DOMINANDO A LINGUAGEM C EM 1 ANO

Caderno de Respostas

Gabarito comentado: 52 aulas de prática, do #01 ao #52

C17 · gcc ou clang · 9 módulos · 71 questões · 2 desafios integradores

Autor Desconhecido · theknowledgeisle.com

Consulte este gabarito só depois de tentar.

As respostas são comentadas para ensinar, não só para conferir. Nos exercícios de código existem outras soluções igualmente válidas: o que importa é entender por que a sua funciona e o que ela troca em relação à mostrada aqui.

Os programas com um quadro verde de **Saída** foram compilados com clang (-std=c17 -Wall -Wextra) e executados no Linux; com gcc o resultado é o mesmo, e a saída mostrada é a real.

Sumário

Módulo	Tema	Aulas	Questões
MÓDULO 1	Fundamentos da linguagem	#01–#06	Q1–Q9
MÓDULO 2	Ponteiros e strings	#07–#10	Q10–Q16
MÓDULO 3	Memória: pilha, heap e vazamentos	#11–#13	Q17–Q23
MÓDULO 4	Tipos próprios, pré-processador e compilação	#14–#20	Q24–Q32
MÓDULO 5	Entrada, saída e biblioteca padrão	#21–#25	Q33–Q40
MÓDULO 6	Ponteiros para função, recursão e bits	#26–#30	Q41–Q47
MÓDULO 7	Estruturas de dados	#31–#35	Q48–Q54
MÓDULO 8	Ferramentas, qualidade e comportamento indefinido	#36–#46	Q55–Q63
MÓDULO 9	Programação de sistemas	#47–#51	Q64–Q71
DESAFIOS	Desafios integradores	todas	D1–D2

MÓDULO 1

Fundamentos da linguagem

Por que C existe, tipos e memória, funções, decisões, laços e o primeiro vetor.

Artigos 01 a 06

#01 Por que C existe e o que ele quer de você
#02 Variáveis, Tipos e a Memória por Trás Deles
#03 Funções, Escopo e Passagem por Valor
#04 Quando o Programa Decide: if, else e switch
#05 A Arte da Repetição com while, do-while e for
#06 Vetores e o Primeiro Contato com a Memória Contígua

Perguntas conceituais — respostas

Q1 O que C quer de você

> *O artigo #01 diz que C "confia no programador". O que isso significa na prática? Cite duas coisas que a linguagem **não** faz por você e que linguagens como Python ou Java fazem.*

Significa que o compilador gera exatamente o que foi pedido, sem verificações escondidas no caminho: o custo de cada operação fica visível, e a responsabilidade pelos erros também. Duas coisas que C não faz:

- **Verificar limites de vetor**: `v[10]` num vetor de 10 elementos compila e executa, lendo ou escrevendo memória que não é do vetor (comportamento indefinido).
- **Liberar memória**: o que foi alocado com `malloc` fica alocado até um `free`; não há coletor de lixo.

Outras respostas válidas: não inicializa variáveis locais, não detecta estouro de inteiro com sinal, não tem exceções (erros voltam como valor de retorno).

Q2 Tamanho dos tipos

> *Por que o padrão C não fixa que `int` tem 32 bits? O que ele garante sobre `char`, `short`, `int` e `long`? Quando usar `int32_t` de `<stdint.h>`?*

Porque C nasceu para rodar em máquinas muito diferentes, e o `int` deve ser o tamanho "natural" de cada uma. O padrão garante só **mínimos e uma ordem**: `char` tem pelo menos 8 bits e `sizeof(char)` é 1 por definição; `short` e `int` têm pelo menos 16 bits; `long` pelo menos 32; e `sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`.

Use `int32_t` (e companhia) quando o tamanho exato importa: formatos de arquivo, protocolos de rede, registradores de hardware. Para contadores e índices comuns, `int` ou `size_t` bastam.

Q3 Passagem por valor

> *"Em C, toda passagem de argumento é por valor." Explique a frase e diga como, então, uma função consegue alterar uma variável de quem a chamou.*

A função recebe **cópias** dos argumentos: alterar o parâmetro dentro dela não muda a variável original. Para alterar a variável de quem chamou, passa-se o **endereço** dela (um ponteiro); o ponteiro também é copiado, mas a cópia aponta para o mesmo lugar, e `*p = ...` escreve na variável original. É o caso do `scanf("%d", &x)`.

Q4 switch e o break esquecido

> *O que acontece quando um `case` de um `switch` não termina com `break`? Dê uma situação em que isso é intencional.*

A execução **continua no case seguinte** (*fall-through*), sem testar o rótulo dele. É intencional quando vários valores levam à mesma ação:

```
switch (c) {
case 'a': case 'e': case 'i': case 'o': case 'u':
    vogais++;
    break;
default:
    outras++;
}
```

Quando não é intencional, é bug; o gcc e o clang avisam com `-Wimplicit-fallthrough`, e o C23 tem o atributo `[[fallthrough]]` para marcar o caso desejado.

Q5 Vetor como argumento

> *Por que, dentro de uma função `void f(int v[])`, a expressão `sizeof v / sizeof v[0]` não dá o número de elementos? Como as funções da biblioteca resolvem isso?*

Porque um parâmetro declarado como vetor **é um ponteiro**: o vetor "decai" para o endereço do primeiro elemento na chamada. Dentro de `f`, `sizeof v` é o tamanho de um ponteiro (8 bytes num sistema de 64 bits), não do vetor. A solução é passar o tamanho junto, como fazem `qsort(base, n, tamanho, cmp)`, `memcpy(dest, orig, n)` e `fgets(buf, tamanho, arquivo)`; strings resolvem de outro jeito, com o `'\0'` no fim.

Exercícios de código — respostas

Q6 Divisão inteira e conversões

Sem executar, diga o que o programa imprime. Depois execute e explique cada linha.

```c
#include <stdio.h>

int main(void) {
    int a = 7, b = 2;
    printf("%d\n", a / b);
    printf("%d\n", -7 / 2);
    printf("%d\n", -7 % 2);
    printf("%.2f\n", a / b * 1.0);
    printf("%.2f\n", (double) a / b);
    unsigned int u = 0;
    printf("%u\n", u - 1);
    return 0;
}
```


```c
#include <stdio.h>

int main(void) {
    int a = 7, b = 2;
    printf("%d\n", a / b);
    printf("%d\n", -7 / 2);
    printf("%d\n", -7 % 2);
    printf("%.2f\n", a / b * 1.0);
    printf("%.2f\n", (double) a / b);
    unsigned int u = 0;
    printf("%u\n", u - 1);
    return 0;
}
```


Saída

```
3
-3
-1
3.00
3.50
4294967295
```

Entre inteiros, `/` descarta a parte fracionária **em direção ao zero** (desde o C99), por isso `-7 / 2` é `-3`, e o resto acompanha o sinal do dividendo: `-7 % 2` é `-1`. Em `a / b * 1.0` a divisão inteira acontece primeiro (`3`) e só depois vira `double`; com `(double) a / b` a conversão vem antes da divisão. Por fim, `unsigned` não tem negativo: `0 - 1` dá a volta e vira o maior valor do tipo, 4294967295 num `unsigned int` de 32 bits. Isso é definido pelo padrão; já o estouro de um `int` com sinal é comportamento indefinido.

Q7 Troca com ponteiros

Escreva `void troca(int *a, int *b)` que troca os valores de duas variáveis e mostre, no mesmo programa, por que uma versão `void troca_errada(int a, int b)` não funciona.

```c
#include <stdio.h>

void troca_errada(int a, int b) {
    int t = a;
    a = b;
    b = t;  /* troca as cópias, que somem ao retornar */
}

void troca(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int main(void) {
    int x = 1, y = 2;
    troca_errada(x, y);
    printf("depois de troca_errada: x=%d y=%d\n", x, y);
    troca(&x, &y);
    printf("depois de troca:        x=%d y=%d\n", x, y);
    return 0;
}
```

Saída

```
depois de troca_errada: x=1 y=2
depois de troca:        x=2 y=1
```

`troca_errada` recebe cópias de `x` e `y`; `troca` recebe os endereços, e o `*` escreve nas variáveis do `main`.

Q8 Laços e o do-while

Escreva um programa que leia números inteiros da entrada até chegar um `0` e mostre quantos foram lidos (sem contar o zero), a soma e a média com uma casa decimal. Use `do-while` e trate o caso de nenhum número antes do zero.

```c
#include <stdio.h>

int main(void) {
    int n, quantos = 0;
    long soma = 0;

    do {
        if (scanf("%d", &n) != 1)  /* entrada acabou ou não é número */
            break;
        if (n != 0) {
            soma += n;
            quantos++;
        }
    } while (n != 0);

    if (quantos == 0) {
        printf("nenhum número lido\n");
        return 0;
    }
    printf("lidos: %d, soma: %ld, média: %.1f\n", quantos, soma, (double) soma / quantos);
    return 0;
}
```

Entrada digitada:

```
8 3 10 4 0
```

Saída

```
lidos: 4, soma: 25, média: 6.2
```

O `scanf` devolve quantos itens converteu: testar `!= 1` evita um laço infinito quando a entrada termina ou tem texto. A soma é `long` e a média converte para `double` antes de dividir.

A média é 6,25 e aparece como `6.2`, não `6.3`. O valor 6,25 é exato em binário, e o `printf` desempata arredondando para o dígito par. Já `6.35` vira `6.3` por outro motivo: em binário ele é 6,3499999..., então nem chega a ser empate.

Q9 Busca do maior e do segundo maior

*Escreva uma função `void dois_maiores(const int v[], int n, int *maior, int *segundo)` que encontra, numa única passada, o maior e o segundo maior valor **distintos** do vetor. Teste com `{5, 9, 2, 9, 7}`.*

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

void dois_maiores(const int v[], int n, int *maior, int *segundo) {
    *maior = *segundo = INT_MIN;
    for (int i = 0; i < n; i++) {
        if (v[i] > *maior) {
            *segundo = *maior;
            *maior = v[i];
        } else if (v[i] < *maior && v[i] > *segundo) {
            *segundo = v[i];  /* "<" e não "!=": repetir o maior não o torna segundo */
        }
    }
}

int main(void) {
    int v[] = {5, 9, 2, 9, 7};
    int m, s;
    dois_maiores(v, sizeof v / sizeof v[0], &m, &s);
    printf("maior: %d, segundo: %d\n", m, s);
    return 0;
}
```

Saída

