minha_biblioteca.c`, incluímos o arquivo `minha_biblioteca.h`. Arquivos '.h' são chamados de "headers", que são basicamente arquivos que contêm os protótipos das funções que pretendemos chamar. Uma vez incluídos esses protótipos, o compilador capaz de lidar sozinho com todo o trabalho de incluir os arquivos necessários para que o código seja compilado com sucesso.

Além disso, caso você tenha observado atentamente é possível perceber a inclusão das seguintes linhas de código:

```
1    #ifndef MINHA_BIBLIOTECA_H
2    #define MINHA_BIBLIOTECA_H
3    ...
4    #endif
```

Essas três linhas são marcadores que, embora não sejam obrigatórios, são extremamente importantes, pois evitam que uma mesma biblioteca seja incluída repetidamente de maneira desnecessária. Para entender melhor como isso funciona, vamos analisar o que cada uma dessas linhas significa.

```
1    #ifndef MINHA_BIBLIOTECA_H
```

Verifica se `MINHA_BIBLIOTECA_H` ainda não foi definido. Se já tiver sido definido, a inclusão do restante do código é cancelada.

```
1    #define MINHA_BIBLIOTECA_H
```

Define `MINHA_BIBLIOTECA_H` e inclui todo o código abaixo.

```
1    #endif
```

Delimita o fim do bloco que deve ser definido junto a `MINHA_BIBLIOTECA_H`.

No trecho de código acima, você pode entender `MINHA_BIBLIOTECA_H` como sendo uma variável booleana, pois quando utilizada desta forma ela possui apenas dois estados, sendo esses definido e não definido.

O nome que se pode dar é de escolha do usuário, não tendo que seguir obrigatoriamente nenhum tipo de padrão, porém, por convenção e padronização o costume é que se utilize, sempre em maiúsculo, o nome da biblioteca, seguido de '_H' para identificar que se refere a um arquivo do tipo header (.h).

Depois de compreender corretamente os conceitos explicados acima, podemos passar para o processo de compilação do código, que vai mudar um pouco devido aos múltiplos arquivos. Para compilar o código acima, usaremos o seguinte comando:

```
gcc -o main main.c minha_biblioteca.c
```

Com isso, o compilador deve processar todos os arquivos e gerar um executável com sucesso.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Modifique o nome da biblioteca para qualquer outra coisa e tente compilar o código da mesma forma
2. Tente criar uma biblioteca para executar alguma outra função que você ache interessante
3. Tente compilar o código sem incluir a biblioteca que você criou no comando de compilação

4.2.3 Escopo

Embora intimamente relacionado às funções, o escopo de variáveis é um conceito que merece atenção própria. Em termos gerais, o escopo refere-se a um conjunto de regras que determinam como as variáveis podem ser usadas em um programa.

Variáveis locais

Variáveis locais são aquelas que só podem ser acessadas em uma função ou bloco de código específico. Em outras palavras, uma vez declaradas numa função, elas não podem ser utilizadas ou modificadas em outros contextos, e deixam de existir ao término da função.

Para ilustrar essas propriedades das variáveis locais, considere o seguinte exemplo:

```
1   #include <stdio.h>
2
3   void printnum(int x) { // Recebe argumento 'x'
4   printf("\%i\n", x);
5
6   int y = 369; // Cria variável local
7   printf("\%i\n", y);
8   }
9
10  int main (void) {
11      printnum(369);
12      return 0;
13  }
```

Neste código, a função `printnum` imprime dois valores na tela: o argumento x passado para a função e a variável local y declarada dentro dela. Ambos os valores são considerados variáveis locais, pois só podem ser acessados e modificados dentro da função em que foram declarados.

> Em alguns contextos, as variáveis locais recebidas como argumentos são chamadas de "parâmetros locais", mas, para fins de entendimento e praticidade, não há necessidade de distinção, uma vez que ambas são manipuladas da mesma forma.

Variáveis globais

Similarmente às variáveis locais, as variáveis globais podem ser acessadas e modificadas por funções ou blocos de código. No entanto, elas têm a característica de serem acessíveis de qualquer parte do código e permanecerem em existência após o término de um bloco ou função.

Em aplicações simples, variáveis globais podem ser extremamente úteis, pois eliminam a necessidade de passar valores entre funções para realizar determinadas tarefas. No entanto, em aplicações mais complexas, o uso excessivo de variáveis globais pode levar a problemas de controle de acesso, já que diferentes partes do programa podem tentar acessar ou modificar a mesma variável simultaneamente, o que pode acabar gerando algumas inconveniências.

Apesar de inicialmente intimidador, o uso de variáveis globais é uma prática que deve ser dominada, pois é fundamental para compreender o fluxo de dados em programas mais complexos.

Considere o seguinte exemplo prático de como uma variável global pode ser utilizada em um projeto real, como para controlar a vida do personagem em um jogo:

```c
#include <stdio.h>

unsigned int life = 100; // Vida do personagem

// Danifica a vida do personagem
void lifeDamage(int points) {
    life -= points;
}

// Recupera a vida do personagem
void lifeHeal(int points) {
    life += points;
}

// Exibe a vida do personagem
void lifePrint(void) {
    printf("Vida: \%i\n", life);
}

int main(void) {
    lifePrint();

    lifeDamage(10);
    lifePrint();

    lifeHeal(10);
    lifePrint();

    // Modifica o valor da vida diretamente
    life = 10;
    lifePrint();

    return 0;
}
```

Neste código, a variável `life` é uma variável global que armazena a vida do personagem. Ela pode ser acessada e modificada por todas as funções do programa, que simplifica o controle do estado do personagem ao longo do jogo. É notável também que, ao contrário das variáveis locais, uma variável global mantém seu valor durante toda a execução do programa.

A diretiva static

Variáveis precedidas pela diretiva `static` mantêm seus valores mesmo quando saem de escopo. Isso significa que, ao executar uma função que contém uma variável `static`, a variável é inicializada apenas na primeira vez que a função é chamada, e seu valor é preservado para chamadas subsequentes.

Variáveis `static` são alocadas em uma área da memória do programa conhecida como `data`, enquanto todas as outras variáveis, com exceção das variáveis globais comuns que aprendemos a declarar até agora são alocadas em uma área conhecida como `stack`. Por isso, as variáveis `static` são inicializadas com o valor 0 por padrão, a menos que o usuário as inicialize explicitamente com um valor diferente.

O código a seguir ilustra essa propriedade:

```
1    #include <stdio.h>
2
3    int main(void)
4    {
5        static int x;
6        int y;
7        printf("x: \%d\n", x);
8        printf("y: \%d\n", y);
9    }
```

Ao compilar e executar o código acima você vai notar que a variável y terá um valor aleatório, enquanto a variável x sempre terá o valor 0. Isso ocorre porque, quando uma variável é alocada no espaço `data` da memória, seu valor deve ser constante.

Usemos esse código como exemplo:

```
1    int initializer(void)
2    {
3        return 50;
4    }
5
6    int main(void)
7    {
8        // Inicializa o valor de x com o valor retornado pela
            função
9        static int x = initializer();
10       printf("x: \%d\n", x);
11
12       return 0;
13   }
```

Ao compilar o código acima você verá um erro semelhante a:

```
main.c: In function ‘main’:
main.c:10:24: error: initializer element is not constant
10 |         static int x = initializer();
|                        ^~~~~~~~~~~
```

Isso se deve ao fato de que o valor de x não é conhecido durante a compilação do programa, sendo assim dependente de outra função ou variável para ser atribuído.

> Em relação aos espaços `heap` e `data` da memória, esses conceitos serão discutidos com mais detalhes no capítulo 5.2

Além disso, é possível declarar variáveis `static` no contexto de variáveis globais e funções. Quando isso é feito, o compilador limita o uso dessa variável (ou função) ao arquivo em que foi declarada. Por exemplo, em programas mais complexos que são divididos em vários arquivos com diferentes segmentos de código, pode haver variáveis globais e funções com o mesmo nome, mas com propósitos diferentes. Nesses casos, a diretiva `static` pode ser útil.

O código a seguir ilustra o uso de variáveis globais `static`:

```
1  // Arquivo: main.c
2  #include <stdio.h>
3
4  static int x = 10;
5
6  int main(void)
7  {
8      printf("x in main.c: \%d\n", x);
9      printX();
10     return 0;
11 }
12
13 // Arquivo: example.c
14 #include <stdio.h>
15
16 static int x = 20;
17
18 void printX(void)
19 {
20     printf("x in example.c: \%d\n", x);
21 }
```

E aqui está um exemplo de código que ilustra o uso de funções `static`:

```c
// Arquivo: main.c
#include <stdio.h>

static void printHello(void)
{
    printf("Hello from main.c!\n");
}

int main(void)
{
    printHello();
    return 0;
}

// Arquivo: example.c
#include <stdio.h>

static void printHello(void)
{
    printf("Hello from example.c!\n");
}
```

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Escreva um programa que declare duas variáveis com o mesmo nome, uma global e uma local em uma função que não seja a `main`. O que acontece quando você tenta imprimir o valor dessas variáveis dentro da função? E fora da função?

4.2.4 Condicionais e loops

Através de condicionais e loops, é possível criar programas que executam seções específicas de código com base em determinadas condições definidas no próprio código.

Condicionais

Condicionais são estruturas de controle que direcionam o fluxo de execução do programa. Este conceito pode ser exemplificado diretamente com um código:

A estrutura `if` geralmente segue o seguinte formato:

```
1     if (condição) {
2         // Código a executar se condição for verdadeira...
3     }
```

A condição fornecida entre parênteses para o `if` deve ser uma expressão que resulta em verdadeiro ou falso. Dependendo do resultado da expressão, o bloco de código entre chaves será executado ou ignorado.

É possível utilizar valores constantes como 0 ou 1 para representar verdadeiro e falso, respectivamente. No entanto, é redundante utilizar uma expressão que sempre resulta em verdadeiro ou falso. Embora esse tipo de operação seja raramente utilizado, segue um exemplo para ilustrar o conceito:

```
1     #include <stdio.h>
2
3     int main(void)
4     {
5         if (0) { // 0 representa falso printf("Esse texto nunca
              será impresso\n");
6         }
7         if (1 > 2) { // 1 nunca será maior que 2
8             printf("Esse texto nunca será impresso\n");
9         }
10        if (1) { // Qualquer número que não seja 0 representa
              verdadeiro
11            printf("Esse texto sempre será impresso\n");
12        }
13        if (10 > 0) { // 10 sempre será maior que 0
14            printf("Esse texto sempre será impresso\n");
15        }
16
17        return 0;
18    }
```

Neste exemplo, as verificações produzem resultados constantes, ou seja, o resultado não varia independentemente do número de vezes que o código é executado.

Outra diretiva importante de se entender é o `else`, que complementa o `if`. O bloco de código dentro do `else` é executado se a condição do `if` for falsa. Um exemplo prático ilustra sua utilização:

```c
#include <stdio.h>

int main(void)
{
    int number = 0;
    printf("Digite um número: ");
    scanf("\%d", &number);

    if (number > 0) {
        printf("Número é maior que 0\n");
    } else {
        printf("Número não é maior que 0\n");
    }

    return 0;
}
```

Neste código, se o número digitado for maior que 0, o primeiro bloco de código será executado; caso contrário, o bloco de código dentro do `else` será executado.

Além disso, é possível criar fluxos de execução mais complexos utilizando `if` dentro de `else` para definir condições adicionais. Apesar de ser complexo de entender apenas com explicações textuais, um exemplo de código esclarecerá o conceito:

```c
#include <stdio.h>

int main(void)
{
    int number = 0;
    printf("Digite um número: ");
    scanf("\%d", &number);

    if (number < 0) {
        // Executa bloco se número for menor que 0
        printf("Número é menor que 0\n");
    } else if (number > 0) {
        // Executa bloco se número for maior que 0
        printf("Número é maior que 0\n");
    } else {
        // Caso nenhuma das condições sejam
        // verdadeiras, o número só pode ser igual a 0
        printf("Número é exatamente igual a 0\n");
    }

    return 0;
}
```

Neste exemplo, além de verificar se o número é maior ou menor que 0, o programa também identifica quando o número é exatamente 0. Vale ressaltar que, ao utilizar `else if`, a expressão dentro dos parênteses só é avaliada se as condições anteriores forem falsas.

Em situações em que é necessário avaliar cada condição individualmente, vários `if` consecutivos são utilizados. Como por exemplo:

```c
#include <stdio.h>

int main(void)
{
    int number = 120;

    if (number > 0) {
        printf("Número é maior que 0\n");
    }
    if (number > 10) {
        printf("Número é maior que 10\n");
    }
    if (number > 100) {
        printf("Número é maior que 100\n");
    }
    if (number > 1000) {
        printf("Número é maior que 1000\n");
    }

    return 0;
}
```

Ao executar o código acima, você deve ver algo como:

```
Número é maior que 0
Número é maior que 10
Número é maior que 100
```

Isso ocorre porque cada bloco `if` é independente dos outros; portanto, todos são avaliados individualmente. Se fosse utilizado um `else` dentro dos `if`, como no exemplo a seguir:

```c
#include <stdio.h>

int main(void)
{
    int number = 120;

    if (number > 0) {
        printf("Número é maior que 0\n");
    } else if (number > 10) {
        printf("Número é maior que 10\n");
    } else if (number > 100) {
        printf("Número é maior que 100\n");
    } else if (number > 1000) {
        printf("Número é maior que 1000\n");
    }

    return 0;
}
```

O resultado seria apenas:

```
Número é maior que 0
```

Isso ocorre porque a primeira condição já é avaliada como verdadeira, e as condições subsequentes dependem da falsidade das condições anteriores, o que não ocorre neste caso.

While

A estrutura de loop mais simples é criada com a diretiva `while`. Essa estrutura é responsável por repetir uma seção de código enquanto uma expressão for verdadeira.

```c
while (expressão) {
    // Bloco de código a ser executado
}
```

Vamos ilustrar um simples caso de uso para a estrutura while:

```c
#include <stdio.h>

int main(void)
{
    int x = 0;

    while (x != 10) {
        printf("x: %d\n", x);
        ++x; // Incrementa o valor de x em 1
    }

    return 0;
}
```

Neste exemplo, criamos uma estrutura `while` que verifica se x é diferente de 10. Enquanto o valor for diferente de 10, o código entre chaves é repetido. Quando executado, este código produzirá a seguinte saída:

É importante mencionar que o loop `while` está verificando apenas se o valor é diferente de 10, o que poderia ocasionar em erros caso fosse possível o valor de `x` pular de 9 para 11, caso ele fosse incrementado de 2 em 2 por exemplo.

