



linked\_lists\_sedgewick\_style.pdf  
PDF

Este artigo cita uma espécie de polimorfismo de dados em C, usando structs. Poderia me dar uma explicação detalhada do que é esse polimorfismo de dados? Por favor, faça um texto completo explicando isso, desde os conceitos fundamentais até implementações avançadas.

O artigo que você enviou discute uma técnica que o autor chama de "**polimorfismo de dados**" em C, utilizando estruturas (`struct`) para implementar listas encadeadas de maneira genérica. Embora C não tenha suporte nativo para polimorfismo como linguagens orientadas a objetos (ex: C++, Java), é possível simular uma forma de **polimorfismo estrutural** — que podemos entender como *data polymorphism* ou *structural polymorphism* — usando truques com ponteiros e layout de memória.

Vou te explicar em profundidade:



## Conceitos Fundamentais

### ◆ Polimorfismo

O termo "polimorfismo" significa "muitas formas". Em ciência da computação, refere-se à habilidade de tratar diferentes tipos de dados como se fossem o mesmo tipo. Existem diferentes tipos de polimorfismo:

- **Ad-hoc (sobrecarga)**: múltiplas funções com mesmo nome.
- **Paramétrico**: o tipo é um parâmetro (como em C++ templates ou genéricos em Java).
- **Subtipagem (inclusão)**: herança de classes (C++/Java).
- **Polimorfismo estrutural (ou de dados)**: diferentes estruturas que compartilham um "prefixo comum" de campos, permitindo tratá-las como se fossem do mesmo tipo base. **Este é o usado no artigo.**



## Layout de memória em structs

Em C, o layout de campos em uma `struct` é sequencial. Isso significa que:

```
c
struct base {
    int id;
};

struct derivada {
    int id;           // mesmo campo da base
    char nome[50];
};
```

Você pode **converter um ponteiro para struct derivada em ponteiro para struct base** com segurança, pois `id` está no mesmo lugar da memória.

---

## "Polimorfismo" com structs em listas

### A Estrutura Base

O artigo define uma estrutura de nó base para lista duplamente encadeada:

```
c
struct node_s {
    struct node_s *prev, *next;
};
```

Essa estrutura **não tem dados**. É apenas um *conector* para formar a lista.

### Estrutura Derivada (com dados)

Para armazenar dados, define-se um novo tipo que **começa com os campos da base**:

```
c
struct intnode_s {
    struct node_s *prev, *next; // herança "manual"
    int x;                      // dados específicos
};
```

Graças ao layout de memória, um ponteiro para `struct intnode_s` pode ser convertido para `struct node_s *`:

```
c
struct intnode_s item = { ... };
list_add_end(&list, (struct node_s *) &item);
```

Isso permite que **as funções de manipulação de lista trabalhem com struct node\_s**, ignorando o conteúdo específico dos nós.

E depois, para acessar os dados:

```
c
for (struct node_s *i = list.next; i != &list; i = i->next)
{
    printf("%d\n", ((struct intnode_s *) i)->x);
}
```

Esse *type cast* funciona porque os dados estão logo após os campos comuns.

---

## Exemplo Completo

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

```

struct node_s {
    struct node_s *prev, *next;
};

struct intnode_s {
    struct node_s *prev, *next;
    int x;
};

#define INIT_LIST(head) { .prev = (head), .next = (head) }

void list_init(struct node_s *head) {
    head->next = head->prev = head;
}

void list_add_end(struct node_s *element, struct node_s *head) {
    element->prev = head->prev;
    element->next = head;
    head->prev->next = element;
    head->prev = element;
}

#define list_for_each(iter, head) \
    for ((iter) = (head)->next; (iter) != (head); (iter) = (iter)->next)

int main(void) {
    struct node_s list;
    list_init(&list);

    struct intnode_s *n1 = malloc(sizeof *n1);
    n1->x = 10;
    list_add_end((struct node_s *)n1, &list);

    struct intnode_s *n2 = malloc(sizeof *n2);
    n2->x = 20;
    list_add_end((struct node_s *)n2, &list);

    struct node_s *it;
    list_for_each(it, &list) {
        printf("x = %d\n", ((struct intnode_s *)it)->x);
    }

    // Liberação
    list_for_each(it, &list) {
        free(it); // cuidado! aqui assumimos que todos são mallocados
    }

    return 0;
}