```
maior: 9, segundo: 7
```

Aqui `sizeof v / sizeof v[0]` funciona porque `v` é o vetor de verdade, no `main`; dentro da função ele já seria só um ponteiro. `INT_MIN` como valor inicial dispensa tratar o primeiro elemento à parte. Se todos os valores forem iguais, `segundo` continua `INT_MIN`, e quem chama precisa tratar esse caso.

MÓDULO 2

Ponteiros e strings

Endereços, aritmética de ponteiros, strings e argumentos de linha de comando.

Artigos 07 a 10

#07 Ponteiros: O que É um Endereço de Memória
#08 Aritmética de Ponteiros e a Ligação Profunda com Vetores
#09 Strings: Vetores de Caracteres e a Biblioteca string.h
#10 Ponteiro para Ponteiro e Argumentos de Linha de Comando

Perguntas conceituais — respostas

Q10 Vetor e ponteiro não são a mesma coisa

> *`int v[5]` e `int *p = v` permitem escrever `v[2]` e `p[2]`. Cite duas diferenças reais entre `v` e `p`.*

- `sizeof v` é o tamanho do vetor inteiro (20 bytes com `int` de 4), e `sizeof p` é o tamanho de um ponteiro.
- `v` não é uma variável que se possa reatribuir: `v = outro` não compila, enquanto `p = outro` é normal.
- `&v` tem tipo "ponteiro para vetor de 5 int" (`int (*)[5]`), e `&p` é "ponteiro para ponteiro" (`int **`).

O que elas têm em comum: na maioria das expressões, `v` decai para `&v[0]`, e `v[i]` é definido como `*(v + i)`.

Q11 Aritmética de ponteiros

> *Se `p` é um `int *` que vale `0x1000` e `int` tem 4 bytes, quanto vale `p + 3`? E se `p` fosse um `char *`? O que significa subtrair dois ponteiros?*

`p + 3` avança 3 **elementos**, não 3 bytes: com `int` de 4 bytes vale `0x100C`; com `char *` valeria `0x1003`. Subtrair dois ponteiros para o mesmo vetor dá a distância em elementos, do tipo `ptrdiff_t`; subtrair ponteiros de vetores diferentes é comportamento indefinido.

Q12 O '\0' e o tamanho da string

> *Qual a diferença entre `sizeof` e `strlen` aplicados a `char s[20] = "casa";`? Por que `char t[4] = "casa";` é perigoso?*

`sizeof s` é 20 (o tamanho do vetor) e `strlen(s)` é 4 (os caracteres antes do `'\0'`). Em `char t[4] = "casa";` o literal tem 5 caracteres contando o `'\0'`; C permite a inicialização, mas descarta o terminador. `t` não é uma string válida, e `strlen(t)` ou `printf("%s", t)` continuam lendo memória até achar um zero por acaso.

Q13 Literal de string

> *Por que `char *s = "texto"; s[0] = 'T';` pode derrubar o programa, enquanto `char s[] = "texto"; s[0] = 'T';` funciona?*

No primeiro caso, `s` aponta para o próprio literal, que costuma ficar numa área somente leitura; modificá-lo é comportamento indefinido e, na prática, dá falha de segmentação. No segundo, `s` é um vetor local **inicializado com uma cópia** do literal, e a cópia é modificável. Por isso o hábito de declarar `const char *s = "texto";`: o compilador passa a recusar a escrita.

Exercícios de código — respostas

Q14 Percorrendo com ponteiro

> *Escreva `size_t meu_strlen(const char *s)` sem usar índices, só ponteiros, e `void inverte(char *s)` que inverte a string no lugar. Teste com "ponteiro".*

```
#include <stdio.h>

size_t meu_strlen(const char *s) {
    const char *p = s;
    while (*p)
        p++;
    return (size_t) (p - s);  /* distância em elementos */
}

void inverte(char *s) {
    char *fim = s + meu_strlen(s);
    while (s < --fim) {
        char t = *s;
        *s++ = *fim;
        *fim = t;
    }
}

int main(void) {
    char palavra[] = "ponteiro";
    printf("%zu\n", meu_strlen(palavra));
    inverte(palavra);
    printf("%s\n", palavra);
    return 0;
}
```

Saída

```
8
orietnop
```

`palavra` é um vetor (cópia modificável do literal); com `char *palavra = "ponteiro"` o `inverte` tentaria escrever num literal. O `%zu` é o formato de `size_t`.

Q15 Contando palavras

Escreva `int conta_palavras(const char *s)` que conta palavras separadas por um ou mais espaços, tabs ou quebras de linha, usando `isspace` de `<ctype.h>`. Teste com `" uma frase\tcom\n cinco palavras "`.

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

int conta_palavras(const char *s) {
    int n = 0, dentro = 0;
    for (; *s; s++) {
        if (isspace((unsigned char) *s)) {
            dentro = 0;
        } else if (!dentro) {
            dentro = 1;  /* início de uma palavra */
            n++;
        }
    }
    return n;
}

int main(void) {
    printf("%d\n", conta_palavras("  uma   frase\tcom\n cinco palavras "));
    printf("%d\n", conta_palavras(""));
    return 0;
}
```

Saída

```
5
0
```

O cast para `unsigned char` não é enfeite: as funções de `<ctype.h>` só aceitam valores de `unsigned char` ou `EOF`, e um `char` com sinal negativo (letras acentuadas em Latin-1, por exemplo) é comportamento indefinido.

Q16 Argumentos de linha de comando

Escreva um programa que receba números pela linha de comando e imprima a soma. Argumentos que não forem números inteiros devem ser avisados em `stderr` e ignorados. Use `strtol`, não `atoi`, e diga por quê.

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

int main(int argc, char *argv[]) {
    long soma = 0;
    for (int i = 1; i < argc; i++) {
        char *fim;
        errno = 0;
        long n = strtol(argv[i], &fim, 10);
        if (fim == argv[i] || *fim != '\0' || errno == ERANGE) {
            fprintf(stderr, "ignorado: %s\n", argv[i]);
            continue;
        }
        soma += n;
    }
    printf("soma: %ld\n", soma);
    return 0;
}
```

Executado sem argumentos, o programa imprime:

Saída

```
soma: 0
```

Com `./soma 10 abc 5 7x 20`, a saída é `ignorado: abc`, `ignorado: 7x` e `soma: 35`. O `atoi` devolve `0` tanto para `"0"` quanto para `"abc"` e não tem como avisar o erro; o `strtol` informa onde parou (`fim`) e sinaliza estouro em `errno`. Testar `*fim != '\0'` recusa `"7x"`, que o `atoi` aceitaria como 7.

MÓDULO 3

Memória: pilha, heap e vazamentos

Onde cada coisa vive, alocação dinâmica e a caça aos vazamentos.

Artigos 11 a 13

#11 A Pilha e o Heap: Onde Cada Coisa Vive
#12 Alocação Dinâmica: malloc, calloc, realloc e free
#13 Vazamentos de Memória e Como Caçá-los

Perguntas conceituais — respostas

Q17 Pilha ou heap

Para cada variável, diga onde ela vive e quando deixa de existir: (a) um `int` local de uma função; (b) uma variável `static` dentro de uma função; (c) o bloco devolvido por `malloc`; (d) uma variável global.

- (a) Na **pilha**: nasce na entrada da função e some no retorno. Devolver o endereço dela é bug.
- (b) Na área de dados estáticos: existe do início ao fim do programa e mantém o valor entre chamadas.
- (c) No **heap**: existe até um `free`, independentemente da função que alocou.
- (d) Também na área estática, durante toda a execução.

Q18 realloc sem perder memória

Por que `p = realloc(p, novo_tamanho);` é um padrão perigoso? Escreva a forma correta.

Se o `realloc` falhar, ele devolve `NULL` **e o bloco antigo continua alocado**. Atribuindo direto a `p`, o único ponteiro para o bloco é sobrescrito: vazamento, e os dados antigos ficam inacessíveis. O certo é usar um temporário:

```
int *tmp = realloc(p, novo * sizeof *tmp);
if (tmp == NULL) {
    /* p ainda é válido: trate o erro e libere quando for o caso */
} else {
    p = tmp;
}
```

Q19 calloc e malloc

Quais as duas diferenças entre `malloc(n * sizeof(int))` e `calloc(n, sizeof(int))`?

O `calloc` **zera** a memória (o `malloc` a entrega com lixo) e recebe quantidade e tamanho separados, o que permite detectar o estouro da multiplicação `n * tamanho` e devolver `NULL` em vez de alocar um bloco menor do que o pedido.

Q20 Dangling pointer

O que é um dangling pointer? Dê dois jeitos de criar um e um hábito que reduz o estrago.

É um ponteiro que aponta para memória que não é mais válida. Dois jeitos: usar um ponteiro depois do `free` do bloco (*use-after-free*) e devolver o endereço de uma variável local. O hábito de fazer `p = NULL;` logo depois do `free(p)` transforma um uso posterior num erro imediato e previsível (desreferenciar `NULL` derruba o programa na hora), em vez de corromper memória silenciosamente. O AddressSanitizer (`-fsanitize=address`) detecta os dois casos.

Exercícios de código — respostas

Q21 Vetor que cresce

Implemente um vetor dinâmico de `int` com `push` que dobra a capacidade quando enche (comece com capacidade 2). Insira os números de 1 a 10 e mostre a capacidade a cada realocação. Libere tudo no fim.

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

typedef struct {
    int *dados;
    size_t tam, cap;
} Vetor;

int push(Vetor *v, int x) {
    if (v->tam == v->cap) {
        size_t nova = v->cap ? v->cap * 2 : 2;
        int *tmp = realloc(v->dados, nova * sizeof *tmp);
        if (tmp == NULL)
            return -1;  /* v continua válido */
        v->dados = tmp;
        v->cap = nova;
        printf("capacidade: %zu\n", v->cap);
    }
    v->dados[v->tam++] = x;
    return 0;
}

int main(void) {
    Vetor v = {0};
    for (int i = 1; i <= 10; i++)
        if (push(&v, i) != 0) {
            fprintf(stderr, "sem memória\n");
            free(v.dados);
            return 1;
        }
    for (size_t i = 0; i < v.tam; i++)
        printf("%d ", v.dados[i]);
    printf("\n");
    free(v.dados);
    return 0;
}
```

Saída

```
capacidade: 2
capacidade: 4
capacidade: 8
capacidade: 16
1 2 3 4 5 6 7 8 9 10
```

Dobrar a capacidade faz o custo médio de cada `push` ser constante: as 10 inserções custaram só 4 realocações. `realloc(NULL, n)` funciona como `malloc(n)`, por isso o vetor começa zerado (`{0}`) e não precisa de caso especial.

Q22 Duplicando uma string

Escreva `char *duplica(const char *s)`, equivalente ao `strdup` (que só entrou no padrão no C23), e um programa que duplica, modifica a cópia e mostra que o original não mudou.