No caso acima isso não é possível mas deve se ficar atento a casos específicos como esse em aplicações reais que usem algum looping do tipo, em caso de dúvida vale sempre a pena usar comparações como:

```c
while (x < 10) {
    ...
}
```

Assim, você garante a quebra do looping ao chegar no valor 10 e evita casos específicos em que seja possível pular de 9 para qualquer valor maior que 10.

```
x: 0
x: 1
x: 2
x: 3
x: 4
x: 5
x: 6
x: 7
x: 8
x: 9
```

Essa funcionalidade pode ser usada para criar algoritmos mais complexos, repetindo um mesmo código várias vezes sem a necessidade de escrever mais linhas de código.

Um código equivalente, porém menos eficiente, do ponto de vista da criação, seria:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("x: 0\n");
6      printf("x: 1\n");
7      printf("x: 2\n");
8      printf("x: 3\n");
9      printf("x: 4\n");
10     printf("x: 5\n");
11     printf("x: 6\n");
12     printf("x: 7\n");
13     printf("x: 8\n");
14     printf("x: 9\n");
15
16     return 0;
17 }
```

Imagine repetir essa estrutura para números maiores, como até 10000. Seria um processo extremamente tedioso e ineficiente ter que escrever tudo isso manualmente, daí a utilidade das estruturas como `while`.

For

Outra estrutura de loop semelhante, mas um pouco mais complexa, é o `for` (conhecido também como for loop). Esta estrutura segue o seguinte padrão:

```
1  for (inicialização; condição; expressão) {
```

```
2         // Código a ser executado
3     }
```

Dentro de uma única estrutura, há três seções com características únicas:

- **Inicialização**: uma expressão que é executada uma única vez antes do looping começar.
- **Condição (de parada)**: verificada antes de cada execução do looping.
- **Expressão**: geralmente é um incremento, mas pode ser qualquer tipo de expressão que precise ser executada ao fim do looping.

Para ilustrar, vamos usar essa estrutura para replicar o exemplo do `while`:

```
1     #include <stdio.h>
2
3     int main(void)
4     {
5         for (int x = 0; x != 10; ++x) {
6             printf("x: \%d\n", x);
7         }
8
9         return 0;
10    }
```

Para entender melhor o código acima, vamos explicar cada parâmetro individualmente:

- `int x = 0`: Inicializa a variável x com o valor 0.
- `x != 10`: Expressão avaliada no início de cada looping.
- `++x`: Incrementa o valor de x ao final de cada looping.

Ao executar o código, o resultado será idêntico ao exemplo utilizando `while`.

É importante notar que, em um `for` loop, é possível omitir qualquer um desses parâmetros, embora seja contra-intuitivo. Por exemplo, podemos criar um loop infinito da seguinte forma:

```
1     #include <stdio.h>
2
3     int main(void)
4     {
5         for (;;) {
6             // Esse looping deve continuar sendo executado
                 indefinidamente
7         }
8
```

```
9           while (1) {
10              // O while requer uma expressão constante
                   verdadeira para
11              // funcionar como um loop infinito
12          }
13      }
```

Sabendo que é possível emitir parâmetros de um `for`, é possível também, imitar o funcionamento de um `while` loop usando um `for` loop da seguinte forma:

```
1       #include <stdio.h>
2
3       int main() {
4           int i = 0;
5           for (; i < 5; ) { // inicialização vazia, condição
                igual ao while, expressão vazia
6               printf("%d\n", i);
7               i++; // incremento feito no corpo
8           }
9           return 0;
10      }
```

Fazendo um `for` com a inicialização e a expressão vazia, podemos imitar perfeitamente o funcionamento de um `while`.

do while

Por fim, temos o menos utilizado loop da linguagem C, chamado `do while`. A diferença fundamental deste loop é que ele executa o bloco de código antes de verificar a expressão. A estrutura deste loop é a seguinte:

```
1       do {
2           // Código a ser executado
3       } while (expressão);
```

Vamos demonstrar o funcionamento desse looping com um exemplo simples:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       do {
6           print("Esse texto será impresso\n");
7       } while (0); // Verificação é feita após a execução
8
9       while (0) { // Verificação é feita antes da execução
10          print("Esse texto não será impresso\n");
11      }
12
13      return 0;
14  }
```

Caso executemos o código acima, veremos o seguinte resultado:

```
Esse texto será impresso
```

Nota-se que o texto do looping `while`, ao contrário do texto do looping `do while`, não foi impresso. Isso ocorre porque o `do while` primeiro executa o código dentro do bloco e, em seguida, verifica a expressão. Como ambas as expressões no código acima são falsas, o `do while` executará o código uma vez e sairá do looping, enquanto o `while` nunca executará seu bloco de código.

Novamente, é possível imitar o funcionamento (não tão exato) de um `for` loop usando o `do while`:

```
1   do
2   {
3       //  Código a ser executado
4       printf( "%d ", i);
5       // Expressão
6       i++;
7   }  while (i <= 10); // Checagem de condição
```

O código acima deve executar exatamente como um `for` loop, com a única exceção de que a primeira iteração será executado antes da checagem da condição ser feita.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Escreva um programa que verifica se número inteiro e informe se ele é positivo, negativo ou zero. Utilize as estruturas `if`, `else if` e `else`.

2. Crie um programa que imprima todos os números pares de 1 até 100 utilizando um loop `while`. Em seguida, refaça o mesmo programa usando um loop `for`.

3. Escreva um código que leia um caractere. Se o caractere for uma vogal (a, e, i, o, u, maiúscula ou minúscula), exiba uma mensagem informando que é vogal. Caso contrário, informe que não é vogal. Utilize `if` e `else`.

4. Crie um programa que exiba a tabuada de um número (de 1 a 10). Utilize um loop `for` para gerar e mostrar cada linha da tabuada no formato: `5 x 1 = 5`.

4.3 Fluxo de dados

4.3.1 Entrada e saída de dados

Quando discutimos entrada e saída de dados em C, frequentemente nos referimos a essa interação por meio de “arquivos”. Embora esses “arquivos” não sejam, de fato, arquivos dentro do seu computador, essa é uma abstração útil para compreender o processo.

Na prática, um ”arquivo”é simplesmente um espaço na memória usado pelo sistema para armazenar informações de forma organizada. Uma analogia útil é imaginar que cada vez que o computador recebe uma informação, ela é escrita em um arquivo para armazenamento, e cada vez que uma informação precisa ser exibida, ela é escrita em um arquivo padrão que o terminal utiliza para determinar o que deve ser exibido na tela.

Saída

Além disso, é importante mencionar que os nomes oficiais, por assim a se dizer, utilizados para se referir a esses arquivos são: `stdin (entrada), stdout (saída)`.

Porém, um detalhe que vale ser mencionado, e que poucas pessoas se perguntam sobre é, existe algum outro tipo de arquivo que não sejam os de entrada e saída? E a resposta é que sim, esse arquivo seria o `stderr`. Que é unicamente designado para exibição de mensagens de erro e é completamente independente do `stdout`, permitindo assim, que haja uma distinção entre todo o conteúdo padrão que está sendo exibido no terminal e mensagens de erro.

Para demonstrar a utilização desses dois tipos de saída de dados diferentes vamos usar o seguinte exemplo:

```c
#include <stdio.h>

void print_error(void)
{
    // Escreve a mensagem no arquivo ‘stdout’
    fprintf(stdout, "Essa é uma mensagem normal.");
}

void print_message(void)
{
    // Escreve a mensagem no arquivo ‘stderr’
    fprintf(stderr, "Essa é uma mensagem de erro.");
}

int main(void)
{
    print_message();
    print_error();
    return 0;
}
```

O código acima utiliza da função `fprintf` para direcionar o output de cada uma das mensagens de maneira mais específica. Diferente do `printf`, nessa outra função você deve escolher entre o `stdout e o stderr` antes de escolher qual mensagem você quer que seja exibida.

Ao compilar e executar o código você deve conseguir ver as duas mensagens sendo exibidas na tela da mesma forma, porém, as duas estão sendo exibidas de locais diferentes. Em sistemas Linux e Windows é possível filtrar o fluxo dessas mensagens da seguinte forma:

```
# Envia o conteúdo de stdout e stderr para 'output.txt'
./main > output.txt
# Envia somente as mensagens de stderr para 'error.txt'
./main 2> error.txt
# Envia somente as mensagens de stderr para 'error.txt'
./main > output.txt 2> error.txt
```

Entrada

Em C, a entrada de dados geralmente é realizada através do arquivo padrão de entrada, conhecido como `stdin`. Este arquivo é associado ao teclado, permitindo que o programa receba dados inseridos pelo usuário durante a execução.

Para ler dados do `stdin`, podemos usar funções de entrada como `scanf` e `fgets`. A

função `scanf` é usada para ler entrada formatada, enquanto `fgets` é usada para ler uma linha de entrada como um texto.

Aqui está um exemplo de uso do `scanf` para ler um número inteiro:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int num;
6       printf("Digite um número inteiro: ");
7       scanf("\%d", &num);
8       printf("Você digitou: \%d\n", num);
9       return 0;
10  }
```

No exemplo acima, o programa solicita ao usuário que digite um número inteiro. O valor inserido pelo usuário é lido pelo `scanf` e armazenado na variável `num`, o qual é impressa na tela.

Se o programa precisar ler uma linha inteira de texto, pode-se usar a função `fgets`, como mostrado no exemplo a seguir:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       char linha[100];
6       printf("Digite uma frase: ");
7       fgets(linha, 100, stdin);
8       printf("Você digitou: %s\n", linha);
9       return 0;
10  }
```

Neste exemplo, o programa solicita ao usuário que digite uma frase. A função `fgets` lê a linha de entrada e armazena na variável `linha`, a qual é então impressa na tela.

Esses são exemplos simples de como usar o `stdin` para entrada de dados em um programa em C. É importante lembrar de validar e manipular os dados de entrada adequadamente para garantir o bom funcionamento do programa.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Escreva um programa que leia dois números inteiros do usuário usando `scanf` e exiba a soma deles no `stdout`. Em seguida, modifique o programa para que, caso o usuário

digite um número negativo, uma mensagem de erro seja enviada para o `stderr` em vez de interromper o programa.

2. Crie um programa que contenha duas funções que escreve uma mensagem no `stdout` usando `fprintf` e uma que escreva a mensagem no `stderr`. No `main`, chame ambas as funções com mensagens diferentes. Compile e execute o programa redirecionando apenas o `stderr` para um arquivo chamado `erros.txt`. Após isso, analise o que você observa no terminal.

3. Faça um programa que leia uma linha de texto completa (incluindo espaços) usando `fgets` e, em seguida, exiba o texto lido caractere por caractere (um por linha) utilizando um loop.

4.3.2 Manipulação de Arquivos

A manipulação de arquivos em C envolve operações como criação, abertura, leitura, escrita e fechamento de arquivos. Como discutido anteriormente na seção 4.3.1, o terminal também se enquadra na definição de arquivos, pois pode receber operações de escrita, leitura, entre outras.

Em aplicações mais complexas, pode ser necessário armazenar algumas informações fornecidas pelo usuário para reutilização pelo programa na próxima execução. Por exemplo, seria impraticável e inconveniente ter que criar uma nova conta ou reconfigurar todas as opções toda vez que um jogo é iniciado.

O uso da manipulação de arquivos é crucial devido às seguintes características:

- **Reusabilidade:** As informações armazenadas em um arquivo podem ser acessadas, atualizadas e removidas conforme necessário.

- **Portabilidade:** Como as informações são geralmente escritas seguindo algum padrão lógico, um arquivo de configurações pode ser facilmente transferido para outro computador sem complicações, reduzindo erros durante a transferência de dados.

- **Eficiência:** Em muitas situações, é mais eficiente gerenciar um grande volume de informações do usuário com apenas algumas instruções, economizando tempo e reduzindo erros.

- **Capacidade de Armazenamento:** Armazenar informações em arquivo elimina a necessidade de manter todas as informações de uma aplicação na memória RAM simultaneamente, evitando preocupações e erros relacionados à sobrecarga de memória.

4.3.3 Tipos de arquivos

Arquivos de texto

Esses arquivos contêm dados em formato de caracteres ASCII, podendo ser lidos e editados por qualquer editor de texto. Geralmente, mas não obrigatoriamente, esses arquivos possuem uma extensão '.txt' e todas as suas linhas terminam com o caractere '\n'.

É comum encontrar esses tipos de arquivos sendo usados para guardar, por exemplo:

- **Registro de logs:** Os arquivos de texto são comumente usados para registrar eventos e informações relevantes durante a execução de um programa. Por exemplo, um servidor web pode registrar todas as solicitações de página em um arquivo de log para fins de monitoramento e análise.
- **Listas de contatos:** Arquivos de texto também podem ser usados para armazenar dados estruturados simples, como listas de contatos, configurações de aplicativos, ou até mesmo informações de configuração de rede.
- **Arquivos de código fonte:** Embora não seja exclusivo de arquivos de texto, os arquivos de código-fonte em linguagens de programação são geralmente armazenados em formato de texto para facilidade de leitura e manipulação. Eles são editáveis por qualquer editor de texto e podem ser versionados facilmente por sistemas de controle de versão como Git.

Arquivos binários

Diferentemente dos arquivos de texto, que têm o seu conteúdo em formato de caracteres ASCII, os arquivos binários têm o seu conteúdo em um formato numérico, que fora de contexto não podem ser interpretados por um ser humano, mas que, quando corretamente interpretados por um programa, podem ser lidos e transformados em informação útil. Por conta disso, os arquivos binários tendem a ter uma maior segurança, pois mesmo que caiam em mãos de usuários maliciosos, não podem ser facilmente interpretados e manipulados de maneira mal intencionada.

Geralmente, esse tipo de arquivo é usado nas seguintes aplicações:

- **Bancos de dados locais:** Muitos sistemas de gerenciamento de banco de dados usam arquivos binários para armazenar dados localmente. Embora os bancos de dados sejam mais comumente associados a servidores e sistemas de gerenciamento de dados complexos, eles também podem ser usados para aplicações simples de desktop.
- **Guardar imagens e multimídias:** Arquivos binários são amplamente utilizados para armazenar imagens, vídeos, áudio e outros tipos de multimídia. Isso ocorre porque esses tipos de dados são melhor representados em formato binário, o que permite uma manipulação eficiente e compactação.

- **Arquivos executáveis:** Os arquivos binários também incluem executáveis de programas. Quando você compila um programa em linguagem de máquina, o resultado é um arquivo binário que contém as instruções que o computador pode executar diretamente.