```

## ⚠ Considerações Técnicas

### ✅ Vantagens

- Separação clara entre **manipulação de lista** e **conteúdo dos nós**.
- Permite que múltiplos tipos de dados compartilhem a **mesma estrutura de lista**.
- Muito rápido: operações inline com poucas instruções.
- Evita alocação dinâmica dentro das funções da lista.
- Reutiliza o mesmo código para qualquer tipo de dado.

### ⚠ Desvantagens

- É responsabilidade do programador fazer o cast corretamente.
- Não há verificação de tipo em tempo de compilação.
- Pode ser perigoso se o layout da struct derivada for alterado.
- Não é um verdadeiro polimorfismo — é mais uma **simulação via convenção**.

## Comparações com OO

Linguagem OO	C com polimorfismo de dados
Herança com <b>extends</b>	Structs com os mesmos campos iniciais
<b>virtual</b> / métodos	Funções externas que recebem <b>struct *</b>
<b>instanceof</b>	Verificações manuais de tipo / enum
RTTI (Run-Time Info)	Simulado com tags ou campo <b>type</b>

## Implementações Avançadas

- **Campos "tagged union"**: usar um campo `enum type`; para identificar o tipo da struct.
- **Lista heterogênea**: com ponteiros void e um "desempacotador" do tipo adequado.
- **Funções virtuais manuais**: incluir ponteiros de função dentro da struct derivada, como em *vtable*:

```
c
struct intnode_s {
    struct node_s *prev, *next;
    int x;
    void (*print)(struct intnode_s *);
};
```

## Conclusão

O que o autor chamou de *polimorfismo de dados* é uma **técnica idiomática em C** que simula herança e polimorfismo estrutural por meio de organização cuidadosa da memória. Essa técnica permite escrever código genérico de altíssimo desempenho e é usada em muitos projetos reais, incluindo:

- **Kernel do Linux** (estrutura `list_head`)
- **Estruturas internas do glibc**
- **Sistemas embarcados com restrição de alocação**

É uma excelente ferramenta no arsenal de um programador C experiente.

Se desejar, posso expandir com exemplos para listas heterogêneas, simulação de *vtable* ou uso em TADs genéricos.

Faça isso, explique exemplos com listas heterogêneas, simulação de *vtable* e tads genéricos.

Ótimo! Vamos aprofundar essas técnicas avançadas que simulam polimorfismo em C puro — mesmo sem suporte nativo a orientação a objetos. Vamos por partes:

## Polimorfismo de Dados : 1

### ✓ Objetivo

Permitir que diferentes tipos de dados convivam na mesma lista, usando uma base comum.

### 🌱 Abordagem: Tagged Union + Cast

Cada elemento carrega um "rótulo" (**tag**) que diz qual tipo ele representa, e os dados ficam em uma **union**.

### 🔧 Exemplo:

```
c

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

enum tipo_dado {
    INT, DOUBLE, STRING
};

struct node {
    struct node *next;
    enum tipo_dado tag;
    union {
        int i;
        double d;
        char *s;
    } valor;
};

void print_node(const struct node *n) {
    switch (n->tag) {
        case INT:    printf("INT: %d\n", n->valor.i); break;
        case DOUBLE: printf("DOUBLE: %f\n", n->valor.d); break;
        case STRING: printf("STRING: %s\n", n->valor.s); break;
    }
}

void free_node(struct node *n) {
    if (n->tag == STRING)
        free(n->valor.s);
    free(n);
}

int main() {
    struct node *lista = NULL;

    // Cria nó inteiro
    struct node *n1 = malloc(sizeof *n1);
    n1->tag = INT;
    n1->valor.i = 42;
    n1->next = lista;
    lista = n1;

    // Cria nó double
    struct node *n2 = malloc(sizeof *n2);
    n2->tag = DOUBLE;
```

```

n2->valor.d = 3.14;
n2->next = lista;
lista = n2;

// Cria nó string
struct node *n3 = malloc(sizeof *n3);
n3->tag = STRING;
n3->valor.s = strdup("Olá, mundo!");
n3->next = lista;
lista = n3;

// Imprime tudo
for (struct node *p = lista; p; p = p->next)
    print_node(p);

// Libera
while (lista) {
    struct node *tmp = lista;
    lista = lista->next;
    free_node(tmp);
}
}