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

char *duplica(const char *s) {
    size_t n = strlen(s) + 1;  /* +1 para o '\0' */
    char *copia = malloc(n);
    if (copia != NULL)
        memcpy(copia, s, n);
    return copia;
}

int main(void) {
    const char *original = "memória";
    char *copia = duplica(original);
    if (copia == NULL)
        return 1;
    copia[0] = 'M';
    printf("%s / %s\n", original, copia);
    free(copia);
    return 0;
}
```

Saída

```
memória / Memória
```

O erro clássico é `malloc(strlen(s))`, sem o `+1`: a cópia do `'\0'` escreve um byte fora do bloco. Quem recebe o ponteiro passa a ser dono da memória e precisa fazer o `free`.

Q23 Achando o vazamento

O programa abaixo vaza memória. Encontre os dois vazamentos, corrija e diga qual ferramenta do curso confirmaria a correção.

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

char *saudacao(const char *nome) {
    char *s = malloc(strlen(nome) + 7);
    strcpy(s, "Olá, ");
    strcat(s, nome);
    return s;
}

int main(void) {
    for (int i = 0; i < 3; i++) {
        char *msg = saudacao("Ana");
        msg = saudacao("Bia");
        free(msg);
    }
    return 0;
}
```

Os dois vazamentos: em cada volta, o resultado de `saudacao("Ana")` é perdido quando `msg` recebe `saudacao("Bia")`; e o tamanho também está errado, porque `"Olá, "` tem 6 bytes em UTF-8 (o `á` ocupa 2) e, somando o `'\0'`, o `+ 7` só cabe por sorte. Corrigido:

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

char *saudacao(const char *nome) {
    const char *prefixo = "Olá, ";
    char *s = malloc(strlen(prefixo) + strlen(nome) + 1);
    if (s == NULL)
        return NULL;
    strcpy(s, prefixo);
    strcat(s, nome);
    return s;
}

int main(void) {
    for (int i = 0; i < 3; i++) {
        char *msg = saudacao("Ana");
        printf("%s\n", msg ? msg : "(sem memória)");
        free(msg);
        msg = saudacao("Bia");
        printf("%s\n", msg ? msg : "(sem memória)");
        free(msg);
    }
    return 0;
}
```

Saída

```
Olá, Ana
Olá, Bia
Olá, Ana
Olá, Bia
Olá, Ana
Olá, Bia
```

Para confirmar: `valgrind --leak-check=full ./prog` deve terminar com "All heap blocks were freed", ou compilar com `-fsanitize=address`, cujo LeakSanitizer reporta os blocos perdidos ao fim da execução. Calcular o tamanho com `strlen` do próprio prefixo evita contar bytes à mão.

MÓDULO 4

Tipos próprios, pré-processador e compilação

Structs, unions, enums, typedef, macros, cabeçalhos e classes de armazenamento.

Artigos 14 a 20

#14 Structs: Criando Seus Próprios Tipos
#15 Ponteiros para Struct e o Operador Seta
#16 Unions e Enums: Economia e Clareza
#17 typedef e o Design de Tipos Legíveis
#18 O Pré-processador: #define, Macros e Inclusões
#19 Compilação Separada: Cabeçalhos e Múltiplos Arquivos
#20 Classes de Armazenamento: static, extern e o Tempo de Vida

Perguntas conceituais — respostas

Q24 Padding em structs

Por que `sizeof` de `struct { char c; int i; char d; }` costuma ser 12 e não 6? Como reordenar os campos para gastar menos?

Por **alinhamento**: o processador acessa um `int` de 4 bytes mais rápido (e algumas arquiteturas só conseguem) num endereço múltiplo de 4. O compilador insere 3 bytes de preenchimento depois de `c` e mais 3 no fim (para que um vetor dessas structs mantenha cada `i` alinhado): 1 + 3 + 4 + 1 + 3 = 12. Colocando os campos do maior para o menor, `struct { int i; char c; char d; }` ocupa 8.

Q25 union

O que é uma `union` e qual o perigo de ler um campo diferente do último escrito? Dê um uso legítimo.

Numa `union` todos os campos ocupam **o mesmo endereço**, e o tamanho é o do maior campo. Só o último campo escrito tem valor significativo; ler outro reinterpreta os bytes (em C isso é permitido para ver a representação, mas o resultado depende da plataforma). O uso legítimo mais comum é a **união etiquetada**: uma `struct` com um `enum` dizendo qual campo da `union` está valendo, como num valor que pode ser número ou texto.

Q26 Macros com efeito colateral

Por que `#define QUADRADO(x) x * x` está errado duas vezes? Corrija e mostre um caso que nem a versão corrigida resolve.

Primeiro, falta parêntese: `QUADRADO(1 + 2)` vira `1 + 2 * 1 + 2`, que dá 5. Segundo, `QUADRADO(a) / QUADRADO(b)` também se desarranja sem o parêntese externo. A versão correta é `#define QUADRADO(x) ((x) * (x))`. Mesmo assim, `QUADRADO(i++)` incrementa `i` duas vezes (e sem ponto de sequência entre elas: comportamento indefinido), porque a macro copia o texto do argumento. Uma função `static inline` não tem esse problema.

Q27 static e extern

Explique os dois significados de `static`: numa variável dentro de uma função e numa função ou variável no escopo do arquivo. E para que serve `extern`?

Dentro de uma função, `static` muda a **duração**: a variável existe durante todo o programa e guarda o valor entre chamadas. No escopo do arquivo, muda a **ligação**: a função ou variável fica visível só naquele `.c` (ligação interna), o que evita colisão de nomes entre arquivos. `extern` declara que uma variável global **existe em outro lugar**; aparece no cabeçalho (`extern int contador;`), enquanto um único `.c` faz a definição (`int contador = 0;`).

Q28 Include guard

Para que serve o trecho `#ifndef PILHA_H / #define PILHA_H / ... / #endif` num cabeçalho? O que acontece sem ele?

Impede que o conteúdo do cabeçalho seja processado duas vezes no mesmo arquivo `.c`, o que acontece quando dois cabeçalhos incluem um terceiro. Sem a guarda, definições como a de uma `struct` ou de um `typedef` aparecem duplicadas e a compilação falha com "redefinition". `#pragma once` faz o mesmo e é aceito por todos os compiladores populares, mas não é padrão.

Exercícios de código — respostas

Q29 Ponto e retângulo

Defina `typedef struct { double x, y; } Ponto;` e `typedef struct { Ponto canto; double largura, altura; } Retangulo;`. Escreva `int contem(const Retangulo *r, Ponto p)` e teste três pontos, usando o operador seta.

```
#include <stdio.h>

typedef struct { double x, y; } Ponto;
typedef struct { Ponto canto; double largura, altura; } Retangulo;

int contem(const Retangulo *r, Ponto p) {
    return p.x >= r->canto.x && p.x <= r->canto.x + r->largura
        && p.y >= r->canto.y && p.y <= r->canto.y + r->altura;
}

int main(void) {
    Retangulo r = { .canto = {0, 0}, .largura = 4, .altura = 2 };
    Ponto testes[] = { {1, 1}, {4, 2}, {5, 1} };
    for (int i = 0; i < 3; i++)
        printf("(%.0f, %.0f): %s\n", testes[i].x, testes[i].y,
               contem(&r, testes[i]) ? "dentro" : "fora");
    return 0;
}
```

Saída

```
(1, 1): dentro
(4, 2): dentro
(5, 1): fora
```

A função recebe o retângulo por ponteiro (`const`, porque só lê) para não copiar a struct a cada chamada, e `r->canto.x` é o mesmo que `(*r).canto.x`. Os inicializadores designados (`.largura = 4`) deixam claro qual campo recebe qual valor.

Q30 União etiquetada

> *Escreva um tipo `Valor` que guarda um inteiro **ou** um texto (até 31 caracteres), com um `enum` dizendo qual. Escreva `void imprime(const Valor *v)` e teste os dois casos. Mostre também o `sizeof` do tipo.*

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

typedef enum { INTEIRO, TEXTO } Tipo;

typedef struct {
    Tipo tipo;
    union {
        long inteiro;
        char texto[32];
    };  /* union anônima (C11): acesso direto por v.inteiro */
} Valor;

void imprime(const Valor *v) {
    switch (v->tipo) {
    case INTEIRO: printf("inteiro: %ld\n", v->inteiro); break;
    case TEXTO:   printf("texto: %s\n", v->texto);      break;
    }
}

int main(void) {
    Valor a = { .tipo = INTEIRO, .inteiro = 42 };
    Valor b = { .tipo = TEXTO };
    strncpy(b.texto, "quarenta e dois", sizeof b.texto - 1);
    imprime(&a);
    imprime(&b);
    printf("sizeof(Valor) = %zu\n", sizeof(Valor));
    return 0;
}
```

Saída

```
inteiro: 42
texto: quarenta e dois
sizeof(Valor) = 40
```

O tamanho é o do `enum` (4 bytes) mais o maior campo da union (32), já alinhado: 36 não seria múltiplo de 8 (o alinhamento do `long`), então o compilador arredonda para 40. Sem o `switch` cobrindo todos os casos do `enum`, o `-Wall` avisa (`-Wswitch`).

Q31 Contador com static

> *Escreva uma função `int proximo_id(void)` que devolve 1, 2, 3... a cada chamada, sem variável global. Depois explique por que ela não serve, como está, para um programa com várias threads.*

```
#include <stdio.h>

int proximo_id(void) {
    static int ultimo = 0;  /* inicializado uma única vez, vive até o fim do programa */
    return ++ultimo;
}

int main(void) {
    for (int i = 0; i < 3; i++)
        printf("id %d\n", proximo_id());
    return 0;
}
```

Saída

```
id 1
id 2
id 3
```

Com threads, duas chamadas simultâneas podem ler o mesmo `ultimo` antes de qualquer uma gravar e devolver o mesmo id: é uma condição de corrida. A correção é proteger o incremento com um mutex ou usar `atomic_int` de `<stdatomic.h>` com `atomic_fetch_add`.

Q32 Macro de depuração

> *Escreva uma macro `LOG(fmt, ...)` que imprime em `stderr` o arquivo, a linha e a mensagem formatada, e que some completamente quando o programa é compilado com `-DNDEBUG`.*

```c
#include <stdio.h>

#ifdef NDEBUG
#define LOG(...) ((void) 0)
#else
#define LOG(fmt, ...) fprintf(stderr, "[%s:%d] " fmt "\n", __FILE__, __LINE__ __VA_OPT__(,)
__VA_ARGS__)
#endif

int main(void) {
    int x = 42;
    LOG("começou");
    LOG("x vale %d", x);
    return 0;
}
```

Saída

```
[main.c:11] começou
[main.c:12] x vale 42
```

`__FILE__` e `__LINE__` são preenchidos pelo pré-processador no ponto de uso, por isso precisa ser macro e não função. O `__VA_OPT__(,)` (padronizado no C23, aceito por gcc e clang antes disso) só coloca a vírgula quando há argumentos, o que permite `LOG("começou")`. O `((void) 0)` mantém `LOG(...);` válido como comando quando a macro é desligada.

MÓDULO 5

Entrada, saída e biblioteca padrão

printf e scanf a fundo, arquivos de texto e binários, stdlib, math, ctype, time e strings.

Artigos 21 a 25

#21 Entrada e Saída Formatada: printf e scanf a Fundo
#22 Lendo e Gravando Arquivos de Texto
#23 Arquivos Binários e Posicionamento no Disco
#24 Explorando a Biblioteca Padrão: stdlib, math, ctype, time
#25 Manipulação de Strings sem Sofrimento

Perguntas conceituais — respostas

Q33 scanf e o '\n' que sobra

Por que, depois de `scanf("%d", &idade);`, um `fgets(nome, sizeof nome, stdin);` parece "pular" a leitura? Como resolver?

O `scanf("%d")` lê o número e para **antes** do `'\n'` que o usuário digitou com o Enter. O `fgets` seguinte encontra esse `'\n'` no buffer e termina na hora, com uma linha vazia. Soluções: ler tudo com `fgets` e converter com `sscanf` / `strtol` (o caminho mais robusto), ou descartar o resto da linha depois do `scanf` com `int c; while ((c = getchar()) != '\n' && c != EOF);`.

Q34 Modo texto e binário

Qual a diferença entre abrir um arquivo com `"r"` e com `"rb"`? Em que sistema ela aparece de verdade?

Em modo texto a biblioteca pode **traduzir** o fim de linha: no Windows, `"\r\n"` vira `'\n'` na leitura e o contrário na escrita, e um byte `0x1A` pode ser tratado como fim de arquivo. Em modo binário os bytes passam sem alteração. No Linux e no macOS os dois modos são idênticos; no Windows, ler um arquivo binário (imagem, struct gravada com `fwrite`) em modo texto corrompe os dados.