4.3.4 fopen, fread, fwrite e fclose

fopen

A função `fopen` é usada para abrir um arquivo existente ou criar um novo arquivo. Ela retorna um ponteiro para o arquivo que pode ser usado em outras operações de entrada e saída de arquivo. A assinatura da função é a seguinte:

```
1    FILE *fopen(const char *filename, const char *mode);
```

O primeiro argumento `filename` é um vetor que contém o caminho do arquivo a ser aberto ou criado. O segundo argumento `mode` especifica o modo de abertura do arquivo, como leitura ("r"), escrita ("w"), anexação ("a"), etc.

fread

A função `fread` é usada para ler blocos de dados de um arquivo. Ela recebe quatro argumentos: um ponteiro para o vetor onde os dados lidos serão armazenados, o tamanho em bytes de cada elemento a ser lido, o número de elementos a serem lidos e um ponteiro para o arquivo a partir do qual os dados serão lidos. A assinatura da função é a seguinte:

```
1    size_t fread(void *ptr, size_t size, size_t nmemb, FILE *
       stream);
```

O retorno da função é o número total de elementos lidos com sucesso. Valor esse que pode ser utilizado para identificar quando um arquivo foi lido por completo, já que quando a função alcança o final do arquivo, 0 bytes são lidos do mesmo.

fwrite

A função `fwrite` é usada para escrever blocos de dados em um arquivo. Ela recebe quatro argumentos: um ponteiro para um vetor de dados a ser escrito, o tamanho em bytes de cada elemento a ser escrito, o número de elementos a serem escritos e um ponteiro para o arquivo no qual os dados serão escritos. A assinatura da função é a seguinte:

```
1    size_t fwrite(const void *ptr, size_t size, size_t nmemb,
       FILE *stream);
```

O retorno da função é o número total de elementos gravados com sucesso.

fclose

A função `fclose` é usada para fechar um arquivo previamente aberto. Ela recebe um ponteiro para o arquivo a ser fechado como seu único argumento. A assinatura da função é a seguinte:

```
1    int fclose(FILE *stream);
```

Se a operação for bem-sucedida, `fclose` retorna zero. Se ocorrer um erro, retorna um valor diferente de zero.

Essas funções são essenciais para manipular arquivos em C e são amplamente utilizadas em programas que necessitam de entrada e saída de dados persistentes.

Exemplo prático

Após entender o que cada uma das funções acima faz, vamos tentar exemplificar um uso prático das mesmas:

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <ctype.h>
4
5    int main() {
6        // Abre o arquivo para leitura
7        FILE *arquivo_entrada = fopen("dados.txt", "r");
8        if (arquivo_entrada == NULL) {
9            perror("Erro ao abrir arquivo de entrada");
10           return EXIT_FAILURE;
11       }
12
13       // Abre um novo arquivo para escrita
14       FILE *arquivo_saida = fopen("dados_modificados.txt", "w
             ");
15       if (arquivo_saida == NULL) {
16           perror("Erro ao criar arquivo de saída");
17           fclose(arquivo_entrada);
18           return EXIT_FAILURE;
19       }
20
21       // Lê o conteúdo do arquivo de entrada
22       char buffer[100];
23       size_t bytes_lidos = fread(buffer, sizeof(char), 100,
             arquivo_entrada);
24       if (bytes_lidos == 0) {
25           perror("Erro ao ler arquivo de entrada");
26           fclose(arquivo_entrada);
```

```
27              fclose(arquivo_saida);
28              return EXIT_FAILURE;
29          }
30
31          // Modifica os dados lidos (exemplo: converter para mai
               úsculas)
32          for (size_t i = 0; i < bytes_lidos; i++) {
33              buffer[i] = toupper(buffer[i]);
34          }
35
36          // Escreve os dados modificados no arquivo de saída
37          size_t bytes_escritos = fwrite(buffer, sizeof(char),
               bytes_lidos, arquivo_saida);
38          if (bytes_escritos != bytes_lidos) {
39              perror("Erro ao escrever no arquivo de saída");
40              fclose(arquivo_entrada);
41              fclose(arquivo_saida);
42              return EXIT_FAILURE;
43          }
44
45          // Fecha os arquivos abertos
46          if (fclose(arquivo_entrada) == EOF) {
47              perror("Erro ao fechar arquivo de entrada");
48              return EXIT_FAILURE;
49          }
50          if (fclose(arquivo_saida) == EOF) {
51              perror("Erro ao fechar arquivo de saída");
52              return EXIT_FAILURE;
53          }
54
55          printf("Arquivo modificado com sucesso!\n");
56
57          return EXIT_SUCCESS;
58      }
```

Neste exemplo:

- Abrimos um arquivo chamado `dados.txt` para leitura (”r”) e outro chamado `dados_modificados.txt` para escrita (”w”).
- Usamos `fread` para ler até 100 bytes do arquivo de entrada.
- Modificamos os dados lidos (nesse caso, convertemos todas as letras para maiúsculas).
- Usamos `fwrite` para escrever os dados modificados no arquivo de saída.

- Finalmente, fechamos ambos os arquivos com `fclose`.

> "Dê um peixe a um homem e ele se alimentará por um dia. Ensine-o a pescar e ele se alimentará pelo resto da vida."
>
> Seria injusto que em um livro eu apenas ensinasse como usar as funções `fopen`, `fread`, `fwrite` e `fclose`, sem considerar a possibilidade de que novos padrões possam surgir no futuro, ou que você precise atualizar seus conhecimentos. Por isso, vou compartilhar um pequeno "truque"muito utilizado entre programadores em linguagem C, que pode ser útil para relembrar ou aprender como uma função funciona.
>
> Para isso, vamos usar o comando Linux `man` (abreviação de manual), cujo nome por si só é autoexplicativo. Ele é usado para pesquisar o manual de utilização de alguma função ou comando. Felizmente, também é possível pesquisar por funções da biblioteca padrão da linguagem C, conhecida como `libc`. Para aqueles que não usam sistemas Linux, é possível usar o comando `man` para pesquisar sobre funções até mesmo pelo Google.
>
> Basta usar o comando:
>
> ```
> man nome_da_funcao_ou_comando
> ```
>
> Com isso, você será capaz de encontrar o manual de qualquer função padrão da biblioteca `libc`, e até mesmo de algumas funções que estão fora das bibliotecas padrão.

4.3.5 fseek, ftell e rewind

Essa seção aborda funções específicas para manipulação do ponteiro de posição em arquivos, permitindo navegar por diferentes partes de um arquivo sem necessariamente ler todo o seu conteúdo sequencialmente.

fseek

A função `fseek()` possui o seguinte protótipo:

```
1 int fseek(FILE *stream, long offset, int whence);
```

A função `fseek` move o ponteiro de leitura/escrita do arquivo para uma posição específica. Segue abaixo uma lista explicando os parâmetros da função:

- `stream`: Ponteiro para o arquivo (obtido com `fopen()`)

- `offset`: Número de bytes de deslocamento (pode ser positivo ou negativo)
- `whence`: Ponto de referência para o deslocamento que aceita os valores `SEEK_SET`, `SEEK_CUR` ou `SEEK_END`.

Sendo `SEEK_SET` o início do arquivo, `SEEK_CUR` a posição atual do arquivo e `SEEK_END` o final do arquivo. Além disso, a função retorna 0 em caso de sucesso e um valor diferente de 0 em caso de erro.

Exemplos Práticos

```c
#include <stdio.h>

int main() {
    FILE *fp;

    // Abre o arquivo para leitura
    fp = fopen("dados.txt", "r");

    if (fp == NULL) {
        printf("Erro ao abrir o arquivo!\n");
        return 1;
    }

    // 1. Ir para o final do arquivo
    fseek(fp, 0, SEEK_END);

    // 2. Retornar ao início do arquivo
    fseek(fp, 0, SEEK_SET);

    // 3. Avançar 10 bytes a partir da posição atual
    fseek(fp, 10, SEEK_CUR);

    // 4. Retroceder 5 bytes a partir da posição atual
    fseek(fp, -5, SEEK_CUR);

    // 5. Posicionar 50 bytes antes do final do arquivo
    fseek(fp, -50, SEEK_END);

    fclose(fp);
    return 0;
}
```

ftell

A função `ftell()` possui o seguinte protótipo:

```
long ftell(FILE *stream);
```

A função funciona de forma bem simples, simplesmente retornando a posição atual do ponteiro no arquivo relativa ao seu início (em bytes). Sendo assim, extremamente conveniente para calcular o tamanho de um arquivo, como é mostrado no código abaixo:

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("arquivo.txt", "r");
    long tamanho, pos_atual;

    if (fp == NULL) {
        printf("Erro ao abrir arquivo!\n");
        return 1;
    }

    // Vai para o final
    fseek(fp, 0, SEEK_END);

    // Obtém o tamanho do arquivo
    tamanho = ftell(fp);
    printf("Tamanho do arquivo: %ld bytes\n", tamanho);

    // Retorna ao início
    fseek(fp, 0, SEEK_SET);

    fclose(fp);
    return 0;
}
```

rewind

A função `rewind()` possui o seguinte protótipo:

```
void rewind(FILE *stream);
```

Essa função reposiciona o ponteiro para o início do arquivo, sendo assim equivalente a:

```
fseek(fp, 0, SEEK_SET);
```

Com um único porém que o indicador de erro é limpo. O que pode também ser feito da seguinte forma:

```
fseek(fp, 0, SEEK_SET);
clearerr(fp);
```

Segue abaixo um exemplo de como utilizar a função `rewind`:

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("exemplo.txt", "r+");
    char buffer[100];

    if (fp == NULL) {
        printf("Erro ao abrir arquivo!\n");
        return 1;
    }

    // Lê uma linha do arquivo
    fgets(buffer, 100, fp);
    printf("Primeira leitura: %s", buffer);

    // Retorna ao início para ler novamente
    rewind(fp);

    // Lê a mesma linha novamente
    fgets(buffer, 100, fp);
    printf("Leitura apos rewind: %s", buffer);

    fclose(fp);
    return 0;
}
```

O indicador de erro de um arquivo é um **sinalizador interno** (flag) associado ao ponteiro de arquivo `FILE` que indica se a última operação de entrada/saída teve sucesso ou falhou. Quando uma função como `fread()`, `fwrite()`, `fscanf()` falha, esse indicador é ativado.

Portanto, ao limpar o indicador de erro, é possível tentar fazer novamente a parte do processo que falhou tendo o benefício de poder usar o indicador de erro novamente.

Considerações Importantes

- **Arquivos binários vs texto:** Em arquivos binários, `fseek()` com `SEEK_END` funciona normalmente. Em arquivos de texto, alguns sistemas podem não suportar deslocamentos negativos.
- **Portabilidade:** Para arquivos muito grandes (mais de 2GB), use `fseeko()` e `ftello()` que utilizam `off_t` em vez de `long`.
- **Verificação de erros:** Sempre verifique o retorno de `fseek()` em aplicações críticas:

```
1     if (fseek(fp, offset, SEEK_SET) != 0) {
2         printf("Erro ao reposicionar ponteiro!\n");
3     }
```

Capítulo 5

Estrutura de dados e memória

Após assimilar os fundamentos básicos da sintaxe da linguagem C, você está preparado para criar e compreender aplicações simples. No entanto, há um aspecto crítico que ainda não foi abordado neste livro: a gestão de memória pelo programa. Embora possa parecer um tema subjetivo, trata-se de um componente técnico essencial que requer consideração cuidadosa na elaboração de qualquer software.

Dominar os conceitos relacionados à gestão de memória permitirá não apenas o desenvolvimento de aplicações mais eficientes e econômicas em termos de recursos, mas também uma compreensão mais profunda do funcionamento do computador como um todo.

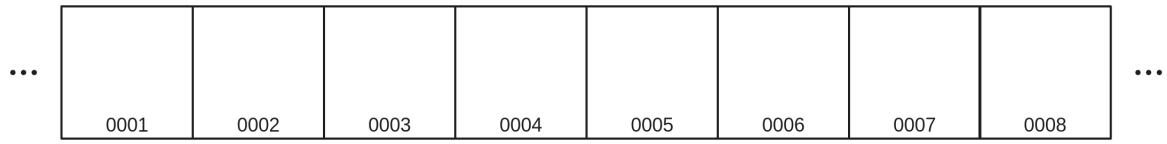
Portanto, é crucial que os leitores dediquem atenção especial a este capítulo e, se necessário, revisitem partes que possam parecer complexas. Qualquer lacuna de entendimento teórico neste assunto pode resultar em horas perdidas devido à falta de conhecimento detalhado (Falo por experiência própria).

5.1 Ponteiros

5.1.1 O que são endereços

Entendendo endereços

A memória RAM[1], local onde as variáveis são guardadas e manipuladas, é essencialmente uma sequência linear de bytes, organizada como uma fila, com a capacidade de armazenar informações. Essa estrutura pode ser visualizada como uma série de "caixas" sequenciais, cada uma capaz de reter um valor específico, conforme demonstrado na figura a seguir:



O conceito de endereço refere-se à posição específica de uma variável x dentro desta sequência linear. Esses endereços funcionam como números de identificação únicos para a localização de variáveis dentro do espaço de memória do programa. Na figura apresentada, os endereços são indicados pelos números na parte inferior de cada "caixa".

Cada variável vai ocupar uma quantidade diferente de bytes, a depender de seu tipo. Por exemplo, variáveis do tipo `char` ocupam 1 byte, variáveis do tipo `int` ocupam 4 bytes e uma variável do tipo `double`, 8 bytes. O tamanho relativo de cada tipo pode ser encontrado na tabela do subcapítulo 4.1.2.

Para acessar o endereço de memória de uma variável, utilizamos o operador &, conforme exemplificado no código a seguir:

[1]Random Access Memory

```c
#include <stdio.h>

int main(void)
{
    int x = 0;
    printf("Endereço de x: %p\n", &x);

    return 0;
}
```

O código inicializa uma variável x com o valor 0 e, em seguida, emprega a função `printf`, utilizando o especificador de formato `%p` para indicar a impressão de um endereço. Dentro da função `printf`, o argumento `&x` é utilizado para obter e imprimir o endereço da variável x.

Ao executar o código, espera-se um resultado semelhante ao seguinte:

```
Endereço de x: 0x7ffd3a450b3c
```

Bases numéricas

O valor impresso representa o endereço da variável x no espaço de memória do programa. Nota-se que o endereço é exibido em formato hexadecimal, que difere dos números decimais convencionais. A escolha do formato hexadecimal (base 16) sobre o decimal é consequência da expansão da capacidade de memória ao longo do tempo, tornando os endereços de memória progressivamente maiores e mais complexos para representação em base decimal. A base hexadecimal, ao permitir uma representação mais compacta de números grandes, tornou-se a convenção adotada.

Além do sistema hexadecimal, existem outros sistemas numéricos como o octal (base 8) e o binário (base 2) que também podem vir a ser úteis de se conhecer. A seguir, apresentamos o número 42.000 representado nessas diferentes bases:

```
Hexadecimal: 0xA410
Octal: 0122020
Binário: 1010010000010000
```

Observa-se que com o aumento da base numérica, os números representados tornam-se mais compactos, enquanto em bases menores, os números expandem. Além disso, é importante destacar as convenções para declarar números em diferentes bases em C:

- Números hexadecimais devem iniciar com `0x`

- Números Octais devem iniciar com `0`
- Números binários devem começar com `0b`

Abaixo, um exemplo prático de como declarar números em diferentes bases na linguagem C e imprimir seus valores usando formatações distintas:

```c
#include <stdio.h>

int main(void)
{
    int hexa = 0xa410;
    int octal = 0122020; // Não confundir com 122020
    int bin = 0b1010010000010000;

    // %x -> Imprime número em formato hexadecimal
    // %o -> Imprime número em formato octal
    printf("Valor de hexa: %x (%d)\n", hexa, hexa);
    printf("Valor de octal: %o (%d)\n", octal, octal);
    printf("Valor de bin: %d\n", bin);

    return 0;
}
```

Infelizmente, a linguagem C padrão não suporta a formatação direta de números binários com `%b`. Esta funcionalidade pode requerer implementações específicas ou bibliotecas adicionais. Ademais, executando o código acima você deve ter o seguinte resultado:

```
Valor de hexa: 0xa410 (42000)
Valor de octal: 122020 (42000)
Valor de bin: 42000
```

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Escreva um programa que declare três variáveis de tipos diferentes: um `char`, um `int` e um `double`. Imprima o endereço de cada uma delas usando o operador `&` e o especificador `%p`. Observe e anote os valores dos endereços. Eles são sequenciais? Qual a diferença entre os endereços do `char` e do `int`? Tente explicar o que você entende que ocorreu pelos valores observados.

2. Considerando que um `short int` ocupa 2 bytes em muitos sistemas, e um `int` ocupa 4 bytes, escreva um programa que declare uma variável do tipo `short` e uma do tipo `int`. Imprima os endereços de ambas. A diferença entre os endereços é sempre igual ao tamanho do tipo? Por quê?

3. Tente explicar por que os endereços de memória são geralmente exibidos em hexadecimal em vez de decimal. Dê um exemplo de um número grande (como $2^{32} - 1$) representado em decimal e em hexadecimal, mostrando quantos dígitos cada representação possui.

4. Escreva um programa que declare uma variável inteira com o valor `0b101010` (binário), outra com o valor `052` (octal) e uma terceira com o valor `0x2A` (hexadecimal). Imprima todas elas em formato decimal. Os três valores são iguais? Por quê?

5. Faça um programa que peça ao usuário um número decimal (usando `scanf`), e em seguida imprima esse número em formato hexadecimal (com `%x`) e octal (com `%o`). Teste com os valores 0, 100, 1000 e 65535. O que acontece se você tentar imprimir um número negativo nesses formatos?

5.1.2 O que são ponteiros

Entendendo ponteiros

Após compreender o conceito de endereços na linguagem C, o entendimento e a aplicação de ponteiros se tornam uma questão de prática. Ponteiros são variáveis especiais que armazenam o endereço de memória de outras variáveis de um tipo definido.

Para esclarecer, vejamos um exemplo prático de como declarar e utilizar ponteiros:

```c
#include <stdio.h>

int main(void)
{
    int x = 10;
    int *ptr = &x;
    printf("Valor de ptr: %d (%p)\n", *ptr, ptr);

    return 0;
}
```

Neste exemplo, inicializamos a variável x com o valor 10. Em seguida, declaramos uma variável `ptr` do tipo `int *`, indicando que `ptr` é um ponteiro para um inteiro. O símbolo `*`, posicionado entre o tipo e o nome da variável, define `ptr` como um ponteiro. Atribuímos a `ptr` o endereço de x, utilizando o operador `&`.

Para acessar o valor armazenado no endereço apontado por `ptr`, usamos `*ptr`, onde

o operador `*` é utilizado para deferenciar[2] o ponteiro. A função `printf` é então utilizada para imprimir tanto o valor de x acessado por meio de `ptr` quanto o endereço de memória armazenado em `ptr`.

Acessando variáveis fora de escopo

Acessar variáveis fora do escopo atual de uma função pode ser útil em muitas situações, e os ponteiros facilitam esse processo. Vamos explorar um exemplo prático para ilustrar como isso pode ser feito:

```
1   #include <stdio.h>
2
3   void foo(int x)
4   {
5       x = 20;
6       printf("foo: valor de x: %d\n", x);
7   }
8
9   int main(void)
10  {
11      int x = 10;
12      printf("main: valor de x: %d\n", x);
13      foo(x);
14      printf("main: valor de x: %d\n", x);
15
16      return 0;
17  }
```

No código acima, declaramos uma variável x com o valor 10 na função main, imprimimos o valor de x e, em seguida, chamamos a função `foo` passando x como argumento. Dentro de `foo`, definimos o valor de x como 20 e imprimimos seu valor. No entanto, vale observar que a alteração feita dentro de `foo` não afeta o valor de x na função main, pois `foo` recebe uma cópia do valor de x e qualquer alteração é feita apenas nessa cópia local.

Portanto, ao executar o programa acima é possível notar que o valor de x continua o mesmo na função `main` mesmo após a execução de `foo`:

```
main: valor de x: 10
foo: valor de x: 20
main: valor de x: 10
```

Para alterar efetivamente o valor de x dentro de foo, podemos utilizar ponteiros para acessar o endereço de memória da variável x original. Veja como isso pode ser feito:

[2]Processo de acessar o valor contido dentro de um endereço de memória

```c
#include <stdio.h>

void foo(int *x)
{
    *x = 20;
    printf("foo: valor de x: %d\n", *x);
}

int main(void)
{
    int x = 10;
    int *ptr = &x;

    printf("main: valor de x: %d\n", x);
    foo(ptr);
    printf("main: valor de x: %d\n", x);

    return 0;
}
```

Neste código, declaramos um ponteiro `ptr` que guarda o endereço de memória de x. Em seguida, passamos esse ponteiro para a função `foo`. Dentro de `foo`, utilizamos o operador * para acessar o valor apontado pelo ponteiro x e alteramos seu valor para 20. Dessa forma, ao executar o programa, o valor de x também é alterado na função `main`, resultando no seguinte:

```
main: valor de x: 10
foo: valor de x: 20
main: valor de x: 20
```

Esse exemplo demonstra como os ponteiros permitem acessar e modificar variáveis fora do escopo atual de uma função, facilitando a manipulação de dados e a comunicação entre diferentes partes de um programa.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Escreva um programa que declare uma variável inteira `valor` e um ponteiro `ptr` que aponte para `valor`. Atribua um valor a `valor` usando o ponteiro (ou seja, através de `*ptr`). Imprima o conteúdo de `valor` e o endereço armazenado em `ptr` para verificar que ambos se referem ao mesmo local.

2. Crie uma função `incrementa(int *n)` que receba um ponteiro para inteiro e incremente o valor apontado em 1. Na função `main`, declare um inteiro `a` com valor 5, chame `incrementa(&a)` e imprima `a` antes e depois. Explique por que a mudança persiste fora da função.

3. Escreva uma função `max_min(int *a, int *b)` que receba dois ponteiros para inteiros e modifique o conteúdo de forma que, ao final, `*a` contenha o maior valor e `*b` o menor valor. Na `main`, leia dois números, chame a função e imprima os valores modificados.

4. O que acontece se você tentar desreferenciar um ponteiro que não foi inicializado (ponteiro selvagem)? Escreva um pequeno programa que declare um ponteiro `int *p` e tente imprimir `*p` sem atribuir nenhum endereço a ele. Compile e execute (se possível). Explique o comportamento observado e por que isso é perigoso.

5.1.3 Vetores

Entendendo vetores

Ao abordar o conceito de vetores, entramos em um dos fundamentos da organização de dados em programação, especialmente na linguagem C. Vetores permitem agrupar dados do mesmo tipo em uma única estrutura acessível através de ponteiros. Isso simplifica a manipulação de coleções de dados como, por exemplo, um conjunto de idades tal qual no exemplo proposto. Vamos ver como isso pode ser implementado:

```c
#include <stdio.h>

int main(void)
{
    // Declara um vetor de inteiros com espaço para 5
       idades.
    int idades[5] = {23, 45, 56, 19, 30};
    int soma = 0;
    float media;

    // Calcula a soma das idades
    for (int i = 0; i < 5; i++) {
        soma += idades[i];
    }

    // Calcula a média das idades
    media = (float)soma / 5;

    printf("A média das idades é: %.2f\n", media);

    return 0;
}
```

Neste exemplo, declaramos um vetor `idades` capaz de armazenar 5 valores inteiros, correspondentes às idades de pessoas aleatórias. Em seguida, utilizamos um `for loop` para somar todas as idades armazenadas no vetor. Após calcular a soma, dividimos esse valor pelo número total de idades para encontrar a média, que é armazenada na variável `media` e, por fim, exibida na tela.

Algumas nomenclaturas que podem facilitar o entendimento acerca dos vetores são as seguintes:

- **Tamanho:** Refere-se à quantidade de espaço total alocada para o vetor na memória, o tamanho de um vetor estático é definido durante a sua declaração e **não pode ser alterado durante a execução do programa.** Geralmente o tamanho do vetor se encontra entre os colchetes durante a declaração do mesmo ou é definido de maneira implícita pelo próprio compilador.
- **Comprimento ou quantidade de itens:** Refere-se à quantidade de elementos atualmente armazenados no vetor, esse valor pode ser menor que o tamanho total do vetor, mas nunca maior.
- **Índice:** Refere-se a posição de um item específico dentro do vetor. Os índices são valores que podem ser utilizados como um número de identificação para um item específico em um vetor de tamanho N e podem ir de 0 até N - 1.

Um detalhe importante a se saber sobre o código acima, é que quando declaramos um vetor para guardar a idade de 5 pessoas, na verdade estamos alocando o espaço necessário para guardar 5 números, em sequência, na memória do computador. Ou seja, ao declarar:

```
1         int idades[5] = {23, 45, 56, 19, 30};
```

Estamos criando na memória do computador uma sequência de dados semelhante a seguinte:

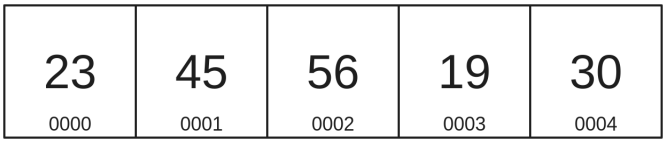


Figura 5.1: Os valores do endereço da imagem acima são apenas ilustrativos.

Ao considerarmos a criação de uma sequência de dados na memória do computador, conforme ilustrado anteriormente, compreendemos que um vetor é essencialmente uma série de valores organizados de maneira contínua na memória. A variável do vetor só precisa guardar o endereço do primeiro número na memória, semelhante ao que um ponteiro faria, e usa de um truque interessante para acessar cada um de seus elementos usando o valor do índice. No código acima, especificamente na seguinte linha:

```
1     soma += idades[i];
```

Usamos o valor de i, que vai de 0 a 4 dentro do looping, como um valor de índice para indicar qual item queremos acessar dentro do vetor. Por exemplo, quando i é igual a 0, o computador acessa o endereço do primeiro número contido na variável do vetor e pula X itens a frente na memória baseado no valor passado entre colchetes.

Caso você queira entender como esse processo ocorre, de maneira mais técnica, seria como se o hardware realizasse esse cálculo para retornar o valor relativo ao elemento do vetor:

```
1         soma += *(idades + sizeof(int) * i)
```

Neste contexto, `idades` atua como um ponteiro para o endereço do primeiro elemento do vetor. A operação `sizeof(int) * i` calcula o deslocamento em bytes necessários para alcançar o elemento do vetor relativo ao valor de i, considerando o tamanho de um inteiro (expresso por sizeof(int)). Somando este deslocamento ao endereço base armazenado em idades, obtemos o endereço do elemento i.

E em seguida usamos o operador de deferenciação * para acessar o valor armazenado neste endereço específico, permitindo sua adição a soma.

Utilizando vetores de texto

O uso mais comum de vetores em C é para armazenar textos longos, como nomes e frases. Como a linguagem não possui tipos de variáveis nativos capazes de armazenar textos, nomes ou frases, recorre-se a vetores de variáveis do tipo `char` para representá-los. Vejamos o exemplo abaixo para ilustrar esse conceito:

```c
#include <stdio.h>

int main(void)
{
    // Ao deixar o espaço entre cochetes vazio
    // o compilador calcula automaticamente o tamanho
    // necessário para guardar todos os itens do vetor
    char nome1[] = "Alan";
    char nome2[] = "Vinicius";
    char nome3[] = "Janiel";

    // A linguagem C prevê a impressão de
    // vetores de caracteres com o formatador '%s'
    printf("1º Nome: %s\n", nome1);
    printf("2º Nome: %s\n", nome1);
    printf("3º Nome: %s\n", nome1);

    return 0;
}
```

No exemplo acima, criamos vetores de caracteres para armazenar os nomes de três pessoas, os quais são impressos posteriormente. Apesar de não existir um tipo de dado de texto nativo em C, a linguagem oferece implementações e convenções para a manipulação de vetores de caracteres como textos.

Um detalhe extremamente importante a mencionar é que, em relação ao uso de vetores para armazenar informações de texto, muitos projetos e bibliotecas utilizam um caractere especial para sinalizar o fim do vetor, conhecido como caractere nulo. Carácter esse que caso não esteja presente no vetor, ou até mesmo mal posicionado, pode ocasionar erros de acesso de memória inválida ou perda de informação.

Devido à capacidade de tamanho variável dos vetores de texto, não é possível determinar com certeza o tamanho do vetor quando ele é recebido por uma função, por exemplo. Para evitar erros relacionados ao acesso de memória não autorizada, uma convenção foi estabelecida: adicionar o carácter '\0' ao final de cada vetor de texto, sinalizando o seu término.

```c
char nome1[] = "Alan";
char nome2[] = "Vinicius";
char nome3[] = "Janiel";
```

No caso acima, quando declaramos e inicializamos os vetores de texto com os nomes, o compilador calcula o tamanho necessário para que o vetor possa conter todos os caracteres e mais um byte extra que deve conter o carácter nulo adicional.

Declarando vetores de texto manualmente

Outra forma não tão prática de definir um vetor, mas que ilustra bem a utilização do carácter nulo em um vetor, é a seguinte:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5       char nome[5];
6       nome[0] = 'A';
7       nome[1] = 'l';
8       nome[2] = 'a';
9       nome[3] = 'n';
10      nome[4] = '\0';
11
12      printf("Nome: %s\n", nome);
13
14      return 0;
15  }
```

Aqui, declaramos manualmente um vetor com o tamanho de 5 caracteres e inicializamos cada um dos itens com os caracteres desejados seguidos ao carácter nulo especial no fim do vetor.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Escreva um programa que declare um vetor de inteiros com 10 posições e inicialize-o com os valores de 1 a 10 usando um loop `for`. Em seguida, imprima todos os valores do vetor na ordem inversa (do último para o primeiro).