```

## 2. Simulação de VTables (Funções Virtuais)

### Objetivo

Associar diferentes comportamentos a objetos com interface comum, como métodos virtuais em C++.

### Abordagem: Ponteiros para funções na struct

#### Exemplo:

```

c

#include <stdio.h>

struct animal;

typedef void (*fala_fn)(struct animal *);

struct animal_vtable {
    fala_fn falar;
};

struct animal {
    const struct animal_vtable *vtable;
};

struct cachorro {
    struct animal base;
    char nome[32];
};

struct gato {
    struct animal base;
    int idade;
};

// Implementações dos "métodos"
void cachorro_falar(struct animal *a) {
    struct cachorro *c = (struct cachorro *)a;
    printf("%s: au au!\n", c->nome);
}

```

```

void gato_falar(struct animal *a) {
    struct gato *g = (struct gato *)a;
    printf("gato de %d anos: miau!\n", g->idade);
}

// VTables
const struct animal_vtable cachorro_vt = { .falar = cachorro_falar };
const struct animal_vtable gato_vt = { .falar = gato_falar };

int main() {
    struct cachorro rex = { .base.vtable = &cachorro_vt, .nome = "Rex" };
    struct gato felix = { .base.vtable = &gato_vt, .idade = 3 };

    struct animal *zoo[] = { (struct animal *)&rex, (struct animal *)&felix };

    for (int i = 0; i < 2; i++) {
        zoo[i]->vtable->falar(zoo[i]); // "Método virtual"
    }
}

```

### 3. TAD Genérico com void \*

#### Objetivo

Implementar um Tipo Abstrato de Dado (TAD) que possa armazenar qualquer tipo de valor.

#### Abordagem: void \* + função de cópia/destrutor

#### Exemplo: Pilha Genérica

```

c

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

typedef struct elemento {
    void *dados;
    struct elemento *prox;
} elemento;

typedef struct {
    elemento *topo;
    size_t tam_elemento;
} pilha;

// Cria a pilha
void pilha_init(pilha *p, size_t tam_elemento) {
    p->topo = NULL;
    p->tam_elemento = tam_elemento;
}

// Empilha cópia
void pilha_push(pilha *p, void *valor) {
    elemento *e = malloc(sizeof *e);
    e->dados = malloc(p->tam_elemento);
    memcpy(e->dados, valor, p->tam_elemento);
    e->prox = p->topo;
    p->topo = e;
}

// Desempilha cópia para destino
int pilha_pop(pilha *p, void *dest) {
    if (!p->topo) return 0;
    elemento *e = p->topo;

```

```

memcpy(dest, e->dados, p->tam_elemento);
p->topo = e->prox;
free(e->dados);
free(e);
return 1;
}

// Destroi a pilha
void pilha_destroi(pilha *p) {
    void *tmp = malloc(p->tam_elemento);
    while (pilha_pop(p, tmp));
    free(tmp);
}

int main() {
    pilha p;
    pilha_init(&p, sizeof(int));

    int x = 10;
    pilha_push(&p, &x);

    x = 20;
    pilha_push(&p, &x);

    while (pilha_pop(&p, &x))
        printf("Desempilhado: %d\n", x);

    pilha_destroi(&p);
}

```

Você pode usar `pilha_init(&p, sizeof(T))` para qualquer tipo de `T`.