Q35 feof no lugar errado

Por que o laço `while (!feof(f)) { fgets(linha, sizeof linha, f); ... }` costuma processar a última linha duas vezes?

Porque `feof` só fica verdadeiro **depois** que uma leitura tenta passar do fim. Após ler a última linha, `feof` ainda é falso; o laço roda de novo, o `fgets` falha (devolve `NULL` e não altera `linha`) e o corpo processa o conteúdo antigo outra vez. O certo é testar o retorno da própria leitura: `while (fgets(linha, sizeof linha, f) != NULL) { ... }`.

Q36 rand e a semente

O que faz `srand(time(NULL))`? Por que chamá-lo dentro de um laço, antes de cada `rand()`, é um erro?

Define a semente do gerador pseudoaleatório a partir do relógio, para que cada execução produza uma sequência diferente. Chamado dentro do laço, ele reinicia o gerador várias vezes no mesmo segundo (`time` tem resolução de segundos) com a mesma semente, e `rand()` devolve o mesmo número repetidamente. A semente deve ser definida uma vez, no início do programa. E `rand()` não serve para nada de segurança, como senhas ou tokens.

Exercícios de código — respostas

Q37 Gravando e lendo texto

Escreva um programa que grava três linhas `nome;nota` num arquivo `notas.txt`, fecha, reabre para leitura e imprime a média, tratando a falha de `fopen`.

```
#include <stdio.h>

int main(void) {
    FILE *f = fopen("notas.txt", "w");
    if (f == NULL) {
        perror("notas.txt");
        return 1;
    }
    fprintf(f, "Ana;9.5\nBruno;7.0\nCarla;8.0\n");
    fclose(f);

    f = fopen("notas.txt", "r");
    if (f == NULL) {
        perror("notas.txt");
        return 1;
    }
    char nome[64];
    double nota, soma = 0;
    int n = 0;
    while (fscanf(f, "%63[^;];%lf\n", nome, &nota) == 2) {
        printf("%-6s %4.1f\n", nome, nota);
        soma += nota;
        n++;
    }
    fclose(f);
    if (n > 0)
        printf("média: %.2f\n", soma / n);
    return 0;
}
```

Saída

```
Ana     9.5
Bruno   7.0
Carla   8.0
média: 8.17
```

`%63[^;]` lê até 63 caracteres que não sejam `;`: o limite impede estourar `nome[64]`, e o conjunto `[^;]` é o separador. O laço para quando `fscanf` não converte os dois campos, o que inclui o fim do arquivo. `perror` mostra o motivo da falha (`No such file or directory`, `Permission denied`).

Q38 Arquivo binário com fseek

*Grave 5 structs `{int id; double preco;}` com `fwrite`, depois leia **só a quarta** usando `fseek`, sem ler as anteriores. Mostre também o tamanho do arquivo com `ftell`.*

```c
#include <stdio.h>

typedef struct { int id; double preco; } Produto;

int main(void) {
    FILE *f = fopen("produtos.bin", "w+b");
    if (f == NULL) {
        perror("produtos.bin");
        return 1;
    }
    for (int i = 1; i <= 5; i++) {
        Produto p = { i, i * 10.5 };
        fwrite(&p, sizeof p, 1, f);
    }

    fseek(f, 0, SEEK_END);
    printf("tamanho: %ld bytes (%zu por registro)\n", ftell(f), sizeof(Produto));

    Produto quarto;
    fseek(f, 3 * (long) sizeof(Produto), SEEK_SET);  /* pula 3 registros */
    if (fread(&quarto, sizeof quarto, 1, f) == 1)
        printf("quarto: id=%d preço=%.2f\n", quarto.id, quarto.preco);
    fclose(f);
    return 0;
}
```

Saída

```
tamanho: 80 bytes (16 por registro)
quarto: id=4 preço=42.00
```

Cada registro ocupa 16 bytes, não 12, por causa do alinhamento do `double`; por isso o deslocamento usa `sizeof(Produto)`. Um arquivo assim só é lido corretamente por um programa compilado com o mesmo layout de struct e a mesma ordem de bytes: para trocar dados entre máquinas, use um formato de texto ou serialize campo a campo.

Q39 Estatísticas com math.h

Escreva um programa que calcula média, desvio padrão (populacional) e mediana do vetor `{4, 8, 15, 16, 23, 42}`, usando `qsort` para a mediana e `sqrt` de `<math.h>`.

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

static int compara(const void *a, const void *b) {
    double x = *(const double *) a, y = *(const double *) b;
    return (x > y) - (x < y);  /* -1, 0 ou 1, sem o risco de subtrair */
}

int main(void) {
    double v[] = {4, 8, 15, 16, 23, 42};
    size_t n = sizeof v / sizeof v[0];

    double soma = 0;
    for (size_t i = 0; i < n; i++)
        soma += v[i];
    double media = soma / n;

    double quad = 0;
    for (size_t i = 0; i < n; i++)
        quad += (v[i] - media) * (v[i] - media);
    double desvio = sqrt(quad / n);

    qsort(v, n, sizeof v[0], compara);
    double mediana = n % 2 ? v[n / 2] : (v[n / 2 - 1] + v[n / 2]) / 2;

    printf("média %.2f, desvio %.2f, mediana %.2f\n", media, desvio, mediana);
    return 0;
}
```

Saída

```
média 18.00, desvio 12.32, mediana 15.50
```

No Linux, as funções de `<math.h>` ficam na `libm`, por isso o `-lm` no fim da linha do gcc. A comparação `(x > y) - (x < y)` evita o `return x - y`, que com `double` seria convertido para `int` e daria resultados errados para diferenças menores que 1.

Q40 Texto em maiúsculas e sem espaços duplicados

Escreva `void normaliza(char *s)` que, no próprio vetor, converte para maiúsculas, remove espaços do início e do fim e reduz espaços repetidos a um só. Teste com `" olá mundo em C "` e explique o que acontece com o `á`.

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

void normaliza(char *s) {
    char *escrita = s;
    int espaco = 0;
    for (char *leitura = s; *leitura; leitura++) {
        unsigned char c = (unsigned char) *leitura;
        if (isspace(c)) {
            espaco = (escrita != s);  /* ignora espaços antes da primeira palavra */
            continue;
        }
        if (espaco) {
            *escrita++ = ' ';
            espaco = 0;
        }
        *escrita++ = (char) toupper(c);
    }
    *escrita = '\0';  /* espaços do fim nunca são escritos */
}

int main(void) {
    char s[] = "   olá   mundo  em  C   ";
    normaliza(s);
    printf("[%s]\n", s);
    return 0;
}
```

Saída

```
[OLá MUNDO EM C]
```

A técnica de dois ponteiros (um lê, outro escreve) faz tudo numa passada, sem vetor auxiliar. O `á` não vira `Á`: em UTF-8 ele ocupa dois bytes, e `toupper` trabalha byte a byte no locale "C", que só conhece ASCII. Converter acentos exige tratar o texto como caracteres largos (`wchar_t`, `towupper`) com um locale UTF-8 configurado por `setlocale`.

MÓDULO 6

Ponteiros para função, recursão e bits

Comportamento como argumento, qsort e bsearch, recursão, máscaras e campos de bits.

Artigos 26 a 30

#26 Ponteiros para Função: Passando Comportamento como Argumento
#27 qsort e bsearch: Ordenando e Buscando com a Biblioteca
#28 Recursão: Quando uma Função Chama a Si Mesma
#29 Manipulação de Bits e Máscaras
#30 Campos de Bits e Estruturas Compactas

Perguntas conceituais — respostas

Q41 Lendo declarações

Explique em palavras o que declara cada linha: (a) `int (*f)(int, int);` (b) `int *g(int, int);` (c) `void (*tabela[3])(void);`

- (a) `f` é um **ponteiro para função** que recebe dois `int` e devolve `int`.
- (b) `g` é uma **função** que recebe dois `int` e devolve um ponteiro para `int`.
- (c) `tabela` é um vetor de 3 ponteiros para funções sem parâmetros e sem retorno: uma tabela de despacho.

Os parênteses em (a) mudam tudo: sem eles, o `*` se prende ao tipo de retorno, como em (b). Um `typedef int (*Operacao)(int, int);` deixa esse tipo de código muito mais legível.

Q42 Recursão e a pilha

O que é estouro de pilha (stack overflow) numa função recursiva? O que é recursão de cauda e por que o padrão C não garante que ela economize pilha?

Cada chamada guarda parâmetros, variáveis locais e o endereço de retorno num quadro da pilha; uma recursão profunda demais (ou sem caso base) esgota a pilha, de alguns megabytes, e o sistema derruba o programa. Na recursão de cauda, a chamada recursiva é a **última** coisa que a função faz, e o compilador **pode** reaproveitar o quadro atual, transformando a recursão num laço. O padrão C não obriga essa otimização: o gcc e o clang a fazem com `-O2`, mas não sem otimização. Código que depende dela para não estourar a pilha deve virar laço explícito.

Q43 Operações com bits

Escreva as expressões para, num `unsigned` chamado `flags`: ligar o bit 3, desligar o bit 5, inverter o bit 0 e testar se o bit 7 está ligado. Por que usar tipos sem sinal para isso?

- Ligar o bit 3: `flags |= 1u << 3;`
- Desligar o bit 5: `flags &= ~(1u << 5);`
- Inverter o bit 0: `flags ^= 1u << 0;`

- Testar o bit 7: `if (flags & (1u << 7))`

Com tipos sem sinal, os deslocamentos e o `~` têm comportamento totalmente definido. Com sinal, deslocar um 1 para o bit de sinal (`1 << 31` num `int` de 32 bits) é comportamento indefinido, e deslocar negativos para a direita depende da implementação. Por isso o sufixo `u` em `1u`.

