Manual Avançado de C

Ponteiros, Alocação Dinâmica e Structs

Aplicação: Uma MLP para classificação MNIST

Seu Nome

13 de fevereiro de 2026

Conteúdo

Capítulo 1

Ponteiros em C

1.1 Conceitos Fundamentais

Um ponteiro é uma variável que armazena o endereço de memória de outra variável. Em C, eles são essenciais para manipulação direta da memória, passagem eficiente de parâmetros e implementação de estruturas de dados dinâmicas.

1.1.1 Declaração e Operadores Básicos

```c
int x = 10;
int *p;        // declara um ponteiro para inteiro
p = &x;        // p recebe o endereco de x
printf("%d\n", *p); // acessa o valor apontado (de refer ncia)
```

Operadores:

- `&` : operador endereço (referência)
- `*` : operador de referência (acesso ao valor apontado)

1.1.2 Aritmética de Ponteiros

Ao incrementar um ponteiro, ele avança pelo tamanho do tipo apontado.

```c
int arr[5] = {10,20,30,40,50};
int *ptr = arr; // ptr aponta para arr[0]
ptr++;          // agora aponta para arr[1]
printf("%d\n", *ptr); // 20
```

Isso permite percorrer arrays de forma eficiente.

1.1.3 Ponteiros e Arrays

O nome de um array é um ponteiro constante para o primeiro elemento.

```c
int vet[3] = {1,2,3};
int *p = vet;   // equivalente a int *p = &vet[0];
vet[1] == *(p+1); // verdadeiro
```

1.1.4 Ponteiros para Ponteiros — Uso Real

Ponteiros para ponteiros são essenciais quando precisamos modificar o endereço de um ponteiro dentro de uma função (ex.: alocação dinâmica) ou construir estruturas como matrizes dinâmicas.

Exemplo 1 — Alocação dinâmica dentro da função

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

void alloc_vector(int **v, size_t n) {
    *v = malloc(n * sizeof(int));
    if (*v == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    for (size_t i = 0; i < n; i++)
        (*v)[i] = i * 10;
}

int main() {
    int *vec = NULL;

    alloc_vector(&vec, 5);

    for (int i = 0; i < 5; i++)
        printf("%d ", vec[i]);

    free(vec);
    return 0;
}
```

O que aconteceu:

- Passamos o endereço do ponteiro: `&vec`
- A função modifica o ponteiro original
- Memória foi alocada na **heap**
- Evitamos segmentation fault verificando `malloc`

Exemplo 2 — Matriz dinâmica usando ponteiro para ponteiro

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

void matrix_alloc(int ***M, size_t rows, size_t cols) {
    *M = malloc(rows * sizeof(int*));
    if (*M == NULL) exit(EXIT_FAILURE);
```

```
7
8       for (size_t i = 0; i < rows; i++) {
9           (*M)[i] = malloc(cols * sizeof(int));
10          if ((*M)[i] == NULL) exit(EXIT_FAILURE);
11      }
12  }
13
14  void matrix_free(int **M, size_t rows) {
15      for (size_t i = 0; i < rows; i++)
16          free(M[i]);
17      free(M);
18  }
19
20  int main() {
21      int **A = NULL;
22
23      matrix_alloc(&A, 3, 3);
24
25      for (int i = 0; i < 3; i++)
26          for (int j = 0; j < 3; j++)
27              A[i][j] = i + j;
28
29      for (int i = 0; i < 3; i++) {
30          for (int j = 0; j < 3; j++)
31              printf("%d ", A[i][j]);
32          printf("\n");
33      }
34
35      matrix_free(A, 3);
36      return 0;
37  }
```

Pontos de engenharia:

- `int ***M` permite modificar a matriz original
- Cada linha é alocada separadamente
- Liberação correta evita *memory leak*
- Estrutura equivalente a uma matriz dinâmica real

1.1.5 Ponteiros para Funções

Permitem que funções sejam passadas como argumentos.

```
1  int soma(int a, int b) { return a+b; }
2  int (*op)(int,int);   // ponteiro para funcao que recebe dois
       ints e retorna int
3  op = &soma;
4  int resultado = op(3,4); // 7
```

São úteis para callbacks e algoritmos genéricos (ex: `qsort`).