## Conclusão

Essas técnicas avançadas permitem que o C, mesmo sem orientação a objetos, implemente:

Técnica	Simula O quê?	Ferramenta em C
Structs com prefixo comum	Herança estrutural	Casts e layout de memória
Ponteiros de função em struct	Métodos virtuais	VTables
Tagged union + enum	Objetos heterogêneos	Union + campo <b>tag</b>
TAD genérico com <b>void *</b>	Templates ou genéricos	Tamanho e cópia dinâmica

Essas práticas são comuns em sistemas como o kernel do Linux, bibliotecas gráficas como GTK, e jogos em C. São indispensáveis para construção de bibliotecas reutilizáveis e abstratas em C.

Se quiser, posso organizar tudo isso num mini-framework didático de listas ou pilhas genéricas e heterogêneas, pronto para ser usado com testes e exemplos. Deseja isso?

Sim, organize tudo em um framework didático de listas e pilhas genéricas e heterogêneas.

Perfeito. Vamos construir um pequeno **framework didático** em C, voltado para o ensino de:

- Listas encadeadas heterogêneas com tagged unions
- Pilha genérica baseada em `void *`
- Simulação de vtables com ponteiros para função



## Estrutura de Arquivos

makefile

```
generic_ds/
├── include/
│   ├── gstack.h           # Pilha genérica
│   ├── gheterolist.h      # Lista heterogênea
│   └── gobject.h          # Interface com vtable
└── src/
    ├── gstack.c
    ├── gheterolist.c
    └── main.c
```

## include/gstack.h — Pilha Genérica

```
c

#ifndef GSTACK_H
#define GSTACK_H

#include <stddef.h>

typedef struct gstack gstack;

gstack *gstack_create(size_t elem_size);
void gstack_push(gstack *s, const void *elem);
int gstack_pop(gstack *s, void *out_elem);
void gstack_destroy(gstack *s);

#endif
```

## src/gstack.c

```
c

#include <stdlib.h>
#include <string.h>
#include "gstack.h"

typedef struct gnode {
    void *data;
    struct gnode *next;
} gnode;

struct gstack {
    size_t elem_size;
    gnode *top;
};

gstack *gstack_create(size_t elem_size) {
    gstack *s = malloc(sizeof *s);
    s->elem_size = elem_size;
    s->top = NULL;
    return s;
}

void gstack_push(gstack *s, const void *elem) {
    gnode *n = malloc(sizeof *n);
    n->data = malloc(s->elem_size);
    memcpy(n->data, elem, s->elem_size);
    n->next = s->top;
```

```

    s->top = n;
}

int gstack_pop(gstack *s, void *out_elem) {
    if (!s->top) return 0;
    gnode *n = s->top;
    memcpy(out_elem, n->data, s->elem_size);
    s->top = n->next;
    free(n->data);
    free(n);
    return 1;
}

void gstack_destroy(gstack *s) {
    void *tmp = malloc(s->elem_size);
    while (gstack_pop(s, tmp));
    free(tmp);
    free(s);
}

```

## include/gheterolist.h — Lista Heterogênea

```

c

#ifndef GHETEROLIST_H
#define GHETEROLIST_H

enum tipo_dado {
    G_INT, G_DOUBLE, G_STRING
};

typedef struct gnode_hetero {
    enum tipo_dado tipo;
    union {
        int i;
        double d;
        char *s;
    } valor;
    struct gnode_hetero *prox;
} gnode_hetero;

void list_add_int(gnode_hetero **lista, int x);
void list_add_double(gnode_hetero **lista, double x);
void list_add_string(gnode_hetero **lista, const char *s);
void list_print(gnode_hetero *lista);
void list_destroy(gnode_hetero *lista);

#endif

```

## src/gheterolist.c

```

c

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

void list_add_int(gnode_hetero **lista, int x) {
    gnode_hetero *n = malloc(sizeof *n);
    n->tipo = G_INT;
    n->valor.i = x;
    n->prox = *lista;
    *lista = n;
}