Exercícios de código — respostas

Q44 Ordenando structs com qsort

> *Ordene um vetor de alunos `{nome, nota}` por nota decrescente e, em caso de empate, por nome em ordem alfabética. Depois use `bsearch` num vetor de inteiros ordenado para achar o 23.*

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

typedef struct { const char *nome; double nota; } Aluno;

static int por_nota(const void *a, const void *b) {
    const Aluno *x = a, *y = b;
    if (x->nota != y->nota)
        return (y->nota > x->nota) - (y->nota < x->nota);  /* decrescente */
    return strcmp(x->nome, y->nome);
}

static int por_int(const void *a, const void *b) {
    int x = *(const int *) a, y = *(const int *) b;
    return (x > y) - (x < y);
}

int main(void) {
    Aluno turma[] = { {"Davi", 8.5}, {"Ana", 9.0}, {"Caio", 8.5}, {"Bia", 7.0} };
    size_t n = sizeof turma / sizeof turma[0];
    qsort(turma, n, sizeof turma[0], por_nota);
    for (size_t i = 0; i < n; i++)
        printf("%-5s %.1f\n", turma[i].nome, turma[i].nota);

    int v[] = {4, 8, 15, 16, 23, 42};
    int chave = 23;
    int *achou = bsearch(&chave, v, 6, sizeof v[0], por_int);
    if (achou)
        printf("23 está no índice %td\n", achou - v);
    return 0;
}
```

Saída

```
Ana   9.0
Caio  8.5
Davi  8.5
Bia   7.0
23 está no índice 4
```

O `qsort` não sabe nada sobre `Aluno`: recebe a função de comparação como argumento e a chama com ponteiros genéricos (`const void *`). O `bsearch` só funciona num vetor já ordenado pelo mesmo critério da comparação; `%td` é o formato de `ptrdiff_t`, o tipo da subtração de ponteiros.

Q45 Calculadora com tabela de funções

> *Escreva uma tabela que associa os caracteres `+ - * /` a funções `double f(double, double)` e um laço que aplica cada operação a 12 e 4, procurando a função pelo símbolo.*

```c
#include <stdio.h>

typedef double (*Operacao)(double, double);

static double soma(double a, double b) { return a + b; }
static double sub(double a, double b)  { return a - b; }
static double mul(double a, double b)  { return a * b; }
static double divi(double a, double b) { return a / b; }

static const struct { char simbolo; Operacao f; } tabela[] = {
    {'+', soma}, {'-', sub}, {'*', mul}, {'/', divi},
};

static Operacao busca(char simbolo) {
    for (size_t i = 0; i < sizeof tabela / sizeof tabela[0]; i++)
        if (tabela[i].simbolo == simbolo)
            return tabela[i].f;
    return NULL;
}

int main(void) {
    const char *ops = "+-*/%";
    for (const char *p = ops; *p; p++) {
        Operacao f = busca(*p);
        if (f)
            printf("12 %c 4 = %g\n", *p, f(12, 4));
        else
            printf("operação %c desconhecida\n", *p);
    }
    return 0;
}
```

Saída

```
12 + 4 = 16
12 - 4 = 8
12 * 4 = 48
12 / 4 = 3
operação % desconhecida
```

Acrescentar uma operação é acrescentar uma linha na tabela, sem mexer na lógica de busca nem num `switch`. O `typedef` dá nome ao tipo "ponteiro para função de dois `double`", que de outro modo apareceria escrito por extenso em três lugares.

Q46 Torres de Hanói

> *Escreva a solução recursiva das Torres de Hanói que imprime os movimentos para 3 discos e conta quantos foram. Quantos movimentos são necessários para n discos?*

```c
#include <stdio.h>

static int movimentos = 0;

void hanoi(int n, char origem, char destino, char auxiliar) {
    if (n == 0)
        return;  /* caso base */
    hanoi(n - 1, origem, auxiliar, destino);
    printf("disco %d: %c -> %c\n", n, origem, destino);
    movimentos++;
    hanoi(n - 1, auxiliar, destino, origem);
}

int main(void) {
    hanoi(3, 'A', 'C', 'B');
    printf("total: %d movimentos\n", movimentos);
    return 0;
}
```

Saída

```
disco 1: A -> C
disco 2: A -> B
disco 1: C -> B
disco 3: A -> C
disco 1: B -> A
disco 2: B -> C
disco 1: A -> C
total: 7 movimentos
```

Para mover n discos é preciso mover n−1 para o pino auxiliar, mover o maior e trazer os n−1 de volta: M(n) = 2·M(n−1) + 1, o que dá **$2^n - 1$** movimentos. Com 64 discos, seriam mais de 18 quintilhões.

Q47 Contando bits e testando potência de 2

Escreva `int conta_bits(unsigned x)` com o truque `x &= x - 1` e `int potencia_de_2(unsigned x)`. Mostre os resultados para 0, 1, 12, 64 e 255, e explique o truque.

```
#include <stdio.h>

int conta_bits(unsigned x) {
    int n = 0;
    while (x) {
        x &= x - 1;  /* apaga o bit 1 mais baixo */
        n++;
    }
    return n;
}

int potencia_de_2(unsigned x) {
    return x != 0 && (x & (x - 1)) == 0;
}

int main(void) {
    unsigned testes[] = {0, 1, 12, 64, 255};
    for (int i = 0; i < 5; i++)
        printf("%3u: %d bits ligados, %s\n", testes[i], conta_bits(testes[i]),
               potencia_de_2(testes[i]) ? "potência de 2" : "não é potência de 2");
    return 0;
}
```

Saída

```
  0: 0 bits ligados, não é potência de 2
  1: 1 bits ligados, potência de 2
 12: 2 bits ligados, não é potência de 2
 64: 1 bits ligados, potência de 2
255: 8 bits ligados, não é potência de 2
```

Subtrair 1 troca o bit 1 mais baixo por 0 e todos os zeros abaixo dele por 1; o `&` com o valor original apaga exatamente esse bit. O laço roda uma vez por bit ligado, não por bit do tipo. Uma potência de 2 tem um único bit ligado, então `x & (x - 1)` dá zero. O gcc e o clang oferecem `__builtin_popcount`, que vira uma única instrução nos processadores modernos.

MÓDULO 7

Estruturas de dados

Listas simples, duplas e circulares, pilhas, filas, árvores de busca e tabelas hash.

Artigos 31 a 35

#31 Listas Encadeadas: A Estrutura que Só Existe com Ponteiros
#32 Listas Duplamente Encadeadas e Circulares
#33 Pilhas e Filas na Prática
#34 Árvores Binárias de Busca
#35 Tabelas Hash: Busca em Tempo Constante

Perguntas conceituais — respostas

Q48 Lista ou vetor

> *Compare lista encadeada e vetor dinâmico em: acesso ao i-ésimo elemento, inserção no início, uso de memória e aproveitamento do cache do processador. Quando a lista ganha?*

- **Acesso por índice**: vetor O(1); lista O(n), porque precisa percorrer os nós.
- **Inserção no início**: lista O(1) (só ajusta ponteiros); vetor O(n), porque desloca todos os elementos.
- **Memória**: a lista gasta um ou dois ponteiros por nó, além do custo de cada `malloc`.
- **Cache**: o vetor é contíguo e muito mais rápido de percorrer; os nós da lista ficam espalhados pelo heap.

A lista ganha quando há muitas inserções e remoções no meio **a partir de um nó que já se tem em mãos**, ou quando os elementos não podem mudar de endereço. Na maioria dos outros casos, o vetor é mais rápido mesmo nas operações em que a teoria favorece a lista.

Q49 Árvore desbalanceada

> *Em que ordem de inserção uma árvore binária de busca vira uma "lista"? Qual o custo da busca nesse caso e que tipos de árvore evitam o problema?*

Inserindo os valores **já ordenados** (1, 2, 3, 4…) ou em ordem decrescente: cada novo nó vira filho direito (ou esquerdo) do anterior, e a altura passa a ser n. A busca cai de O(log n) para O(n). Árvores autobalanceadas, como a AVL e a rubro-negra, fazem rotações a cada inserção e remoção para manter a altura em O(log n).

Q50 Colisões numa tabela hash

> *O que é uma colisão numa tabela hash? Explique o encadeamento separado e diga o que é o fator de carga e por que ele importa.*

Colisão é quando duas chaves diferentes caem no mesmo índice da tabela. No **encadeamento separado**, cada posição guarda uma lista com todas as chaves daquele índice. O **fator de carga** é o número de elementos dividido pelo número de posições: quanto maior, mais longas as listas e mais lenta a busca. Por

isso as implementações redimensionam a tabela (dobram o tamanho e reinserem tudo) quando o fator passa de um limite, em torno de 0,75.

Exercícios de código — respostas

Q51 Lista encadeada com inversão

Implemente uma lista simplesmente encadeada de `int` com `insere_inicio`, `imprime`, `inverte` (no lugar, sem alocar nós) e `libera`. Teste inserindo 1, 2, 3 e 4.

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

typedef struct No {
    int valor;
    struct No *prox;
} No;

int insere_inicio(No **lista, int valor) {
    No *novo = malloc(sizeof *novo);
    if (novo == NULL)
        return -1;
    novo->valor = valor;
    novo->prox = *lista;
    *lista = novo;  /* ponteiro para ponteiro: muda a cabeça da lista de quem chamou */
    return 0;
}

void imprime(const No *p) {
    for (; p; p = p->prox)
        printf("%d -> ", p->valor);
    printf("NULL\n");
}

No *inverte(No *p) {
    No *anterior = NULL;
    while (p) {
        No *prox = p->prox;
        p->prox = anterior;
        anterior = p;
        p = prox;
    }
    return anterior;
}

void libera(No *p) {
    while (p) {
        No *prox = p->prox;  /* guarda antes do free */
        free(p);
        p = prox;
    }
}

int main(void) {
    No *lista = NULL;
    for (int i = 1; i <= 4; i++)
        insere_inicio(&lista, i);
    imprime(lista);
    lista = inverte(lista);
    imprime(lista);
    libera(lista);
    return 0;
}
```

Saída

```
4 -> 3 -> 2 -> 1 -> NULL
1 -> 2 -> 3 -> 4 -> NULL
```

`insere_inicio` recebe `No **` porque precisa alterar o ponteiro da cabeça que pertence ao `main`. Em `libera`, ler `p->prox` depois do `free(p)` seria *use-after-free*; por isso o próximo é guardado antes.

Q52 Verificando expressões com uma pilha

Usando uma pilha (vetor e índice de topo), escreva `int balanceado(const char *s)` que verifica se `()`, `[]` e `{}` estão corretamente aninhados. Teste com quatro expressões.

```c
#include <stdio.h>

int balanceado(const char *s) {
    char pilha[256];
    int topo = 0;
    for (; *s; s++) {
        switch (*s) {
        case '(': case '[': case '{':
            if (topo == (int) sizeof pilha)
                return 0;  /* aninhamento fundo demais para a pilha */
            pilha[topo++] = *s;
            break;
        case ')': case ']': case '}': {
            char esperado = *s == ')' ? '(' : *s == ']' ? '[' : '{';
            if (topo == 0 || pilha[--topo] != esperado)
                return 0;
            break;
        }
        }
    }
    return topo == 0;  /* sobrou abertura sem fechamento? */
}

int main(void) {
    const char *testes[] = { "{a[b(c)d]e}", "(a[b)c]", "((x)", "x + y" };
    for (int i = 0; i < 4; i++)
        printf("%-16s %s\n", testes[i], balanceado(testes[i]) ? "ok" : "desbalanceado");
    return 0;
}
```

Saída

```
{a[b(c)d]e}      ok
(a[b)c]          desbalanceado
((x)             desbalanceado
x + y            ok
```

Cada fechamento precisa casar com a abertura **mais recente**, que é exatamente o topo da pilha. Os três casos de falha são: fechar com a pilha vazia, fechar com o tipo errado e terminar com aberturas pendentes.

Q53 Árvore binária de busca

Implemente uma BST de inteiros com `insere`, `busca`, percurso em ordem e `altura`. Insira 50, 30, 70, 20, 40, 60, 80 e mostre a altura; depois mostre a altura inserindo de 1 a 7 em ordem.