2. Crie um programa que leia 5 números inteiros do usuário e armazene-os em um vetor. Calcule e exiba a soma de todos os elementos, o maior valor e o menor valor presentes no vetor. Use loops para percorrer o vetor.

3. Explique, com um trecho de código, o que acontece se você tentar acessar um índice fora dos limites do vetor (por exemplo, `vetor[10]` em um vetor de tamanho 5). Compile e execute (se possível). Explique o que aconteceu, e por quê isso é perigoso.

4. Faça um programa que declare dois vetores de inteiros, cada um com 5 elementos. Preencha o primeiro vetor com valores fornecidos pelo usuário e o segundo vetor com o dobro dos valores do primeiro. Em seguida, exiba os dois vetores lado a lado (ex.: `vetor1[0] = 3, vetor2[0] = 6`).

5. Teoricamente, todo vetor em C é armazenado de forma contígua na memória, escreva um programa que declare um vetor de 4 inteiros e imprima o endereço de cada elemento usando `%p`. Verifique se a diferença entre os endereços é sempre `sizeof(int)`. O que isso confirma sobre a organização do vetor?

6. Utilizando aritmética de ponteiros (sem usar colchetes), percorra um vetor de 6 inteiros e atribua a cada posição o valor do seu índice ao quadrado. Exiba os valores usando a notação de vetor (com colchetes) para confirmar. Dica: use um ponteiro auxiliar que aponte para o início do vetor.

5.1.4 Alocação de memória

Usando malloc

Foi abordado no capítulo anterior a criação e manipulação de vetores estático, ou seja, que tinham um tamanho definido durante a sua declaração. No entanto, vamos supor que você tenha uma aplicação que contém um vetor, de qualquer tipo, com capacidade para guardar 12 itens. Porém, em algum momento da execução do programa surge a necessidade de se guardar um 13º item, dado a natureza dos vetores estáticos, seria necessário apagar algum dos itens existentes do vetor para guardar esse novo, caso contrário, seria impossível ter um 13º item dentro do vetor.

Esse problema, porém, pode ser resolvido facilmente com alocação de memória dinâmica. Através disso, é possível criar vetores dinâmicos que possuem um tamanho variável que pode ser modificado durante a execução da aplicação. Permitindo ao usuário uma maior flexibilidade e eficiência para armazenamento de certas informações.

Apesar dos pontos positivos, o uso de memória alocada também tem suas desvantagens. Como a alocação de memória não é feita de maneira automática pelo compilador durante o processo de compilação, e sim durante o processo de execução, é de responsabilidade do usuário cuidar daquele pedaço de memória e liberá-lo novamente para o sistema assim que possível. Processo esse que, caso não seja feito corretamente, pode causar problemas de vazamento de memória[3].

Portanto, é necessário que ao utilizar memória alocada em sua aplicação o usuário lembre-se de liberar a memória após o seu uso. Ademais, vamos tentar exemplificar o processo de alocação de memória:

[3]Problemas de vazamento de memória são a causa de maior parte dos bugs e vulnerabilidades encontrados em softwares modernos, seções de memória alocadas e não liberadas podem ser a causa de mal funcionamento do sistema e até de ataques de agentes externas a sua aplicação/sistema.

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

int main(void)
{
    // Declara um ponteiro para int
    int *numeros;

    int n; // Guarda uma quantidade de números a definir
    printf("Escolha a quantidade de números: ");
    scanf("%d", &n);

    // Reserva memória para 'n' inteiros
    numeros = malloc(n * sizeof(int));

    // Verifica se a memória foi alocada corretamente
    if (numeros == NULL) {
        printf("Alocação de memória falhou.\n");
        return 1;
    }

    printf("Digite %d números:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &numeros[i]); // Recebe os números do
            usuário
    }

    // Calcula a soma
    int soma = 0;
    for (int i = 0; i < n; i++) {
        soma += numeros[i];
    }

    // Calcula a média
    float media = (float)soma / n;

    printf("Média: %.2f\n", media);

    // Libera a memória reservada pelo programa
    free(numeros);
    return 0;
}
```

No código acima é pedido ao usuário para escolher uma quantidade N de números, que deverão ser digitados pelo usuário, de maneira que em seguida o programa calcule a média dos números digitados. Como o usuário pode escolher qualquer quantidade de números, não seria possível usar um vetor de tamanho estático.

(Para explicar o processo de alocação e liberação de memória vamos focar nas seguintes linhas de código)

```
1     numeros = malloc(n * sizeof(int));
2     ...
3     if (numeros == NULL) {
4         printf("Alocação de memória falhou.\n");
5         return 1;
6     }
7     ...
8     free(numeros);
```

Sendo assim, declaramos um ponteiro `numeros` e atribuímos como valor o resultado da função `malloc`, que aceita como parâmetro um número corresponde a quantidade de memória desejada a ser alocada e retorna o endereço do início da seção de memória alocada.

Após usar a função `malloc` para alocar a memória necessária, é necessário verificar se a memória foi alocada corretamente verificando se o ponteiro da seção de memória contém o valor `NULL` ou não. No exemplo acima verificamos se `numeros == NULL` e caso a verificação resulte em `verdadeiro` o programa imprime uma mensagem de erro ao usuário e é encerrado logo em seguida.

Após realizados todos os cálculos desejados usamos a função `free`, que aceita como parâmetro um ponteiro para a seção de memória alocada, para liberar a memória alocada devolta para o sistema.

Malloc, Calloc e Realloc

Aqui estão algumas funções similares a malloc que podem ser úteis:

- **calloc:** Tem o mesmo funcionamento da função `malloc`, porém, ao alocar a memória solicitada a função se certifica de inicializar todo o bloco de memória alocado com o valor 0. Em alguns casos seções de memória alocadas podem conter valores aleatórios que foram deixados por outras por outras aplicações que usaram o bloco anteriormente, então a depender do funcionamento da sua aplicação pode ser necessário que se faça essa "limpeza" em blocos de memória alocados. Segue abaixo um exemplo de como usar a função `calloc`:

```
1     ...
2     char *ponteiro = calloc(numero_de_itens,
          tamanho_de_cada_item);
3     ...
```

- **realloc:** A função realloc é a mais diferente dentre as três, pois tem a função de realocar uma mesma seção de memória porém com um tamanho diferente do anterior, geralmente essa função é utilizada quando se utilizou uma seção de memória ao máximo e é necessário expandi-la em algum outro momento. A função retorna um ponteiro para a nova seção de memória com o tamanho modificado sem alterar o conteúdo pré-existente daquela seção. Segue abaixo um exemplo de como usar a função `realloc`:

```
1    ...
2    // Alocando memória para um vetor hipotético
3    int *ponteiro = malloc(sizeof(int) * 10);
4    // Modificando o tamanho da seção de memória alocada
5    ponteiro = realloc(ponteiro, novo_tamanho_da_seção);
6    ...
```

Caso deseje aprender mais sobre o funcionamento das funções `malloc, realloc ou calloc`, é possível pesquisar utilizando o comando `'man nome_da_funcao'` em um terminal Unix/Linux, ou no seu navegador web, para acessar os manuais públicos relacionados a essas funções.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Escreva um programa que pergunte ao usuário quantos números inteiros ele deseja armazenar, aloque dinamicamente um vetor com esse tamanho usando `malloc`, leia os números do usuário e, em seguida, exiba-os na ordem inversa. Não se esqueça de liberar a memória com `free` e verificar se a alocação foi bem-sucedida.

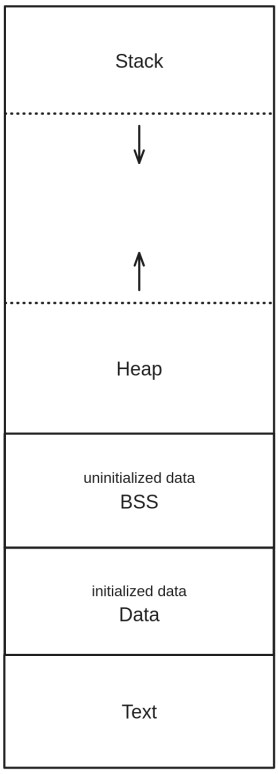
2. Crie um programa que aloque um vetor de `float` com 5 posições usando `calloc`. Imprima os valores logo após a alocação (antes de inicializá-los manualmente). O que você observa? Compare com o comportamento de `malloc` (alocando um vetor similar sem inicialização). Explique a diferença.

3. Faça um programa que simule um vetor dinâmico de inteiros. Inicialmente, aloque espaço para 2 elementos usando `malloc`. Em seguida, peça ao usuário para inserir números até que ele digite 0. A cada nova inserção, se o vetor estiver cheio, use `realloc` para dobrar a capacidade. Ao final, exiba todos os números inseridos (exceto o 0) e libere a memória.

4. O que acontece se você passar um ponteiro que não foi alocado com `malloc`, `calloc` ou `realloc` para a função `free`? E se você passar o ponteiro `NULL`? Escreva um pequeno programa para testar ambos os casos (compile e execute) e explique os resultados.

5.2 Seções da memória

Quando um computador inicia uma aplicação, o sistema operacional aloca automaticamente a memória necessária para a sua execução em diferentes segmentos, cada um com propriedades e funções específicas.



5.2.1 Text

A seção `Text`, contém o código executável do programa. As instruções de máquina presentes nesta seção são acessadas pelo processador para execução da aplicação. Importante destacar que esta seção é somente de leitura, o que significa que, uma vez definida, seu conteúdo não pode ser modificado durante a execução do programa. Isso é essencial para a integridade e segurança do código executável.

5.2.2 Data

A seção `Data`, é onde são armazenadas as variáveis globais e estáticas do programa que foram inicializadas durante o processo de compilação. É importante destacar que a seção de dados inicializados não é apenas de leitura; ela pode ser modificada durante a execução do programa.

Considere o seguinte código:

```
1     static char *z = "Eu sou Z\n"
2     char *w = "Eu sou W\n"
3
4     void func(void)
5     {
6         static const char *y = "Eu sou Y\n"
7     }
8
9     int main(void)
10    {
11        const char *x = "Eu sou X\n"
12        func();
13
14        return 0;
15    }
```

No código acima, são definidas as variáveis `x`, `y`, `z` e `w`. Todas elas, exceto `x`, são direcionadas para a seção `Data`. Isso ocorre porque as variáveis globais, independentemente de usarem a diretiva `static`, e as variáveis locais que utilizam a diretiva `static`, são alocadas na seção `Data` caso sejam inicializadas corretamente.

5.2.3 BSS

A seção `BSS`, armazena variáveis globais e estáticas que não foram inicializadas explicitamente pelo programador. É normal ver também a seção `BSS` ser considerada uma só com a seção `Data` por ambas conterem variáveis globais e estáticas, diferenciando-se apenas pelo estado de inicialização das variáveis.

Repetindo o mesmo código da seção anterior de forma ilustrativa:

```
1     static char *z;
2     char *w;
3
4     void func(void)
5     {
6         static const char *y;
7     }
8
9     int main(void)
10    {
11        const char *x;
12        func();
13
14        return 0;
15    }
```

As variáveis `y`, `z` e `w`, sendo declaradas como globais ou estáticas sem uma inicialização explícita, são alocadas na seção `BSS` e inicializadas automaticamente com zero. Isso é uma característica padrão do compilador para assegurar que todas as variáveis na seção `BSS` comecem com um estado previsível.

Por outro lado, a variável `x` apesar de não inicializada é declarada localmente, portanto não pertence à seção `BSS`. Em C, variáveis locais não inicializadas como `x` podem conter valores indeterminados, pois o compilador não realiza nenhuma inicialização automática para elas. Isso implica que `x` pode conter qualquer valor aleatório presente na stack no momento de sua declaração.

> `BSS` é uma abreviação de "Block Started By Symbol".

5.2.4 Heap

A seção `Heap` é utilizada para armazenar dados alocados e liberados dinamicamente pelo programa durante sua execução, por meio de funções como `malloc`, `calloc`, `realloc` e `free`.

É importante destacar que, ao contrário das outras seções de memória, a seção `Heap` não é necessariamente contígua na memória. Ela pode utilizar técnicas de mapeamento e alocação não contígua para ajustar o tamanho dos blocos de memória alocados dinamicamente. Devido a essa maior complexidade e dinamicidade dos blocos de memória na `Heap`, acessar variáveis localizadas nessa seção pode ser mais lento em comparação com as seções estáticas.

Além disso, a seção `Heap`, conforme ilustrado na figura 5.2, começa imediatamente após a seção `BSS` e pode crescer em direção à seção `Stack` à medida que mais memória dinâmica

é alocada e utilizada.

5.2.5 Stack

A seção de memória da Stack é parte crucial da memória de um programa em C, responsável pelo armazenamento de variáveis locais e pela manutenção do escopo de funções.

A Stack, também conhecida como pilha, é uma estrutura de dados de natureza LIFO[4], o que significa que o último item a ser inserido é o primeiro a ser retirado. No contexto da execução de um programa em C, a `Stack` é utilizada para armazenar variáveis locais, parâmetros de função e endereços de retorno.

Quando uma função é chamada, um novo frame de execução é criado na `Stack` para armazenar suas variáveis locais e outros dados relacionados. Esse frame é desalocado quando a função retorna, liberando assim a memória alocada para suas variáveis locais.

É importante ressaltar que a `Stack` possui um tamanho limitado, determinado pelo sistema operacional ou pelo compilador. Portanto, a alocação excessiva de memória na `Stack` pode levar a um estouro de pilha[5], resultando em falha na execução do programa. Além disso, como a memória na [6] é alocada de forma estática durante a compilação, não é possível alocar ou liberar memória de forma dinâmica, como acontece na `Heap`.

Exercícios

É de extrema importância que você realize os exercícios aqui elaborados para fixar o conteúdo.

1. Escreva um programa que declare uma variável global inicializada, uma variável global não inicializada, uma variável local estática inicializada dentro de uma função e uma variável local automática (não estática). Dentro da função, modifique cada uma delas (se possível) e imprima seus valores e tente explicar em qual seção de memória (Text, Data, BSS, Stack, Heap) cada variável é armazenada

2. Analise o código a seguir e identifique quais variáveis serão alocadas na seção `Data` e quais na seção `BSS`. Justifique sua resposta considerando se são globais/estáticas e se são inicializadas ou não.

[4] Last In, First Out
[5] também conhecido como Stack Overflow
[6] Stack