Capítulo 2

Alocação Dinâmica de Memória

2.1 Stack vs Heap

- **Stack**: memória automática, gerenciada pelo compilador. Variáveis locais, parâmetros de função. Alocação e liberação rápidas, mas tamanho limitado (pode causar stack overflow).
- **Heap**: memória dinâmica, controlada pelo programador. Alocada com `malloc`/`calloc`/`realloc` e liberada com `free`. Maior flexibilidade, porém mais lenta e sujeita a vazamentos.

Organização da memória de um processo

- **Text**: código executável (somente leitura).
- **Data**: variáveis globais/estáticas inicializadas.
- **BSS**: variáveis globais/estáticas não inicializadas (zeradas).
- **Heap**: cresce para cima (endereços crescentes).
- **Stack**: cresce para baixo (endereços decrescentes).

2.2 Funções de Alocação

Todas estão em `<stdlib.h>`.

2.2.1 malloc

```
1 void *malloc(size_t size);
```

Aloca um bloco de `size` bytes na heap. Retorna ponteiro para o início do bloco ou `NULL` se falhar. O conteúdo não é inicializado.

2.2.2 calloc

```
1 void *calloc(size_t nmemb, size_t size);
```

Aloca espaço para um array de `nmemb` elementos de `size` bytes cada. Inicializa todos os bytes com zero.

2.2.3 realloc

```
void *realloc(void *ptr, size_t new_size);
```

Redimensiona o bloco apontado por `ptr` para `new`$_s$*izebytes.Podemoveroblocosenecessário.Retornanovo*

2.2.4 free

```
void free(void *ptr);
```

Libera o bloco de memória previamente alocado. Após free, o ponteiro se torna inválido (deve ser atribuído a NULL para evitar uso acidental).

2.2.5 Exemplo Completo

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

int main() {
    int *vetor;
    int n = 10;
    vetor = (int*) malloc(n * sizeof(int));
    if (vetor == NULL) {
        fprintf(stderr, "Erro de alocacao\n");
        return 1;
    }
    for (int i = 0; i < n; i++)
        vetor[i] = i * i;
    // redimensiona para 20 elementos
    int *temp = (int*) realloc(vetor, 20 * sizeof(int));
    if (temp == NULL) {
        free(vetor);
        return 1;
    }
    vetor = temp;
    // ...
    free(vetor);
    return 0;
}
```

2.3 Erros Comuns e Boas Práticas

- Não verificar o retorno de malloc/calloc.
- Esquecer de liberar memória (*memory leak*).
- Acessar memória já liberada (*use-after-free*).
- Liberar o mesmo ponteiro duas vezes (*double free*).

- Escrever além dos limites do bloco alocado (*buffer overflow*).
- Usar free em ponteiros que não vieram de alocação dinâmica.

Boas práticas:

- Sempre inicialize ponteiros com NULL.
- Após free, atribua NULL ao ponteiro.
- Use sizeof para calcular o tamanho correto.
- Prefira calloc se precisar de zeros.
- Utilize ferramentas como valgrind para detectar vazamentos.

Capítulo 3

Structs e Unions

3.1 Definindo Structs

Struct é uma coleção de variáveis (membros) agrupadas sob um mesmo nome.

```
struct Ponto {
    int x;
    int y;
};
```

Para declarar variáveis:

```
struct Ponto p1;
p1.x = 10;
p1.y = 20;
```

3.2 typedef

Facilita a declaração:

```
typedef struct {
    int x;
    int y;
} Ponto;
Ponto p2;
```

3.3 Ponteiros para Struct

Acesso aos membros via seta (->).

```
Ponto *ptr = &p2;
ptr->x = 30;  // equivalente a (*ptr).x = 30;
```

3.4 Alinhamento e Padding

O compilador pode inserir bytes de preenchimento entre os membros para garantir alinhamento em endereços múltiplos do tamanho da palavra da máquina. Isso

afeta o tamanho total da struct.