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

typedef struct No { int v; struct No *esq, *dir; } No;

No *insere(No *r, int v) {
    if (r == NULL) {
        No *n = calloc(1, sizeof *n);
        if (n)
            n->v = v;
        return n;
    }
    if (v < r->v)
        r->esq = insere(r->esq, v);
    else if (v > r->v)
        r->dir = insere(r->dir, v);
    return r;  /* valor repetido: ignorado */
}

int busca(const No *r, int v) {
    while (r && r->v != v)
        r = v < r->v ? r->esq : r->dir;
    return r != NULL;
}

void em_ordem(const No *r) {
    if (!r) return;
    em_ordem(r->esq);
    printf("%d ", r->v);
    em_ordem(r->dir);
}

int altura(const No *r) {
    if (!r) return 0;
    int e = altura(r->esq), d = altura(r->dir);
    return 1 + (e > d ? e : d);
}

void libera(No *r) {
    if (!r) return;
    libera(r->esq);
    libera(r->dir);
    free(r);
}

int main(void) {
    int valores[] = {50, 30, 70, 20, 40, 60, 80};
    No *a = NULL, *b = NULL;
    for (int i = 0; i < 7; i++) {
        a = insere(a, valores[i]);
        b = insere(b, i + 1);
    }
    em_ordem(a);
```

```
    printf("\nbusca 60: %d, busca 65: %d\n", busca(a, 60), busca(a, 65));
    printf("altura (ordem misturada): %d\n", altura(a));
    printf("altura (1 a 7 em ordem):  %d\n", altura(b));
    libera(a);
    libera(b);
    return 0;
}
```

Saída

```
20 30 40 50 60 70 80
busca 60: 1, busca 65: 0
altura (ordem misturada): 3
altura (1 a 7 em ordem):  7
```

O percurso em ordem sai ordenado pela própria definição da BST. Com os mesmos 7 valores, a árvore pode ter altura 3 (balanceada) ou 7 (uma lista), e a busca custa proporcionalmente à altura.

Q54 Tabela hash de contagem de palavras

Implemente uma tabela hash com encadeamento (31 posições, função djb2) que conta quantas vezes cada palavra aparece em "o rato roeu a roupa do rei de roma e o rei riu do rato". Imprima as palavras que aparecem mais de uma vez.

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

#define TAM 31

typedef struct Entrada {
    char *palavra;
    int conta;
    struct Entrada *prox;
} Entrada;

static unsigned long djb2(const char *s) {
    unsigned long h = 5381;
    for (; *s; s++)
        h = h * 33 + (unsigned char) *s;
    return h;
}

static int conta(Entrada *tabela[], const char *palavra) {
    unsigned long i = djb2(palavra) % TAM;
    for (Entrada *e = tabela[i]; e; e = e->prox)
        if (strcmp(e->palavra, palavra) == 0)
            return ++e->conta;
    Entrada *nova = malloc(sizeof *nova);
    if (!nova || !(nova->palavra = malloc(strlen(palavra) + 1))) {
        free(nova);
        return -1;
    }
    strcpy(nova->palavra, palavra);
    nova->conta = 1;
    nova->prox = tabela[i];
    tabela[i] = nova;
    return 1;
}

int main(void) {
    char texto[] = "o rato roeu a roupa do rei de roma e o rei riu do rato";
    Entrada *tabela[TAM] = {0};
    for (char *p = strtok(texto, " "); p; p = strtok(NULL, " "))
        if (conta(tabela, p) < 0)
            return 1;

    for (int i = 0; i < TAM; i++)
        for (Entrada *e = tabela[i]; e; e = e->prox)
            if (e->conta > 1)
                printf("%s: %d\n", e->palavra, e->conta);

    for (int i = 0; i < TAM; i++)
        for (Entrada *e = tabela[i], *prox; e; e = prox) {
            prox = e->prox;
            free(e->palavra);
            free(e);
```

```
        }
    return 0;
}
```

Saída

```
rei: 2
rato: 2
do: 2
o: 2
```

A ordem da saída é a ordem dos índices da tabela, não a alfabética nem a de aparição: uma tabela hash não guarda ordem. O `strtok` modifica a string (troca os espaços por `'\0'`), por isso `texto` é um vetor e não um ponteiro para literal; ele também não é seguro com threads (há `strtok_r` no POSIX).

MÓDULO 8

Ferramentas, qualidade e comportamento indefinido

GDB, Valgrind, sanitizers, Make, CMake, UB, errno, testes, código limpo e desempenho.

Artigos 36 a 46

#36 Depurando de Verdade com o GDB
#37 Caçando Bugs de Memória com Valgrind e Sanitizers
#38 Automação de Build com Make
#39 Projetos Maiores com CMake
#40 Undefined Behavior: O Território Perigoso do C
#41 Tratamento de Erros Idiomático: errno e Códigos de Retorno
#42 Testes Automatizados em C
#43 Código Limpo em C: Nomes, Funções e Legibilidade
#44 Programação Defensiva e Contratos
#45 Organização de Projetos e Convenções da Comunidade
#46 Otimização Consciente e Noções de Desempenho

Perguntas conceituais — respostas

Q55 Comportamento indefinido

O que é comportamento indefinido (UB)? Por que "no meu computador funciona" não prova nada? Cite três exemplos.

É uma construção para a qual o padrão **não impõe nenhum requisito**: o programa pode funcionar, travar, dar resultado errado ou mudar de comportamento com outro compilador, outra flag de otimização ou outra versão. O otimizador parte do princípio de que UB nunca acontece e pode remover código com base nisso, então o mesmo programa que "funciona" com `-O0` quebra com `-O2`. Exemplos: estouro de `int` com sinal, acesso fora dos limites de um vetor, desreferenciar `NULL` ou um ponteiro liberado, ler variável não inicializada, modificar a mesma variável duas vezes sem ponto de sequência (`i = i++`).

Q56 Sanitizers ou Valgrind

Qual a diferença de funcionamento entre o AddressSanitizer e o Valgrind? Cite uma vantagem de cada.

O **AddressSanitizer** é ativado na compilação (`-fsanitize=address`): o compilador instrumenta cada acesso à memória. É rápido (o programa fica cerca de 2 vezes mais lento) e pega estouros na pilha e em variáveis globais. O **Valgrind** executa o binário já pronto numa máquina virtual que observa cada instrução: não exige recompilar e detecta leitura de memória não inicializada (no sanitizer isso é o MemorySanitizer, só no clang), mas deixa o programa de 20 a 50 vezes mais lento. Os dois não devem ser usados juntos no mesmo binário.

Q57 O que o make sabe

Como o `make` decide o que recompilar? O que acontece se o `Makefile` não declarar que `main.o` depende de `pilha.h` ?

Pela **data de modificação**: um alvo é refeito se não existir ou se algum pré-requisito listado for mais novo que ele. Se `pilha.h` não estiver entre os pré-requisitos de `main.o`, mudar o cabeçalho não recompila `main.c`, e o executável pode misturar um `main.o` com o layout antigo de uma struct e um `pilha.o` com o novo: um bug difícil de entender. As flags `-MMD -MP` fazem o compilador gerar essas dependências automaticamente.

Q58 **errno**

> *Como funciona o `errno`? Por que é errado testar `errno` sem antes verificar o valor de retorno da função?*

`errno` é um valor por thread que as funções da biblioteca ajustam quando **falham**, com códigos como `ENOENT` ou `EACCES` (`strerror(errno)` e `perror` dão a mensagem). Mas uma função bem-sucedida pode deixar `errno` com qualquer valor, inclusive o de uma falha interna já resolvida. Por isso o protocolo é: primeiro olhar o retorno (`NULL`, `-1`, `EOF`), só então consultar `errno`. A exceção são funções como `strtol`, que não têm valor de retorno de erro: nelas se zera o `errno` antes da chamada e se testa depois.

Q59 **assert não é tratamento de erro**

> *Para que serve o `assert`? Por que é errado usá-lo para validar a entrada do usuário ou o retorno de `malloc`?*

`assert(condicao)` documenta e verifica uma **premissa do programador**, algo que só falha se houver bug: um índice dentro dos limites, um ponteiro que nunca deveria ser `NULL`. Se a condição for falsa, o programa aborta mostrando arquivo e linha. Só que compilar com `-DNDEBUG`, como se faz em versões de produção, **remove** todos os `assert`. Entrada inválida e falta de memória são situações esperadas em produção; precisam de um `if` com tratamento de verdade, senão a validação some exatamente quando é mais necessária.

Exercícios de código — respostas

Q60 **Estouro detectado**

> *O código abaixo calcula a média de dois `int` e falha para valores grandes. Explique o problema e escreva uma versão que nunca estoura. Teste com `INT_MAX` e `INT_MAX - 2`.*

```
int media(int a, int b) { return (a + b) / 2; }
```

`a + b` pode passar de `INT_MAX`, e estouro de `int` com sinal é comportamento indefinido; na prática costuma dar a volta e virar negativo. Duas saídas: fazer a soma num tipo maior ou somar a metade da diferença ao menor.

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

int media(int a, int b) {
    return (int) (((long long) a + b) / 2);  /* a soma cabe com folga em long long */
}

int media_sem_tipo_maior(int a, int b) {
    if (a > b) { int t = a; a = b; b = t; }
    return a + (b - a) / 2;  /* b - a não estoura quando a <= b e os dois têm o mesmo sinal */
}

int main(void) {
    printf("%d\n", media(INT_MAX, INT_MAX - 2));
    printf("%d\n", media_sem_tipo_maior(INT_MAX, INT_MAX - 2));
    printf("%d\n", media(-7, 2));
    return 0;
}
```

Saída

```
2147483646
2147483646
-2
```

A segunda versão tem uma restrição: com sinais opostos e valores extremos (`INT_MIN` e `INT_MAX`), `b - a` estoura. A versão com `long long` é a mais simples de acertar. Compilar com `-fsanitize=undefined` faz o programa avisar, em tempo de execução, quando um estouro de verdade acontece.

Q61 Erros por código de retorno

Escreva `int le_inteiro(const char *texto, int *saida)` que devolve 0 em caso de sucesso e códigos de erro distintos para "vazio", "não é número", "sobra texto" e "fora do intervalo de int". Escreva um `main` que testa cinco entradas e mostra a mensagem de cada erro.

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

enum { OK, ERRO_VAZIO, ERRO_NAO_NUMERO, ERRO_SOBRA, ERRO_INTERVALO };

static const char *mensagem[] = {
    "ok", "texto vazio", "não é número", "sobra texto depois do número", "fora do intervalo de
int",
};

int le_inteiro(const char *texto, int *saida) {
    if (texto == NULL || *texto == '\0')
        return ERRO_VAZIO;
    char *fim;
    errno = 0;
    long v = strtol(texto, &fim, 10);
    if (fim == texto)
        return ERRO_NAO_NUMERO;
    if (*fim != '\0')
        return ERRO_SOBRA;
    if (errno == ERANGE || v < INT_MIN || v > INT_MAX)
        return ERRO_INTERVALO;
    *saida = (int) v;  /* só escreve na saída em caso de sucesso */
    return OK;
}

int main(void) {
    const char *testes[] = { "42", "", "abc", "12kg", "99999999999" };
    for (int i = 0; i < 5; i++) {
        int v = 0;
        int erro = le_inteiro(testes[i], &v);
        if (erro == OK)
            printf("\"%s\" -> %d\n", testes[i], v);
        else
            printf("\"%s\" -> erro: %s\n", testes[i], mensagem[erro]);
    }
    return 0;
}
```

Saída

```
"42" -> 42
"" -> erro: texto vazio
"abc" -> erro: não é número
"12kg" -> erro: sobra texto depois do número
"99999999999" -> erro: fora do intervalo de int
```

O padrão do curso: o retorno diz se deu certo, e o resultado útil sai por um parâmetro ponteiro. Como `saida` só é escrita no sucesso, quem chama não corre o risco de usar um valor parcial.