```c
int global_a = 10;
int global_b;
static float static_c = 3.14;
static double static_d;

void func(void) {
    static char static_e = 'A';
    static long static_f;
    int local_g;
}
```

3. Considerando a figura dos segmentos de memória, descreva o que acontece quando o programa solicita alocações dinâmicas muito grandes através de `malloc`. Em que direção a seção `Heap` cresce? O que pode ocorrer se a seção `Heap` e `Stack` se encontrarem?

Capítulo 6

Extra

6.1 Manipulação de bits

A manipulação de bits é uma técnica fundamental na programação em C que permite operar diretamente nos bits individuais de variáveis. Essas operações são extremamente eficientes e são amplamente utilizadas em sistemas embarcados, programação de baixo nível, criptografia, compressão de dados e otimização de performance.

Já que é possível complicar as coisas facilmente quando se trata de operações envolvendo manipulação de valores em binários diretamente, vou tentar manter a explicação simples e direta com alguns exemplos sobre o assunto:

6.1.1 Operadores Bitwise

C fornece seis operadores para manipulação de bits, que operam em cada bit individualmente:

- `~` (NOT) - Complemento de bits
- `&` (AND) - E bit a bit
- `|` (OR) - OU bit a bit
- `^` (XOR) - OU exclusivo bit a bit
- `<<` (Left Shift) - Deslocamento à esquerda
- `>>` (Right Shift) - Deslocamento à direita

```
1     ...
2     int main(void)
3     {
4
5         // a = 5 (00000101 em binário)
6         // b = 9 (00001001 em binário)
7         unsigned int a = 5, b = 9;
8
9         // O resultado é 00000001
10        printf("a&b = %u\n", a & b);
11
12        // O resultado é 00001101
13        printf("a|b = %u\n", a | b);
14
15        // O resultado é 00001100
16        printf("a^b = %u\n", a ^ b);
17
18        // O resultado é 11111111111111111111111111111010
19        // (assumindo um inteiro de 32 bits)
20        printf("~a = %u\n", a = ~a);
21
22        // O resultado é 00010010
23        printf("b<<1 = %u\n", b << 1);
24
25        // O resultado é 00000100
26        printf("b>>1 = %u\n", b >> 1);
27        return 0;
28        }
```

6.1.2 Truques úteis

Definir/inverter/limpar um bit

Usando deslocamento bit-a-bit é possível definir/inverter/limpar bits facilmente.

```
1     ...
2     // Definir um número com apenas o x-ésimo bit como 1.
3     int a = 1 << x;
4     // a = 16
5     int a = 1 << 4;
6
7     // Definir um número com todos os bits como 1,
8     // exceto o x-ésimo bit.
```

```
9     int a = ~(1 << x);
10    // a = 4294967279
11    int a = ~(1 << x);
12    ...
```

Verificando um bit específico

O valor do x-ésimo bit pode ser verificado deslocando o número x posições para a direita, em seguida, realizamos uma operação AND (&) com 1.

```
1     ...
2     bool checar_bit(unsigned int numero, int index)
3     {
4         return (number >> x) & 1;
5     }
6     ...
```

Verificando se o número é ímpar ou par

Usando uma simples verificação de AND (&) é possível verificar se um número é ímpar ou par sem precisar de nenhum outro tipo de verificação mais complexa.

```
1     ...
2     bool divisivel_por_dois(int n)
3     {
4         return !(n & 1);
5     }
6     ...
```

A vantagem de usar o operador de AND para verificar se um número é ímpar ou par ao invés de `return (n % 2) == 0;` é que o compilador da linguagem C consegue otimizar o código muito mais, o que pode ser muito útil em sistemas críticos que necessitem do máximo de desempenho possível.

A nível de curiosidade, essa é a diferença entre ambos os códigos quando compilados pelo `x86-64 gcc 15.2` usando a flag `-O3`.

Usando o método do operador AND:

```
1         ...
2         div_by_2_bitwise:
3         mov     esi, edi
4         test    dil, 1
```

```
5           je      .L2
6           mov     edi, OFFSET FLAT:.LC0
7           xor     eax, eax
8           jmp     printf
9           ...
```

Usando o método do operador AND:

```
1           ...
2           div_by_2_percent:
3           mov     edx, edi
4           mov     esi, edi
5           shr     edx, 31
6           lea     eax, [rdi+rdx]
7           and     eax, 1
8           sub     eax, edx
9           cmp     eax, 1
10          je      .L6
11          mov     edi, OFFSET FLAT:.LC1
12          xor     eax, eax
13          jmp     printf
14          ...
```

Mesmo sem conhecimentos em Assembly é possível notar uma diferença gritante na quantidade de instruções para que a função seja executada corretamente.

6.2 Depuração

Mais importante que saber programar é saber encontrar falhas no próprio código. Por isso, o processo de depuração (debugging) é tão crucial. Não serão poucas as vezes em que, durante a criação de um software mais complexo, você se deparará com erros que parecem não fazer sentido. Seja um novato ou alguém com décadas de experiência, todos enfrentarão esse tipo de problema.

Depurar é uma arte à parte, algo que está em constante evolução dentro do universo da programação. Existem inúmeras ferramentas e estratégias que podem ser utilizadas nesse processo; porém, serão abordadas aqui duas das ferramentas mais conhecidas, disponíveis gratuitamente e com amplo material de suporte na internet:

- **GDB**: uma ferramenta TUI (Text-based User Interface) que permite depurar aplicações em C de maneira detalhada por meio de comandos simples;
- **Valgrind**: outra ferramenta TUI, usada principalmente para detectar vazamentos de memória em programas escritos em C.

Antes de ensinar como utilizar essas duas ferramentas, vamos precisar definir um código exemplo, que deve conter alguns erros a serem consertados. Portanto, vamos considerar o seguinte código:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       int* x = malloc(sizeof(*x));
7       *x = 0;
8
9       while (*x < 10) {
10          printf("X é menor que 10!\n");
11          x += 1;
12      }
13
14      return 0;
15  }
```

O objetivo do código acima é simples: alocar memória para um inteiro `x`, armazenar o valor 0 nessa posição, incrementar esse valor de 1 em 1 enquanto ele for menor que 10 e exibir um texto na tela a cada iteração. Nesse ponto, a execução da aplicação deve ser finalizada.

Quando compilado, o código não gera **erros de compilação**, apenas alguns *warnings* (avisos) que podem ser ignorados dependendo das configurações. Porém, ao executá-lo, você vai ter contato ao que chamamos de **undefined behavior** (ou comportamento indefinido) pois não é possível dizer como o código vai se comportar, podendo acontecer qualquer uma das seguintes situações:

- Executar o `printf` uma única vez.
- Executar o `printf` várias vezes.
- Entrar em um looping infinito.
- Código crashar com um erro de segmentação de memória.

No meu caso, ao executar o código na minha máquina foi possível obter um erro de segmentação de memória, como é mostrado abaixo:

```
1   X é menor que 10!
2   X é menor que 10!
3   X é menor que 10!
4   ...
5   X é menor que 10!
6   Segmentation fault
```

Para resolver isso, vamos aprender a utilizar as ferramentas de debugging.

Vale ressaltar que para utilizar ferramentas de debugging de maneira mais prática e eficiente, é necessário que seja utilizada a flag `-g` durante o processo de compilação de uma aplicação. Por exemplo:

```
gcc -o main main.c -g
```

Usar essa opção faz com que o programa seja compilado com as informações extras de depuração que facilitam MUITO o uso de ferramentas como o GDB ou Valgrind. Nos exemplos abaixo vamos levar em consideração que você compilou o programa corretamente usando a flag `-g`

6.2.1 GDB

A princípio o GDB acaba sendo fundamental por ser uma ferramenta que permite controlar a execução de um programa, possibilitando inspecionar o código linha a linha, definir breakpoints (pontos de interrupção), verificar o estado das variáveis, entre outras funcionalidades.

Para entender seu funcionamento, vamos aprender alguns comandos básicos. Para iniciar a depuração, é necessário abrir o programa pelo GDB com o seguinte comando:

```
gdb [nome_do_programa]
```

Após executar esse comando, o GDB carrega o programa em um contexto de depuração, sem executá-lo imediatamente. Se tudo ocorrer corretamente, você deve ver no terminal:

```
Reading symbols from main...
(gdb)
```

A partir daí, é possível definir pontos de interrupção para investigar possíveis erros. Para visualizar o código-fonte, utiliza-se o comando:

```
list [nome_da_funcao]
// ou
l [nome_da_funcao]
```

Este comando exibe o código da função especificada com suas respectivas numerações de linha. Em nosso caso, usaremos para visualizar a função `main`.

Vale mencionar que o comando `list` exibe uma quantidade limitada de linhas por vez. Para ver as próximas linhas, basta digitar o comando novamente ou pressionar Enter. Ao chegar ao final da função, uma mensagem de aviso será exibida.

```
(gdb) l main
1       #include <stdio.h>
2       #include <stdlib.h>
3
4       int main()
5       {
6           int* x = malloc(sizeof(*x));
7           *x = 0;
8
9           while (*x < 10) {
10              printf("X é menor que 10!\n");
(gdb)
11              x += 1;
12          }
13
14          return 0;
15      }
(gdb)
Line number 16 out of range; main.c has 15 lines.
```

Tendo o número de cada linha de código é possível definir os pontos de interrupção, neste caso, vamos definir um ponto na função `main` com o seguinte comando:

```
1   break main
2   // ou
3   b main
```

Este comando cria um **breakpoint** no início da função especificada e exibe uma saída semelhante a:

```
1   (gdb) break main
2   Breakpoint 1 at 0x1151: file main.c, line 6.
```

No programa de exemplo não há muitas funções para depurar, porém, em códigos maiores e/ou mais complexos, bons pontos de interrupção podem ser definidos ao responder uma pergunta simples: ”Onde o código começou a apresentar problemas?”.

No fim das contas, trata-se de um trabalho investigativo (e por vezes cansativo), mas com a prática você desenvolverá uma noção de onde e como procurar.

Em seguida, inicia-se o processo de depuração com o comando `run` (ou `r`):

```
(gdb) r
Starting program: /file/path/main
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/
   libthread_db.so.1".

Breakpoint 1, main () at main.c:6
6           int* x = malloc(sizeof(*x));
```

Nessa saída, é possível observar informações como:

- O caminho do programa que está sendo depurado
- O identificador do **breakpoint** ativo
- A função em execução
- A próxima linha de código a ser executada

Neste momento, a linha 6 ainda não foi executada, portanto não há nada a verificar. Para executar a linha atual e avançar para a próxima, utiliza-se o comando `next` (ou `n`):

```
(gdb) n
7           *x = 0;
(gdb) n
9           while (*x < 10)
```

Para verificar o valor de variáveis vamos usar o seguinte comando:

```
print nome_da_variavel
// ou
p nome_da_variavel
```

Como executamos as linhas que alocam memória para a variável `x` usando a função `malloc` e definimos o seu valor para 0, podemos verificar se as operações foram realizadas corretamente:

```
(gdb) p x
$3 = (int *) 0x5555555592a0
(gdb) p *x
$4 = 0
```

Note que foi necessário desreferenciar o ponteiro para acessar o valor armazenado na memória alocada. Sem a desreferência, o GDB exibe apenas o endereço da variável em formato hexadecimal.

Até este ponto, nada de anormal foi detectado. Portanto, vamos continuar com a depuração passo a passo, usando apenas os comandos abordados acima:

```
1     (gdb) n
2     9           while (*x < 10) {
3     (gdb) p x
4     $1 = (int *) 0x5555555592a0
5     (gdb) p *x
6     $2 = 0
7     (gdb) n
8     10              printf("X é menor que 10!\n");
9     (gdb) n
10    X é menor que 10! // Programa exibindo o output de printf
11    11              x += 1;
12    (gdb) n
13    9           while (*x < 10) {
14    (gdb) p x
15    $3 = (int *) 0x5555555592a4 // Ponteiro foi incrementado em
          vez do valor!!!!
16    (gdb) p *x
17    $4 = 0
```

Como é possível observar no comentário, uma simples verificação das operações revelou que o comportamento inesperado do código era causado por um pequeno erro na linha 11:

```
1     x += 1;  // Incrementando o ponteiro
2     // Deve ser corrigido para
3     *x += 1; // Incrementando o valor
```

Após identificar o erro, podemos encerrar a depuração com o comando `quit`:

```
1     (gdb) quit
2     A debugging session is active.
3
4             Inferior 1 [process 19336] will be killed.
5
6     Quit anyway? (y or n)
```

Com a correção aplicada e o comportamento indefinido eliminado, ao executar o programa, obtemos o resultado esperado:

```
$ ./main
X é menor que 10!
X é menor que 10!
X é menor que 10!
X é menor que 10!
X é menor que 10!
X é menor que 10!
X é menor que 10!
X é menor que 10!
X é menor que 10!
X é menor que 10!
```

Com as informações desta seção, é possível perceber que, com apenas alguns comandos básicos, já é possível realizar processos de depuração de maneira relativamente eficaz.

O GDB é uma ferramenta que oferece comandos mais avançados, mas como não é o foco deste livro se aprofundar nela, recomendo a você que pesquise mais sobre a ferramenta posteriormente para explorar todo o seu potencial.

Vale mencionar que, durante os exemplos de depuração acima, repeti várias vezes comandos como `next`. Entretanto, você pode simplesmente pressionar **Enter** para que o GDB repita o último comando, facilitando a passagem por trechos mais longos de código.

Além disso, existem *plugins* que podem ser utilizados em conjunto com o GDB. Um que utilizo bastante é o **GEF (GDB Enhanced Features)**, que adiciona funcionalidades extras e uma interface melhorada à ferramenta original.

6.2.2 Valgrind

O Valgrind, apesar de possuir um funcionamento bem diferente do GDB, é uma ferramenta igualmente fundamental. Ele serve como um meio simples e relativamente fácil de usar para detectar vazamentos de memória. O programa funciona da seguinte forma: o usuário deve passar algumas flags juntamente com o executável que será verificado; o Valgrind então executa o programa e gera um relatório final com informações sobre possíveis problemas encontrados.

Para exemplificar o uso do Valgrind, continuaremos utilizando o código corrigido na seção anterior, como forma de complementar as correções de erros que o GDB não nos alerta.