```
struct Exemplo {
    char c;       // 1 byte
    int i;        // 4 bytes (geralmente alinhado a 4)
}; // sizeof pode ser 8 (3 bytes de padding)
```

Para evitar padding, pode-se usar #pragma pack (não portável) ou reordenar membros.

3.5 Unions

Uma union permite armazenar diferentes tipos no mesmo espaço de memória. O tamanho é o do maior membro.

```
union Valor {
    int i;
    float f;
    char str[20];
};
```

Útil para economizar memória quando apenas um dos campos é usado por vez.

Capítulo 4

Aplicação Prática: MLP para MNIST em C

Neste capítulo, desenvolveremos um framework simples de rede neural MLP (Multi-Layer Perceptron) para classificar dígitos manuscritos do dataset MNIST. O foco será no uso correto de ponteiros, alocação dinâmica e structs.

4.1 Visão Geral do MNIST

O MNIST contém 70.000 imagens 28x28 em escala de cinza (0-255), divididas em 60.000 treino e 10.000 teste. Cada imagem representa um dígito de 0 a 9.

4.2 Estruturas de Dados

Definiremos structs para representar matrizes, camadas e a rede.

4.2.1 Matriz

```c
typedef struct {
    int linhas;
    int colunas;
    float **dados;  // ponteiro para ponteiro (linhas)
} Matriz;
```

Funções básicas:

```c
Matriz* matriz_criar(int lin, int col) {
    Matriz *m = (Matriz*) malloc(sizeof(Matriz));
    m->linhas = lin;
    m->colunas = col;
    m->dados = (float**) malloc(lin * sizeof(float*));
    for (int i = 0; i < lin; i++) {
        m->dados[i] = (float*) malloc(col * sizeof(float));
        // inicializar com zeros
        for (int j = 0; j < col; j++)
            m->dados[i][j] = 0.0f;
    }
```

```
    return m;
}

void matriz_destruir(Matriz *m) {
    if (!m) return;
    for (int i = 0; i < m->linhas; i++)
        free(m->dados[i]);
    free(m->dados);
    free(m);
}
```

4.2.2 Camada

Cada camada possui pesos, biases, e armazena a entrada/saída para retropropagação.

```
typedef enum { SIGMOID, RELU, SOFTMAX } Ativacao;

typedef struct {
    int entrada_size;
    int saida_size;
    Matriz *pesos;      // [saida_size x entrada_size]
    Matriz *bias;       // [saida_size x 1]
    Matriz *entrada;    // guarda para backprop
    Matriz *saida;      // saida apos ativacao
    Ativacao ativ;
} Camada;
```

Criação:

```
Camada* camada_criar(int in, int out, Ativacao at) {
    Camada *cam = (Camada*) malloc(sizeof(Camada));
    cam->entrada_size = in;
    cam->saida_size = out;
    cam->pesos = matriz_criar(out, in);
    cam->bias = matriz_criar(out, 1);
    cam->entrada = NULL;
    cam->saida = NULL;
    cam->ativ = at;
    // inicializar pesos aleatoriamente (pequenos)
    for (int i = 0; i < out; i++) {
        for (int j = 0; j < in; j++) {
            cam->pesos->dados[i][j] = ((float)rand() / RAND_MAX -
                0.5f) * 0.1f;
        }
        cam->bias->dados[i][0] = 0.0f;
    }
    return cam;
}

void camada_destruir(Camada *cam) {
    if (!cam) return;
    matriz_destruir(cam->pesos);
```

```
23      matriz_destruir(cam->bias);
24      if (cam->entrada) matriz_destruir(cam->entrada);
25      if (cam->saida) matriz_destruir(cam->saida);
26      free(cam);
27  }
```

4.2.3 Rede

A rede é uma lista (array dinâmico) de camadas.