Q62 Um teste automatizado mínimo

Sem biblioteca externa, escreva uma macro `CHECA(cond)` que conta testes executados e falhos, mostrando a linha de cada falha. Use-a para testar uma função `int fatorial(int n)` (com o caso n = 0) e mostre o resumo. Inclua de propósito um teste errado para ver a falha.

```
#include <stdio.h>

static int executados, falhos;

#define CHECA(cond) do {                                          \
        executados++;                                             \
        if (!(cond)) {                                            \
            falhos++;                                             \
            printf("FALHOU (linha %d): %s\n", __LINE__, #cond);   \
        }                                                         \
    } while (0)

long fatorial(int n) {
    long r = 1;
    for (int i = 2; i <= n; i++)
        r *= i;
    return r;
}

int main(void) {
    CHECA(fatorial(0) == 1);
    CHECA(fatorial(1) == 1);
    CHECA(fatorial(5) == 120);
    CHECA(fatorial(10) == 3628800);
    CHECA(fatorial(3) == 7);  /* errado de propósito */
    printf("%d testes, %d falhas\n", executados, falhos);
    return falhos ? 1 : 0;
}
```

Saída

```
FALHOU (linha 25): fatorial(3) == 7
5 testes, 1 falhas
```

O `#cond` transforma a expressão em texto, e o `do { ... } while (0)` faz a macro se comportar como um único comando (funciona até num `if` sem chaves). Sair com código diferente de zero quando há falha permite que o `make test` ou o CI detectem o problema. Frameworks como Unity e CMocka seguem a mesma ideia, com mais recursos.

Q63 Makefile com dependências automáticas

Escreva um `Makefile` para um projeto com `main.c`, `pilha.c` e `pilha.h` que: compila com `-Wall -Wextra -g`, gera as dependências de cabeçalhos automaticamente, tem os alvos `all` e `clean` e não confunde `clean` com um arquivo.

```
CC      := cc
CFLAGS  := -Wall -Wextra -g -MMD -MP
OBJS    := main.o pilha.o
PROG    := app

all: $(PROG)

$(PROG): $(OBJS)
        $(CC) $(CFLAGS) -o $@ $^

%.o: %.c
        $(CC) $(CFLAGS) -c -o $@ $<

clean:
        rm -f $(PROG) $(OBJS) $(OBJS:.o=.d)

-include $(OBJS:.o=.d)

.PHONY: all clean
```

- `-MMD -MP` fazem o compilador gravar, ao lado de cada `.o`, um arquivo `.d` com os cabeçalhos que ele incluiu; o `-include` traz essas regras para o make (o `-` ignora os `.d` que ainda não existem, na primeira compilação).
- `$@` é o alvo, `$^` todos os pré-requisitos e `$<` o primeiro.
- `.PHONY` diz que `all` e `clean` não são arquivos: sem isso, um arquivo chamado `clean` na pasta faria o make achar que o alvo já está atualizado.
- As linhas de comando começam com **TAB**, não espaços; com espaços, o make reclama de "missing separator".

MÓDULO 9

Programação de sistemas

Processos, threads, sincronização, sockets e sinais no Unix.

Artigos 47 a 51

#47 Processos: fork, exec e wait
#48 Threads com pthreads
#49 Sincronização: Mutexes e Variáveis de Condição
#50 Comunicação em Rede com Sockets
#51 Sinais Unix e Eventos do Sistema Operacional

Perguntas conceituais — respostas

Q64 fork

O que `fork()` devolve no processo pai, no filho e em caso de erro? Por que um `printf("oi")` sem `\n` antes do `fork` pode aparecer duas vezes?

No pai, devolve o **PID do filho**; no filho, devolve **0**; em caso de erro, `-1` (e nenhum filho é criado). O `printf` sem `\n` deixa o texto no buffer de saída da biblioteca, ainda não escrito. O `fork` copia o processo inteiro, buffer incluído, e cada processo descarrega a sua cópia ao terminar. Por isso se faz `fflush(stdout)` antes do `fork`, ou se termina a linha com `\n` quando a saída é um terminal.

Q65 Zumbis

O que é um processo zumbi e como evitá-lo? E um processo órfão?

Um **zumbi** é um filho que já terminou mas cujo pai ainda não leu o código de saída com `wait` / `waitpid`; ele não consome CPU nem memória, mas ocupa uma entrada na tabela de processos. Evita-se chamando `wait` (ou tratando `SIGCHLD`). Um **órfão** é um filho cujo pai terminou antes dele; ele é adotado pelo `init` (PID 1) ou por um processo *subreaper*, que faz o `wait` por ele.

Q66 Mutex e variável de condição

Por que o `pthread_cond_wait` precisa receber o mutex? E por que a condição é testada num `while`, não num `if`?

O `pthread_cond_wait` **libera o mutex e dorme** numa única operação atômica e, ao acordar, readquire o mutex antes de retornar. Se fossem dois passos, um sinal emitido entre liberar e dormir se perderia. O `while` existe porque a thread pode acordar e encontrar a condição falsa: outra thread pode ter chegado antes e consumido o item, e o padrão permite despertares espúrios, sem nenhum sinal. Por isso se testa de novo depois de acordar.

Q67 O que fazer num tratador de sinal

Por que é errado chamar `printf` ou `malloc` dentro de um tratador de sinal? Qual o padrão recomendado?

O sinal pode chegar no meio de qualquer instrução, inclusive dentro de um `printf` ou `malloc` que o próprio programa estava executando. Chamá-los de novo no tratador pode corromper as estruturas internas da biblioteca ou causar um impasse num lock interno. Só funções **async-signal-safe** (como `write` e `_exit`) são permitidas. O padrão é o tratador apenas ligar uma variável `volatile sig_atomic_t` e o laço principal verificá-la e fazer o trabalho de verdade.

Exercícios de código — respostas

Q68 fork, exec e wait

Escreva um programa que cria um processo filho para executar `echo` com os argumentos `"olá do filho"`, espera o filho terminar e mostra o código de saída dele.

```
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    fflush(stdout);  /* nada pendente no buffer para ser duplicado pelo fork */
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        return 1;
    }
    if (pid == 0) {
        execlp("echo", "echo", "olá do filho", (char *) NULL);
        perror("execlp");  /* só chega aqui se o exec falhar */
        _exit(127);
    }
    int status;
    if (waitpid(pid, &status, 0) < 0) {
        perror("waitpid");
        return 1;
    }
    if (WIFEXITED(status))
        printf("filho terminou com código %d\n", WEXITSTATUS(status));
    return 0;
}
```

Saída

```
olá do filho
filho terminou com código 0
```

O `exec` **substitui** o programa do processo filho pelo `echo`: se der certo, nada depois dele executa. Por isso o tratamento de erro vem logo em seguida, com `_exit` (e não `exit`, que descarregaria buffers copiados do pai). Com `-std=c17`, o `#define _POSIX_C_SOURCE` é necessário para expor as funções POSIX nos cabeçalhos.

Q69 Somando em paralelo com threads

> *Some os números de 1 a 10.000.000 dividindo o trabalho entre 4 threads, cada uma com o seu intervalo e o seu resultado parcial, sem nenhum mutex. Explique por que não é preciso mutex.*

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

#define N 10000000L
#define THREADS 4

typedef struct { long inicio, fim, parcial; } Tarefa;

static void *soma(void *arg) {
    Tarefa *t = arg;
    t->parcial = 0;
    for (long i = t->inicio; i <= t->fim; i++)
        t->parcial += i;
    return NULL;
}

int main(void) {
    pthread_t th[THREADS];
    Tarefa tarefas[THREADS];
    long bloco = N / THREADS;

    for (int i = 0; i < THREADS; i++) {
        tarefas[i].inicio = i * bloco + 1;
        tarefas[i].fim = (i == THREADS - 1) ? N : (i + 1) * bloco;
        if (pthread_create(&th[i], NULL, soma, &tarefas[i]) != 0) {
            fprintf(stderr, "falha ao criar thread\n");
            return 1;
        }
    }
    long total = 0;
    for (int i = 0; i < THREADS; i++) {
        pthread_join(th[i], NULL);  /* espera e garante que o parcial já foi escrito */
        total += tarefas[i].parcial;
    }
    printf("total: %ld (esperado %ld)\n", total, N * (N + 1) / 2);
    return 0;
}
```

Saída

```
total: 50000005000000 (esperado 50000005000000)
```

Cada thread escreve só na própria `Tarefa`, e o `main` só lê os parciais depois do `pthread_join`, que garante que a escrita terminou e está visível. Não há dado compartilhado sendo escrito por duas threads ao mesmo tempo, então não há condição de corrida. Compile com `-pthread`.

Q70 Contador compartilhado com mutex

> *Crie 4 threads que incrementam um contador global 100.000 vezes cada. Mostre o resultado com o mutex protegendo o incremento e explique o que muda sem ele.*

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

static long contador = 0;
static pthread_mutex_t trava = PTHREAD_MUTEX_INITIALIZER;

static void *incrementa(void *arg) {
    (void) arg;
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&trava);
        contador++;  /* ler, somar e gravar: só uma thread por vez */
        pthread_mutex_unlock(&trava);
    }
    return NULL;
}

int main(void) {
    pthread_t th[4];
    for (int i = 0; i < 4; i++)
        pthread_create(&th[i], NULL, incrementa, NULL);
    for (int i = 0; i < 4; i++)
        pthread_join(th[i], NULL);
    printf("contador: %ld\n", contador);
    return 0;
}
```

Saída

```
contador: 400000
```

Sem o mutex, `contador++` vira "ler, somar 1, gravar", e duas threads podem ler o mesmo valor e gravar o mesmo resultado: incrementos se perdem e o total fica abaixo de 400.000, variando a cada execução. Travar e destravar 400 mil vezes custa caro; num caso real, cada thread somaria num contador local e travaria uma única vez no fim, ou usaria `atomic_long` de `<stdatomic.h>`.

Q71 Encerrando com Ctrl+C

> *Escreva um programa que fica num laço "trabalhando" e, ao receber `SIGINT`, termina o laço de forma limpa e imprime quantas voltas deu. Para o teste ser automático, o próprio programa envia o sinal a si mesmo depois de 3 voltas com `raise(SIGINT)`.*

```c
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
#include <signal.h>

static volatile sig_atomic_t parar = 0;

static void ao_interromper(int sinal) {
    (void) sinal;
    parar = 1;  /* só isso: o trabalho de verdade fica no laço principal */
}

int main(void) {
    struct sigaction sa = {0};
    sa.sa_handler = ao_interromper;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGINT, &sa, NULL) < 0) {
        perror("sigaction");
        return 1;
    }

    long voltas = 0;
    while (!parar) {
        voltas++;  /* aqui entraria o trabalho real */
        if (voltas == 3)
            raise(SIGINT);  /* simula o Ctrl+C do usuário */
    }
    printf("encerrado com limpeza depois de %ld voltas\n", voltas);
    return 0;
}
```

Saída

```
encerrado com limpeza depois de 3 voltas
```

`sigaction` é preferível ao `signal` antigo, cujo comportamento varia entre sistemas. `volatile sig_atomic_t` garante que a escrita no tratador e a leitura no laço não sejam otimizadas nem divididas ao meio. Rodando de verdade no terminal, basta tirar o `raise` e apertar Ctrl+C.