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

int main()
{
    int* x = malloc(sizeof(*x));
    *x = 0;

    while (*x < 10) {
        printf("X é menor que 10!\n");
        *x += 1;
    }

    return 0;
}
```

À primeira vista, o código aparenta funcionar corretamente, sem qualquer tipo de erro. Isso se deve ao fato de que o problema presente no código não é um erro de lógica, mas sim um vazamento de memória. Para não adiantar a resposta, vamos tentar descobrir o erro usando o Valgrind.

Para verificar o código por meio do Valgrind, precisamos novamente compilar o programa utilizando a flag `-g`, assim como foi feito na seção anterior para depuração com o GDB:

```
gcc -o main main.c -g
```

Em seguida, para iniciar a busca por vazamentos de memória, utilizamos um único comando no terminal com algumas flags específicas. O Valgrind executará o programa e gerará um relatório ao final:

```
valgrind --tool=memcheck --leak-check=yes --show-reachable=
   yes --num-callers=20 --track-fds=yes ./main
```

Explicação das flags

- **–tool=memcheck**: Especifica qual ferramenta do Valgrind será utilizada. O `memcheck` é a ferramenta padrão para verificação de memória e será usada neste exemplo.
- **–leak-check=yes**: Ativa ou desativa a busca por vazamentos de memória ao final da execução. Quando definido como `yes` ou `full`, todos os vazamentos individuais são exibidos em detalhes, incluindo sua classificação como possíveis erros.
- **–show-reachable=yes**: Define se devem ser exibidas no relatório as regiões de memória que não foram liberadas ao sistema, mas que ainda possuíam ponteiros apontando para elas ao final da execução do programa.

- –**num-callers=20**: Define a profundidade do *stacktrace* (rastreamento da pilha de chamadas), ou seja, quantas entradas serão exibidas para rastrear os pontos onde os erros ocorreram no código. O valor pode variar de 0 a 500. É importante notar que um número maior de entradas pode tornar a execução mais lenta, pois o programa precisa gerenciar e armazenar mais informações.
- –**track-fds=yes**: Faz com que sejam exibidos, além dos vazamentos de memória, os descritores de arquivo (*file descriptors*) que permaneceram abertos ao final da execução. O relatório inclui informações sobre onde no código o arquivo foi aberto, além de detalhes sobre *sockets* e nomes de arquivo quando disponíveis.
- **./main**: O executável a ser analisado pela ferramenta.

Depois de executar o comando acima, o seguinte relatório é exibido no terminal:

```
1  ==4405== Memcheck, a memory error detector
2  ==4405== Copyright (C) 2002-2022, and GNU GPL'd, by Julian
       Seward et al.
3  ==4405== Using Valgrind-3.19.0 and LibVEX; rerun with -h
       for copyright info
4  ==4405== Command: ./main
5  ==4405==
6  X é menor que 10!
7  X é menor que 10!
8  X é menor que 10!
9  X é menor que 10!
10 X é menor que 10!
11 X é menor que 10!
12 X é menor que 10!
13 X é menor que 10!
14 X é menor que 10!
15 X é menor que 10!
16 ==4405==
17 ==4405== FILE DESCRIPTORS: 3 open (3 std) at exit.
18 ==4405==
19 ==4405== HEAP SUMMARY:
20 ==4405==     in use at exit: 4 bytes in 1 blocks
21 ==4405==   total heap usage: 2 allocs, 1 frees, 1,028 bytes
       allocated
22 ==4405==
23 ==4405== 4 bytes in 1 blocks are definitely lost in loss
       record 1 of 1
24 ==4405==    at 0x48417B4: malloc (vg_replace_malloc.c:381)
25 ==4405==    by 0x10915A: main (main.c:6)
26 ==4405==
27 ==4405== LEAK SUMMARY:
28 ==4405==    definitely lost: 4 bytes in 1 blocks
29 ==4405==    indirectly lost: 0 bytes in 0 blocks
30 ==4405==      possibly lost: 0 bytes in 0 blocks
31 ==4405==    still reachable: 0 bytes in 0 blocks
32 ==4405==         suppressed: 0 bytes in 0 blocks
33 ==4405==
34 ==4405== For lists of detected and suppressed errors, rerun
       with: -s
35 ==4405== ERROR SUMMARY: 1 errors from 1 contexts (
       suppressed: 0 from 0)
```

Pelas informações do relatório podemos ver facilmente que, foram perdidos 4 bytes em 1 bloco de memória, que foi alocado na função `main`, especficicamente na linha 6. Portanto, tudo que devemos fazer é verificar na linha 6 em que variável foi guardada a o bloco de memória perdido e verificar porquê ele não foi liberado corretamente.

```
1    ...
2    int* x = malloc(sizeof(*x));
3    ...
```

No código é possível notar que após alocarmos memória para a variável `x`, em nenhum outro momento o espaço de memória da variável foi liberado de volta ao sistema através da função `free`, o que ocasiona num problema de vazamento de memória que não é informado ao usuário em momento algum durante a execução do programa, mas que em sistemas críticos ou em produtos reais que possuem um hardware mais limite, pode ocasionar em problemas graves.

Por fim, tudo que precisamos fazer para solucionar o caso acima, é usar devidamente a função `free` para liberar o espaço de memória alocado para `x` quando ele não for mais ser utilizado pelo programa.

```
1    #include <stdio.h>
2    #include <stdlib.h>
3
4    int main()
5    {
6        int* x = malloc(sizeof(*x));
7        *x = 0;
8
9        while (*x < 10) {
10           printf("X é menor que 10!\n");
11           *x += 1;
12       }
13
14       free(x); // Libera o espaço de memória alocado para 'x'
15       return 0;
16   }
```

Após fazer a correção exibida no código acima, é possível verificar o código novamente pelo Valgrind utilizando o mesmo comando, e teremos o seguinte relatório:

```
1  ==4669== Memcheck, a memory error detector
2  ==4669== Copyright (C) 2002-2022, and GNU GPL\'d, by Julian
       Seward et al.
3  ==4669== Using Valgrind-3.19.0 and LibVEX; rerun with -h
      for copyright info
4  ==4669== Command: ./main
5  ==4669==
6  X é menor que 10!
7  X é menor que 10!
8  X é menor que 10!
9  X é menor que 10!
10 X é menor que 10!
11 X é menor que 10!
12 X é menor que 10!
13 X é menor que 10!
14 X é menor que 10!
15 X é menor que 10!
16 ==4669==
17 ==4669== FILE DESCRIPTORS: 3 open (3 std) at exit.
18 ==4669==
19 ==4669== HEAP SUMMARY:
20 ==4669==     in use at exit: 0 bytes in 0 blocks
21 ==4669==   total heap usage: 2 allocs, 2 frees, 1,028 bytes
       allocated
22 ==4669==
23 ==4669== All heap blocks were freed -- no leaks are
      possible
24 ==4669==
25 ==4669== For lists of detected and suppressed errors, rerun
       with: -s
26 ==4669== ERROR SUMMARY: 0 errors from 0 contexts (
      suppressed: 0 from 0)
```

Como é possível notar lendo o relatório acima, o programa continua executando da mesma forma. Porém, desta vez, sem nenhum vazamento de memória, já que todos os espaços de memória alocados foram liberados corretamente.

6.2.3 Outras estratégias

Nesta subseção será abordada uma técnica mais informal de depuração em C, porém ainda bastante utilizada em casos mais simples onde um processo de depuração mais rudimentar seja satisfatório.

Um método que, por mais simples que pareça, pode ser utilizado em alguns casos é o uso

de `printf` para verificar valores de um módulo específico do código, ajudando a identificar se algo está sendo calculado corretamente ou até mesmo para confirmar se determinadas seções estão sendo executadas na ordem esperada.

Considere o seguinte código que soma apenas os números pares de um vetor, mas apresenta um comportamento inesperado:

```
1   #include <stdio.h>
2
3   int soma_pares(int* arr, int n) {
4       int soma = 0;
5       for (int i = 0; i <= n; i++) {
6           if (arr[i] % 2 == 0) {
7               soma += arr[i];
8           }
9       }
10      return soma;
11  }
12
13  int main() {
14      int numeros[] = {1, 2, 3, 4, 5, 6};
15      int tamanho = 6;
16      int resultado = soma_pares(numeros, tamanho);
17      printf("Soma dos pares: %d\n", resultado);
18      return 0;
19  }
```

O objetivo do código é simples: percorrer o vetor `numeros`, acumular apenas os valores pares e exibir o resultado. O valor esperado seria `12` $(2 + 4 + 6)$, porém ao executar o programa, o resultado obtido é diferente:

```
1   Soma dos pares: 1027435228
```

O resultado pode variar entre execuções. Esse é um sinal claro de **undefined behavior**. Caso você tenha suspeitas diretas do porquê disso, é possível utilizar o `printf`, antes de usar ferramentas mais avançadas como o GDB, para inspecionar o que está acontecendo a cada iteração do laço:

```c
#include <stdio.h>

int soma_pares(int* arr, int n) {
    int soma = 0;
    for (int i = 0; i <= n; i++) {
        printf("[debug] i=%d | arr[i]=%d | soma=%d\n", i,
           arr[i], soma);
        if (arr[i] % 2 == 0) {
            soma += arr[i];
        }
    }
    return soma;
}

int main() {
    int numeros[] = {1, 2, 3, 4, 5, 6};
    int tamanho = 6;
    int resultado = soma_pares(numeros, tamanho);
    printf("Soma dos pares: %d\n", resultado);
    return 0;
}
```

Ao executar o código modificado, obtemos uma saída semelhante a:

```
[debug] i=0 | arr[i]=1 | soma=0
[debug] i=1 | arr[i]=2 | soma=0
[debug] i=2 | arr[i]=3 | soma=2
[debug] i=3 | arr[i]=4 | soma=2
[debug] i=4 | arr[i]=5 | soma=6
[debug] i=5 | arr[i]=6 | soma=6
[debug] i=6 | arr[i]=812595920 | soma=12
Soma dos pares: 812595932
```

A saída torna o problema evidente, na iteração `i=6`, o programa acessa `arr[6]`, que está **fora dos limites do vetor** (cujos índices válidos vão de `0` a `5`). O valor lido é lixo de memória, e dependendo do ambiente, pode ou não alterar o resultado final, o que explica o comportamento não determinístico.

A causa do problema está na condição do laço:

```c
for (int i = 0; i <= n; i++)
// Deve ser corrigido para
for (int i = 0; i < n; i++)
```

Após aplicar a correção e remover os `printf` de depuração, o programa passa a produzir o resultado correto de forma consistente:

```
1    Soma dos pares: 12
```

Vale destacar que os `printf` inseridos para depuração devem ser removidos do código ao final do processo. Deixá-los no código de produção polui a saída do programa e pode impactar negativamente a performance em laços muito longos.

Uma prática comum para evitar esse problema é utilizar macros condicionais que ativam as mensagens de depuração apenas quando necessário, como por exemplo:

```
1          #ifdef DEBUG
2              printf("[debug] i=%d | arr[i]=%d\n", i, arr[i
                  ]);
3          #endif
```

Dessa forma, as mensagens só aparecem quando o programa é compilado com a flag `-DDEBUG`, mantendo o código de produção limpo.

6.3 Principais mudanças do C23

Em 1989, a já extremamente popular linguagem C alcançou níveis de popularidade nunca antes vistos, sendo oficialmente reconhecida pelas organizações de padronização ANSI (americana) e ISO (internacional). Esse processo culminou na padronização conhecida como `C89` (ou ANSI C), que é o dialeto da linguagem que mais reconhecemos atualmente.

Atualizações e melhorias na padronização da linguagem são lançadas, geralmente, a cada 10 anos. Atualmente, as versões mais conhecidas e utilizadas no mercado são: `C89`, `C99`, `C11` e `C18`, sendo as versões `C99` e `C11` as mais famosas entre elas.

Apesar disso, há uma versão publicada em 31 de outubro de 2024 que trouxe algumas funcionalidades bem úteis e simples de usar. Gostaria de mencioná-las neste livro, pois de certa forma procuro trazer uma visão moderna da linguagem para iniciantes.

6.3.1 Como ativar o padrão C23

Antes de mostrar qualquer funcionalidade do padrão `C23` (também chamado de `C2X`), é necessário explicar como ativá-lo no compilador. Diferentemente de linguagens modernas como JavaScript ou Python, onde funcionalidades são adicionadas por meio de pacotes, podemos ativar as funcionalidades do `C23` usando uma `flag` de compilação que indica ao GCC/Clang qual padrão desejamos, da seguinte forma:

```
gcc -o main main.c -std=c2x
\\ ou
gcc -o main main.c -std=c23
```

O comando pode variar um pouco dependendo do compilador, mas no geral deve ser como mostrado acima. A opção `-std=` também pode ser utilizada para especificar qualquer outro padrão da linguagem. Portanto, seria possível usar comandos como:

```
gcc -o main main.c -std=c18
gcc -o main main.c -std=c11
gcc -o main main.c -std=c99
gcc -o main main.c -std=c89
```

Qualquer um desses é válido como especificação do padrão a ser utilizado pelo compilador. Isso é bastante útil em projetos mais antigos ou bem consolidados em um padrão específico da linguagem, sendo necessário, nesses casos, indicar ao compilador qual padrão utilizar.

6.3.2 Separador de dígitos

O separador de dígitos é apenas um recurso visual para melhorar a legibilidade de números muito grandes, que dificilmente seriam lidos sem nenhum tipo de separador. Trata-se de uma função meramente visual, sem qualquer implicação lógica no código.

```
1  #include <limits.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6      int x = 19'251'916; // mesmo que 19251
7      long int y = 36'854'775'807'192; // mesmo que 36854775807
8
9      printf("int: %i\\n", x);
10     printf("long int: %li\\n", y);
11
12     return 0;
13 }
```

6.3.3 Literais binários

Em algumas aplicações de baixo nível, por vezes é necessário declarar valores explicitamente bit a bit. Anteriormente, isso precisava ser feito usando hexadecimal ou octal, o que nem sempre era intuitivo.