```
1  typedef struct {
2      int num_camadas;
3      Camada **camadas;  // array de ponteiros para Camada
4  } MLP;
```

Criação e destruição:

```
1  MLP* mlp_criar(int num_camadas, int *tamanhos, Ativacao *
       ativacoes) {
2      MLP *rede = (MLP*) malloc(sizeof(MLP));
3      rede->num_camadas = num_camadas;
4      rede->camadas = (Camada**) malloc(num_camadas * sizeof(Camada
           *));
5      for (int i = 0; i < num_camadas; i++) {
6          int in = (i == 0) ? tamanhos[0] : tamanhos[i]; // na
               verdade, tamanhos[i] eh saida da camada i? Precisamos:
               entrada = tamanho anterior
7          int out = tamanhos[i+1];
8          // Corrigindo: tamanhos[0] = entrada, tamanhos[1] = saida
                cam1, ...
9          // Vamos adotar: tamanhos[0] = entrada, tamanhos[1] =
               saida primeira camada, etc.
10         // num_camadas = numero de camadas, entao tamanhos tem
               num_camadas+1 elementos.
11     }
12     return rede;
13 }
```

Para simplificar, faremos uma função que recebe array de tamanhos (incluindo entrada e saída de cada camada).

4.3 Operações com Matrizes

Implementaremos funções essenciais:

- $\texttt{matriz}_multiplicar(A, B) : retorna nova matriz A * B.$
- $\texttt{matriz}_somar(A, B) : soma elemento a elemento (in-place ou nova).$
- $\texttt{matriz}_aplicar\_ativacao(M, tipo) : aplica função de ativação em cada elemento.$
- $\texttt{matriz}_softmax(M) : aplica softmax (para classificação).$

Exemplo de multiplicação:

```c
Matriz* matriz_multiplicar(Matriz *A, Matriz *B) {
    if (A->colunas != B->linhas) return NULL;
    Matriz *C = matriz_criar(A->linhas, B->colunas);
    for (int i = 0; i < A->linhas; i++) {
        for (int j = 0; j < B->colunas; j++) {
            float soma = 0.0f;
            for (int k = 0; k < A->colunas; k++) {
                soma += A->dados[i][k] * B->dados[k][j];
            }
            C->dados[i][j] = soma;
        }
    }
    return C;
}
```

4.4 Forward Propagation

Dada uma entrada (matriz coluna), propaga pela rede.

```c
Matriz* mlp_forward(MLP *rede, Matriz *entrada) {
    Matriz *atual = entrada;
    for (int i = 0; i < rede->num_camadas; i++) {
        Camada *cam = rede->camadas[i];
        // guarda entrada para backprop
        cam->entrada = matriz_copiar(atual); // precisamos de
           matriz_copiar
        // z = pesos * entrada + bias
        Matriz *z = matriz_multiplicar(cam->pesos, atual);
        // somar bias (broadcast) - faremos funcao especifica
        matriz_soma_bias(z, cam->bias);
        // aplicar ativacao
        if (cam->ativ == SOFTMAX) {
            matriz_softmax(z);
        } else {
            matriz_aplicar_ativacao(z, cam->ativ);
        }
        // se nao for a primeira iteracao, liberar saida anterior
        if (cam->saida) matriz_destruir(cam->saida);
        cam->saida = z;
        atual = z;
    }
    return atual; // saida da ultima camada
}
```

4.5 Backpropagation e Treinamento

Devido à complexidade, implementaremos uma versão simplificada com gradiente descendente. Para uma MLP, precisamos calcular o gradiente para cada camada

e atualizar pesos. Vamos assumir uso de entropia cruzada e softmax na saída.

Devido à extensão, apresentaremos apenas o esqueleto e as funções de atualização.

4.6 Leitura do Dataset MNIST

O formato MNIST (IDX) é simples: arquivos contêm cabeçalho de 16 bytes e depois os dados. Implementaremos funções para ler imagens e rótulos.

Exemplo de leitura de imagens:

```
unsigned char* ler_imagens(const char *filename, int *n_imagens)
   {
    FILE *fp = fopen(filename, "rb");
    if (!fp) return NULL;
    int magic, n, rows, cols;
    fread(&magic, 4, 1, fp); magic = __builtin_bswap32(magic); //
        big-endian
    fread(&n, 4, 1, fp); n = __builtin_bswap32(n);
    fread(&rows, 4, 1, fp); rows = __builtin_bswap32(rows);
    fread(&cols, 4, 1, fp); cols = __builtin_bswap32(cols);
    *n_imagens = n;
    unsigned char *data = (unsigned char*) malloc(n * rows * cols
       );
    fread(data, 1, n * rows * cols, fp);
    fclose(fp);
    return data;
}
```

Normalizamos os pixels para [0,1] convertendo para float.

4.7 Exemplo Completo de Uso

Abaixo, um trecho que cria uma rede com uma camada oculta de 128 neurônios e camada de saída softmax com 10 neurônios, treina por algumas épocas e avalia.

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

// ... (todas as definicoes anteriores)

int main() {
    srand(time(NULL));
    // carregar dados MNIST (supondo arquivos na pasta)
    int n_train;
    unsigned char *img_train = ler_imagens("train-images.idx3-
       ubyte", &n_train);
    unsigned char *lbl_train = ler_rotulos("train-labels.idx1-
       ubyte", &n_train);
    // converter para matrizes de entrada (float)
    // cada imagem 28*28 = 784
```

```
    int entrada_size = 784;
    int saida_size = 10;
    int camadas_tamanhos[] = {entrada_size, 128, saida_size};
    Ativacao ativacoes[] = {RELU, SOFTMAX};
    MLP *rede = mlp_criar(2, camadas_tamanhos, ativacoes); // 2 camadas
    // loop de treinamento (simplificado)
    for (int epoch = 0; epoch < 5; epoch++) {
        float loss = 0.0f;
        for (int i = 0; i < n_train; i++) {
            // preparar entrada: matriz coluna 784x1
            Matriz *entrada = matriz_criar(entrada_size, 1);
            for (int j = 0; j < entrada_size; j++) {
                entrada->dados[j][0] = img_train[i*entrada_size + j] / 255.0f;
            }
            // target: one-hot
            Matriz *target = matriz_criar(saida_size, 1);
            target->dados[lbl_train[i]][0] = 1.0f;

            // forward
            Matriz *saida = mlp_forward(rede, entrada);

            // calcular perda (cross-entropy) e gradientes (backward)
            // ... (implementar backprop)

            // atualizar pesos (gradient descent)
            // ...

            matriz_destruir(entrada);
            matriz_destruir(target);
        }
        printf("Epoca %d concluida\n", epoch);
    }
    // avaliar no teste...
    // liberar memoria
    mlp_destruir(rede);
    free(img_train);
    free(lbl_train);
    return 0;
}
```

4.8 Observações sobre o Código

- O código acima é um esqueleto; a implementação completa exigiria funções detalhadas de backprop e atualização. - É fundamental gerenciar corretamente a memória: cada matriz_criar deve ter seu matriz_destruir correspondente. - O uso de ponteiros duplos para matrizes permite indexação conveniente, mas pode ser menos eficiente que uma única alocação contígua. Para desempenho,

poderíamos armazenar todos os dados em um vetor linear e acessar via índice.
- O projeto serve como excelente exercício para consolidar ponteiros, alocação dinâmica e structs.

Capítulo 5

Conclusão e Boas Práticas

Ao longo deste manual, exploramos os conceitos fundamentais de ponteiros, alocação dinâmica e structs em C, culminando em um projeto prático de rede neural. Lembre-se sempre:

- Inicialize ponteiros.
- Verifique falhas de alocação.
- Libere memória alocada dinamicamente.
- Use ferramentas de depuração (valgrind, gdb).
- Documente seu código, especialmente quando há manipulação complexa de memória.

Dominar esses tópicos é essencial para se tornar um programador C eficiente e para entender o funcionamento interno de sistemas e bibliotecas.

Apêndice A

Funções Auxiliares (Exemplos)

```c
// copiar matriz
Matriz* matriz_copiar(Matriz *origem) {
    Matriz *copia = matriz_criar(origem->linhas, origem->colunas);
    for (int i = 0; i < origem->linhas; i++)
        for (int j = 0; j < origem->colunas; j++)
            copia->dados[i][j] = origem->dados[i][j];
    return copia;
}

// somar bias (vetor coluna) a cada coluna da matriz? Para MLP, bias é somado a cada exemplo.
// Como entrada é coluna, apenas somar elemento a elemento.
void matriz_soma_bias(Matriz *m, Matriz *bias) {
    if (m->linhas != bias->linhas || bias->colunas != 1) return;
    for (int i = 0; i < m->linhas; i++)
        for (int j = 0; j < m->colunas; j++)
            m->dados[i][j] += bias->dados[i][0];
}
```