PARA IR ALÉM

Desafios integradores

Projetos maiores, que juntam vários módulos. Não há uma única resposta certa: o gabarito mostra um caminho.

D1

Uma calculadora de terminal

*Escreva uma calculadora de terminal com sete operações: soma, subtração, multiplicação, divisão, potência (expoente inteiro), porcentagem (`15 % 200` é 15% de 200) e raiz quadrada (`raiz 2`). Ela deve: (a) aceitar vírgula ou ponto como separador decimal; (b) guardar o último resultado em `ans`, utilizável na conta seguinte; (c) recusar divisão por zero, raiz de negativo, expoente fracionário e texto que não é número com uma mensagem clara, **sem encerrar o programa**; (d) ter os comandos `hist` (histórico) e `sair`. Separe o núcleo (`calc.c` e `calc.h`, que não fazem entrada nem saída) da interface (`main.c`).*

Caminho da solução

- **Núcleo**: `calc_avaliar` recebe a linha e devolve um código de erro, com o resultado num parâmetro ponteiro (o padrão do módulo 8). As operações ficam numa tabela de símbolos, como no exercício de ponteiros para função.
- **Números**: o `strtod` só entende ponto no locale padrão, então o núcleo troca a vírgula por ponto numa cópia do texto antes de converter. Os resultados são `double`, com os limites que isso traz (veja a nota depois da saída).
- **Interface**: lê linhas com `fgets`, nunca com `scanf`, guarda o histórico num vetor dinâmico que cresce com `realloc` e libera tudo ao sair.

`calc.h`:

```
#ifndef CALC_H
#define CALC_H

typedef enum {
    CALC_OK,
    CALC_FORMATO,       /* linha que não é "a op b" nem "raiz a" */
    CALC_NUMERO,        /* operando que não é número */
    CALC_OPERACAO,      /* símbolo desconhecido */
    CALC_DIV_ZERO,
    CALC_RAIZ_NEGATIVA,
    CALC_EXPOENTE,
} CalcErro;

typedef struct {
    double ans;  /* último resultado */
} Calculadora;

/* Avalia "12,5 + 3", "2 ^ 10", "15 % 200" ou "raiz 2"; "ans" é o último resultado. */
CalcErro calc_avaliar(Calculadora *c, const char *linha, double *resultado);
const char *calc_mensagem(CalcErro e);

#endif
```

`calc.c`:

```
#include "calc.h"

#include <ctype.h>
#include <errno.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static CalcErro numero(const Calculadora *c, const char *texto, double *v) {
    if (strcmp(texto, "ans") == 0) {
        *v = c->ans;
        return CALC_OK;
    }
    char copia[64];
    if (strlen(texto) >= sizeof copia)
        return CALC_NUMERO;
    strcpy(copia, texto);
    char *virgula = strchr(copia, ',');
    if (virgula)
        *virgula = '.';  /* strtod segue o locale "C": só entende ponto */
    char *fim;
    errno = 0;
    *v = strtod(copia, &fim);
    if (fim == copia || *fim != '\0' || errno == ERANGE || !isfinite(*v))
        return CALC_NUMERO;
    return CALC_OK;
}

static CalcErro operar(char op, double a, double b, double *r) {
    switch (op) {
    case '+': *r = a + b; return CALC_OK;
    case '-': *r = a - b; return CALC_OK;
    case '*': *r = a * b; return CALC_OK;
    case '/':
        if (b == 0) return CALC_DIV_ZERO;
        *r = a / b;
        return CALC_OK;
    case '%': *r = a * b / 100; return CALC_OK;
    case '^':
        if (b != floor(b) || fabs(b) > 999) return CALC_EXPOENTE;
        if (a == 0 && b < 0) return CALC_DIV_ZERO;
        *r = pow(a, b);
        return CALC_OK;
    }
    return CALC_OPERACAO;
}

CalcErro calc_avaliar(Calculadora *c, const char *linha, double *resultado) {
    char t[3][64];
    int n = sscanf(linha, "%63s %63s %63s", t[0], t[1], t[2]);
```

```
    double a, b, r;
    CalcErro e;

    if (n == 2 && strcmp(t[0], "raiz") == 0) {
        if ((e = numero(c, t[1], &a)) != CALC_OK) return e;
        if (a < 0) return CALC_RAIZ_NEGATIVA;
        r = sqrt(a);
    } else if (n == 3 && strlen(t[1]) == 1) {
        if ((e = numero(c, t[0], &a)) != CALC_OK) return e;
        if ((e = numero(c, t[2], &b)) != CALC_OK) return e;
        if ((e = operar(t[1][0], a, b, &r)) != CALC_OK) return e;
    } else {
        return n == 3 ? CALC_OPERACAO : CALC_FORMATO;
    }
    if (!isfinite(r)) return CALC_NUMERO;  /* resultado grande demais para um double */
    c->ans = *resultado = r;
    return CALC_OK;
}

const char *calc_mensagem(CalcErro e) {
    static const char *m[] = {
        [CALC_OK] = "ok",
        [CALC_FORMATO] = "use: número operação número, ou raiz número",
        [CALC_NUMERO] = "não é um número válido",
        [CALC_OPERACAO] = "operação desconhecida",
        [CALC_DIV_ZERO] = "divisão por zero",
        [CALC_RAIZ_NEGATIVA] = "raiz de número negativo",
        [CALC_EXPOENTE] = "o expoente precisa ser inteiro, de -999 a 999",
    };
    return m[e];
}
```

`main.c`:

```
#include "calc.h"

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

/* 16 dígitos significativos, vírgula decimal e sem zeros à direita */
static void formata(double v, char *saida, size_t tam) {
    snprintf(saida, tam, "%.16g", v);
    char *ponto = strchr(saida, '.');
    if (ponto)
        *ponto = ',';
}

int main(void) {
    Calculadora calc = {0};
    char **historico = NULL;
    size_t n_hist = 0;
    char linha[256];

    puts("Calculadora: + - * / ^ % raiz  |  ans, hist, sair");
    for (printf("> "); fgets(linha, sizeof linha, stdin); printf("> ")) {
        linha[strcspn(linha, "\n")] = '\0';
        if (strcmp(linha, "sair") == 0)
            break;
        if (strcmp(linha, "hist") == 0) {
            for (size_t i = 0; i < n_hist; i++)
                puts(historico[i]);
            continue;
        }
        if (linha[0] == '\0')
            continue;

        double r;
        CalcErro e = calc_avaliar(&calc, linha, &r);
        if (e != CALC_OK) {
            printf("erro: %s\n", calc_mensagem(e));  /* a conta errada não derruba o programa */
            continue;
        }
        char num[64], conta[sizeof linha + 80];
        formata(r, num, sizeof num);
        snprintf(conta, sizeof conta, "%s = %s", linha, num);
        puts(conta);

        char **tmp = realloc(historico, (n_hist + 1) * sizeof *tmp);
        char *copia = malloc(strlen(conta) + 1);
        if (tmp == NULL || copia == NULL) {
            free(copia);
            historico = tmp ? tmp : historico;
            fputs("sem memória para o histórico\n", stderr);
            continue;
```

```
        }
        historico = tmp;
        historico[n_hist++] = strcpy(copia, conta);
    }
    for (size_t i = 0; i < n_hist; i++)
        free(historico[i]);
    free(historico);
    return 0;
}
```

Entrada digitada:

```
0,1 + 0,2
10 / 3
ans * 3
1 / 0
2 ^ 100
raiz 2
ans * ans
15 % 200
2 ^ 0,5
raiz -4
abc + 1
hist
sair
```

Saída

```
Calculadora: + - * / ^ % raiz  |  ans, hist, sair
> 0,1 + 0,2 = 0,3
> 10 / 3 = 3,333333333333333
> ans * 3 = 10
> erro: divisão por zero
> 2 ^ 100 = 1,267650600228229e+30
> raiz 2 = 1,414213562373095
> ans * ans = 2
> 15 % 200 = 30
> erro: o expoente precisa ser inteiro, de -999 a 999
> erro: raiz de número negativo
> erro: não é um número válido
> 0,1 + 0,2 = 0,3
10 / 3 = 3,333333333333333
ans * 3 = 10
2 ^ 100 = 1,267650600228229e+30
raiz 2 = 1,414213562373095
ans * ans = 2
15 % 200 = 30
>
```

Compile os três arquivos juntos: `gcc -Wall -Wextra calc.c main.c -o calc -lm`. Com a entrada redirecionada, o terminal não ecoa o que foi digitado, e cada resultado aparece logo depois do `>`.

A saída parece perfeita, mas o `double` está escondendo erros. `0,1 + 0,2` guarda 0,30000000000000004, e `ans * ans` depois de `raiz 2` guarda 2,0000000000000004: os dois só aparecem como `0,3` e `2` porque o

`%.16g` arredonda para 16 dígitos. Troque por `%.17g` e eles surgem. `10 / 3` seguido de `ans * 3` dá 10 por sorte, porque os erros de arredondamento se cancelam. Calculadoras de verdade usam aritmética decimal (como o `_Decimal64` do C23, ainda pouco suportado) ou trabalham com dígitos a mais do que mostram. Uma boa extensão é guardar os números como frações de inteiros, o que torna exatas as quatro operações e a potência inteira.

Uma versão gráfica fica como extensão: com GTK ou raylib, a janela só monta o texto da conta com botões e chama o mesmo `calc_avaliar`, sem mudar uma linha do núcleo.

D2

Um mini-shell

Escreva um mini-shell que mostra um prompt, lê um comando com argumentos, executa-o num processo filho e espera o fim, e tem três comandos internos: `cd`, `historico` (os últimos 10 comandos, numa lista circular) e `sair`. Comandos inexistentes devem mostrar uma mensagem e devolver o prompt. Como extensão, implemente o redirecionamento `> arquivo`.

Caminho da solução

- **Leitura**: `fgets` numa linha de até 1024 caracteres, separada em palavras com `strtok` num vetor `char *argv[64]` terminado em `NULL`, o formato que o `execvp` espera.
- **Comandos internos**: precisam rodar no próprio shell. Um `cd` executado num filho mudaria o diretório do filho, que morre em seguida; por isso o shell testa `argv[0]` antes do `fork` e chama `chdir` diretamente.
- **Execução**: `fork`; no filho, `execvp(argv[0], argv)` e, se ele voltar, `perror` e `_exit(127)`; no pai, `waitpid` e o código de saída guardado para mostrar no prompt.
- **Histórico**: um vetor de 10 `char *` com um índice que avança módulo 10 (a fila circular do módulo 7); ao sobrescrever uma posição, o `free` da string antiga vem antes.
- **Redirecionamento**: no filho, antes do `exec`, `open(arquivo, O_WRONLY | O_CREAT | O_TRUNC, 0644)` e `dup2(fd, STDOUT_FILENO)`; o `>` e o nome do arquivo saem do `argv`.
- **Sinais**: o shell ignora `SIGINT` (Ctrl+C não deve matá-lo), e o filho restaura o comportamento padrão com `signal(SIGINT, SIG_DFL)` antes do `exec`, para que Ctrl+C interrompa só o comando em execução.

Teste com Valgrind ou `-fsanitize=address`: ao digitar `sair` depois de mais de 10 comandos, nenhuma string do histórico pode ter vazado.

Fim do gabarito.
Dominando a Linguagem C em 1 Ano · theknowledgeisle.com