```
1 ...
2 unsigned char mascara = 0b11000101;
3 unsigned char verdadeiro = 0b1;
4 unsigned char falso = 0b0;
5 ...
```

Agora, utilizando o prefixo `0b` ou `0B` é possível definir valores em binário explicitamente, assim como mostrado acima.

6.3.4 Ponteiro nulo

Esta é outra mudança que traz melhorias de legibilidade, porém com um aspecto um pouco mais técnico. Desde a versão C89 da linguagem, a macro `NULL` tem sido definida como:

```
1 ...
2 \#define NULL ((void*)0)
3 ...
```

No geral, essa definição é satisfatória para o valor nulo. Porém, em algumas plataformas essa mesma macro acaba sendo definida simplesmente como `0`, o que gera problemas de portabilidade. Um código que depende fortemente da definição de `NULL` pode quebrar em uma plataforma onde ela é definida de maneira diferente.

Um exemplo simples de como isso poderia acontecer é com a seguinte função:

```
1 ...
2 printf("\%p\n", NULL);  // possui comportamento indefinido
3 ...
```

Caso `NULL` esteja definido como um inteiro de valor `0`, o comportamento é indefinido e pode resultar em lixo de memória, o que é algo certamente indesejado. Assim, a constante `nullptr` foi introduzida para corrigir esse problema:

```
1 ...
2 int* x = nullptr;
3 ...
```

Agora temos um ponteiro para inteiro recebendo um ponteiro nulo genérico de maneira **explícita**. Dessa forma, o compilador consegue diferenciar `nullptr` de `((void*)0)` ou de um inteiro de valor `0`. Isso resolve o problema de forma explícita, forçando o programador a deixar claro o que está sendo feito no código e também resolvendo questões de compatibilidade com outros sistemas e de interoperabilidade com C++.

6.3.5 Formato de binário para saídas/entradas

Em programação de sistemas de baixo nível, é comum ter que lidar com manipulação ou visualização de dados binários. Anteriormente, a linguagem C necessitava de funções implementadas manualmente para realizar tais tarefas. Agora, podemos usar o especificador `%b` para exibir valores em binário de forma nativa:

```c
#include <stdio.h>

int main(void) {
    int i = 67;
    printf("i: %b\n", i); // Exibe "i: 1000011"
    int j = 240;
    printf("j: %b\n", j); // Exibe "j: 11110000"
    int k = 101;
    printf("k: %b\n", k); // Exibe "k: 1100101"

    return 0;
}
```

6.3.6 Valores booleanos

Apesar de ser uma funcionalidade **muito** simples presente na maioria das linguagens, C não suporta variáveis booleanas nativamente desde suas primeiras versões. Por conta disso, sempre que era necessário implementar um sistema de `flags` usando variáveis booleanas, era preciso incluir a biblioteca `<stdbool.h>` para usar as palavras-chave `true`, `false` e `bool`.

Agora, com esse recurso nativo da linguagem, é possível fazer:

```c
#include <stdio.h>

int main(void) {
    bool v = true;
    bool f = false;

    if (v == true)
        printf("v: %i (em binário = %02b)\n", v, v);
    if (f == false)\
        printf("f: %i (em binário = %02b)\n", f, f);

    return 0;
}
```

Tudo isso sem se preocupar com inconsistências ao incluir uma biblioteca para algo tão fundamental.

6.3.7 _BitInt(N)

Em algumas aplicações de baixo nível como protocolos de rede, `hardware`, processamento de áudio, criptografia etc. É necessário trabalhar com tipos inteiros cuja largura não é potência de dois (por exemplo, 7 bits, 13 bits). Simular esses tipos com `structs` e máscaras era trabalhoso. Usando o novo tipo `_BitInt(N)` podemos fazer:

```c
#include <stdio.h>

int main(void) {
    \_BitInt(5) i = 0;
    i = 15;
    printf("i: %08b\n", i);
    i = -16;
    printf("i: %08b\n", i);

    return 0;
}
```

Neste exemplo, declaramos um inteiro de 5 bits com sinal. O bit mais significativo é o bit de sinal, e os 4 bits restantes representam a magnitude (em complemento para dois). Assim, o intervalo de valores representáveis vai de -16 a 15.

6.3.8 typeof e typeof_unqual

Ambos funcionam de forma bem semelhante, mas com uma diferença sutil que possui aplicações práticas na implementação de macros. A palavra-chave `typeof` é um operador que obtém o tipo de uma expressão em tempo de compilação. Isso significa que é possível fazer algo como:

```c
#include <stdio.h>

int main(void) {
    int i = 10;
    typeof(i) j = i; // Mesmo que "int j = i;"

    printf("i: %i\n", i); // Exibe "i: 10"
    printf("j: %i\n", j); // Exibe "j: 10"

    return 0;
}
```

O operador `typeof` sinaliza ao compilador que ele deve substituir `typeof(i)` pelo tipo da variável passada como parâmetro.

Porém, em alguns casos onde se deseja criar variáveis do mesmo tipo que outra, pode ser

indesejado trazer qualificadores como `const`, `volatile` ou `restrict`. É aí que `typeof_unqual` se mostra útil. Com ele podemos fazer:

```c
#include <stdio.h>

int main() {
    const int a = 10;
    volatile double b = 3.14;

    // typeof mantém os qualificadores
    typeof(a) x = a;         // x é const int
    typeof(b) y = b;         // y é volatile double

    // typeof_unqual remove os qualificadores
    typeof_unqual(a) u = a; // u é int (sem const)
    typeof_unqual(b) v = b; // v é double (sem volatile)

    // Possível modificar u e v sem problemas
    u = 20;
    v = 2.71;

    printf("u = %d, v = %f\n", u, v);
    return 0;
}
```

Com esses operadores é possível criar macros como:

```c
...
#define SWAP(a,b) { typeof(a) tmp = a; a=b; b=tmp; }
...
```

Essa macro recebe dois valores, utiliza `typeof` para criar uma variável `tmp` e troca os valores entre `a` e `b`. O uso de `typeof_unqual` é desnecessário nesse caso, pois não é desejado tentar modificar variáveis com qualificadores `const` ou `restrict`.

6.3.9 deprecated e nodiscard

Os atributos `deprecated` e `nodiscard` têm usos focados em legibilidade e documentação, sendo importantes principalmente para explicitar o comportamento de funções e tipos. Vamos entender cada um.

`deprecated` sinaliza que um elemento (função, variável, `struct`, etc.) está obsoleto e não deve ser usado em novos código, embora ainda funcione. Pode incluir uma mensagem opcional.

```c
#include <stdio.h>

[[deprecated("Use a função 'soma_nova()' em vez disso")]]
int soma_antiga(int a, int b)
{
    return a + b;
}

[[deprecated]]
int soma_velhaca(int a, int b)
{
    return a + b;
}

...

int main(void) {
    int a = 10;
    int b = 15;

    printf("soma: %i\n", soma_antiga(a, b));
    printf("soma: %i\n", soma_velhaca(a, b));

    return 0;
}
```

No código acima, temos duas funções obsoletas. Em bases de código antigas, é comum sinalizar funções que ainda funcionam mas serão removidas futuramente por meio de comentários. O atributo `deprecated` torna essa sinalização explícita e gera `warnings` durante a compilação, incentivando a migração para alternativas.

Observe que o atributo pode ser usado com ou sem mensagem entre aspas. O resultado da compilação do código acima exibe o seguinte:

```
main.c: In function 'main\':
main.c:19:5: warning: 'soma_antiga' is deprecated: Use a função
    'soma_nova()' em vez disso [-Wdeprecated-declarations]
   19 |     printf("soma: %i\n", soma_antiga(a, b));
      |     ^~~~~~
main.c:4:5: note: declared here
    4 | int soma_antiga(int a, int b)
      |     ^~~~~~~~~~~
main.c:20:5: warning: 'soma_velhaca' is deprecated
    [-Wdeprecated-declarations]
   20 |     printf("soma: %i\n", soma_velhaca(a, b));
      |     ^~~~~~
main.c:10:5: note: declared here
   10 | int soma_velhaca(int a, int b)
      |     ^~~~~~~~~~~~
```

Já o atributo `nodiscard` indica que o valor retornado por uma função não deve ser ignorado. Se o chamador descartar o valor, o compilador emitirá um aviso. Isso é útil para funções que alocam recursos, retornam códigos de erro ou produzem resultados essenciais.

```c
#include <stdio.h>

[[nodiscard]]
int soma_nova(int a, int b)
{
    return a + b;
}

int main(void) {
    int a = 10;
    int b = 15;

    soma_nova(a, b); // Valor não é atribuído a nenhuma variá
       vel

    return 0;
}
```

No caso acima, a função `soma_nova()` é chamada, mas o valor retornada pela mesma não é atribuída a nenhuma ou usado de nenhuma forma, o que faz com que o compilador retorne um erro como:

```
main.c: In function 'main':
main.c:12:5: warning: ignoring return value of 'soma_nova', declared with
    attribute 'nodiscard' [-Wunused-result]
   12 |     soma_nova(a, b); // Valor não é atribuído a nenhuma variável
      |     ^~~~~~~~~~~~~~
main.c:3:19: note: declared here
    3 | [[nodiscard]] int soma_nova(int a, int b)
      |                   ^~~~~~~~~
```

Dito isso, é notável como ambos os atributos ajudam a tornar o código mais robusto e auto-documentado.

É possível escrever mensagens de erro personalizadas da mesma forma que é feito com o atributo `deprecated`.

```
1 [[nodiscard("O valor de soma_nova() foi descartado")]]
```

Desta forma, caso o valor retornado pela função não seja usado devidamente, a mensagem de erro personalizada deve aparecer como um erro durante o processo de compilação.

6.3.10 #elifdef e #elifndef

Essas duas diretivas foram introduzidas apenas para tornar a definição condicional de macros mais conveniente. Antes, para escrever código que dependesse do sistema operacional, por exemplo, era necessário algo como:

```c
#include <stdio.h>

int main(void)
{
    #ifdef _WIN32
        printf("Executando no Windows\n");
    #elif defined(__linux__)
        printf("Executando no Linux\n");
    #elif defined(__APPLE__)
        printf("Executando no macOS\n");
    #else
        printf("Sistema operacional desconhecido\n");
    #endif

    return 0;
}
```

Com as novas diretivas, o mesmo código pode ser reescrito de forma mais limpa e direta:

```c
#include <stdio.h>

int main(void)
{
    #ifdef _WIN32
    printf("Executando no Windows\n");
    #elifdef __linux__
        printf("Executando no Linux\n");
    #elifdef __APPLE__
        printf("Executando no macOS\n");
    #else
        printf("Sistema operacional desconhecido\n");
    #endif

    return 0;
}
```

A diretiva `#elifdef` é equivalente a `#elif defined(...)`, e `#elifndef` equivale a `#elif !defined(...)`. Elas tornam o código mais legível, especialmente em cadeias longas de condições.

6.3.11 #embed

Esta é, provavelmente, uma das funcionalidades mais úteis, pelo menos na minha opinião, dentre todas as apresentadas. Trata-se de uma diretiva de pré-processamento que permite

incluir dados binários diretamente no executável final gerado pela compilação.

Antes, era comum usar ferramentas como `xxd`, *scripts* para gerar arrays em C ou até mesmo a função `fopen()` para ler arquivos durante a execução. A nova diretiva, no entanto, resolve tudo em tempo de compilação, sem custos de leitura em tempo de execução. Além disso, integra-se bem com sistemas de *build*, que podem detectar alterações no arquivo incorporado e recompilar automaticamente. Tudo isso de forma bem simples, como no exemplo:

```
1 // Incluindo um arquivo de shaders que deve ser utilizado pelo
     programa
2 static const char arquivo_shaders[] = {
3     #embed "shaders/vertex.glsl"
4     , '\0'  // garante terminação nula para o array de bytes do
          arquivo
5 };
```

O código acima incorpora o arquivo `"shaders/vertex.glsl"` no binário gerado pelo compilador e adiciona um caractere nulo ao final do array para garantir que a string seja terminada corretamente.

6.3.12 memset_explicit()

Essa função foi adicionada para resolver problemas de segurança criados por uma interação, um tanto contraditória, entre o compilador e a função `memset()`. A função `memset()` é geralmente usada para limpar áreas de memória que armazenaram informações sensíveis. Por exemplo:

```
1  #include <string.h>
2
3  void processa_senha(const char *senha)
4  {
5      char buffer[64];
6      strcpy(buffer, senha);
7      // ... usa a senha para algo (ex.: autenticar conta)
8
9      // Tenta limpar a senha da memória
10     memset(buffer, 0, sizeof(buffer));
11 }
```

Nesse exemplo, após a chamada de `memset()`, a variável `buffer` não é mais usada. O compilador, ao otimizar o código com as flags `-O2` ou `-O3`, pode remover completamente a chamada de `memset()`, pois considera que escrever em uma variável que logo será descartada é redundante. Com isso, a senha permanece na pilha até ser sobrescrita por outra função, o que pode ser explorado por usuários mal-intencionados para recuperar dados sensíveis que ainda estão na memória.

A função `memset_explicit()` resolve exatamente esse problema: ela foi projetada para ser imune a otimizações do compilador, ou seja, será chamada independentemente do nível de otimização usado. O resultado é um código mais seguro. Basta substituir `memset()` por `memset_explicit()`:

```c
#include <string.h>

void processa_senha(const char *senha)
{
    char buffer[64];
    strcpy(buffer, senha);
    // ... uso da senha

    // Garante a limpeza do buffer da senha
    memset_explicit(buffer, 0, sizeof(buffer));
}
```

No geral, estas são as adições mais relevantes trazidas pelo novo padrão C23. O foco dessa padronização foi remover funcionalidades obsoletas, melhorar a segurança, a portabilidade e a compatibilidade com C++.

Dentro das limitações da linguagem e do que fazia sentido ser modificado ou adicionado, é possível dizer que o novo padrão trouxe melhorias significativamente boas para o C.

No entanto, como já dito, aqui foram abordadas apenas as mudanças mais relevantes. Caso você queira saber **tudo** o que foi adicionado, recomendo fortemente procurar a lista completa de alterações e lê-la por conta própria. Como programador C, é importante ter a capacidade de buscar fontes de conhecimento e saber como se atualizar.

Capítulo 7

Considerações finais

7.0.1 Recursos e referências adicionais

7.0.2 Obrigado

Primeiramente, se você realmente leu este livro até o final, resolveu todos os exercícios e buscou compreender verdadeiramente tudo o que foi apresentado, espero que a leitura tenha sido proveitosa e que eu tenha conseguido, ao menos, lhe ensinar algo novo.

Não sou um escritor ávido, nem sequer acredito ser um bom professor, mas espero que meu esforço ao escrever este e-book tenha ajudado alguém. Conhecendo a mim mesmo, futuramente certamente enxergarei muitos problemas e pontos de melhoria neste livro, e espero que você também possa apontar erros e sugerir aprimoramentos. Quem sabe até enviar um pull request no GitHub com correções ou com trechos que poderiam ser explicados de forma mais clara ou objetiva.

Enfim, o conteúdo técnico deste livro terminou no início desta seção. A única coisa que me resta dizer é: obrigado pela oportunidade de compartilhar o que tinha para ensinar e por dar uma chance a este livro, escrito nas horas vagas. Quem sabe um dia eu faça uma segunda edição! Afinal, o padrão C2Y foi anunciado enquanto este livro ainda estava sendo produzido, a área da computação está sempre evoluindo, e sempre haverá algo novo para aprender.