```

```

}

void list_add_double(gnode_hetero **lista, double x) {
    gnode_hetero *n = malloc(sizeof *n);
    n->tipo = G_DOUBLE;
    n->valor.d = x;
    n->prox = *lista;
    *lista = n;
}

void list_add_string(gnode_hetero **lista, const char *s) {
    gnode_hetero *n = malloc(sizeof *n);
    n->tipo = G_STRING;
    n->valor.s = strdup(s);
    n->prox = *lista;
    *lista = n;
}

void list_print(gnode_hetero *lista) {
    for (; lista; lista = lista->prox) {
        switch (lista->tipo) {
            case G_INT: printf("INT: %d\n", lista->valor.i); break;
            case G_DOUBLE: printf("DOUBLE: %f\n", lista->valor.d); break;
            case G_STRING: printf("STRING: %s\n", lista->valor.s); break;
        }
    }
}

void list_destroy(gnode_hetero *lista) {
    while (lista) {
        gnode_hetero *tmp = lista;
        lista = lista->prox;
        if (tmp->tipo == G_STRING) free(tmp->valor.s);
        free(tmp);
    }
}

```

## include/gobject.h — Objetos com VTables

```

c

#ifndef GOBJECT_H
#define GOBJECT_H

typedef struct gobject gobject;
typedef void (*print_fn)(gobject *);

struct gobject {
    const struct gobject_vtable *vtable;
};

struct gobject_vtable {
    print_fn print;
};

typedef struct {
    gobject base;
    int x;
} objeto_inteiro;

typedef struct {
    gobject base;
    double d;
} objeto_double;

extern const struct gobject_vtable vtable_inteiro;
extern const struct gobject_vtable vtable_double;

```

```
#endif
```

## src/main.c — Demonstração Geral

```
c

#include <stdio.h>
#include <stdlib.h>
#include "gstack.h"
#include "gheterolist.h"
#include "gobject.h"

// Implementações das vtables
void print_int(gobject *obj) {
    objeto_inteiro *o = (objeto_inteiro *)obj;
    printf("Inteiro: %d\n", o->x);
}

void print_double(gobject *obj) {
    objeto_double *o = (objeto_double *)obj;
    printf("Double: %f\n", o->d);
}

const struct gobject_vtable vtable_inteiro = { .print = print_int };
const struct gobject_vtable vtable_double = { .print = print_double };

int main() {
    printf("=== Pilha Genérica ===\n");
    gstack *p = gstack_create(sizeof(double));
    double x = 1.1, y = 2.2;
    gstack_push(p, &x);
    gstack_push(p, &y);
    while (gstack_pop(p, &x))
        printf("Pop: %f\n", x);
    gstack_destroy(p);

    printf("\n=== Lista Heterogênea ===\n");
    gnode_hetero *lista = NULL;
    list_add_int(&lista, 42);
    list_add_string(&lista, "Olá");
    list_add_double(&lista, 3.14);
    list_print(lista);
    list_destroy(lista);

    printf("\n=== Objetos com VTables ===\n");
    objeto_inteiro o1 = { .base.vtable = &vtable_inteiro, .x = 10 };
    objeto_double o2 = { .base.vtable = &vtable_double, .d = 2.71 };

    gobject *objs[] = { (gobject *)&o1, (gobject *)&o2 };
    for (int i = 0; i < 2; i++)
        objs[i]->vtable->print(objs[i]);
}
```

## Compilação (exemplo com gcc):

```
bash
```

```
gcc -Wall -Iinclude -o main src/*.c
```

## Conclusão

Esse framework didático cobre:

- **TADs genéricos com `void *`** (pilha)
- **Listas heterogêneas com `union` e `tag`**
- **Simulação de objetos e métodos com `vtables`**

Se quiser, posso empacotar isso como um projeto Git completo ou converter para uma biblioteca com Makefile e testes. Deseja isso?

Em que livros bem conceituados eu posso estudar sobre esses conceitos avançados?

Excelente pergunta. Embora essas técnicas não sejam todas formalizadas como parte da linguagem C padrão (C89/C99/C11/C17), elas são amplamente utilizadas e ensinadas em materiais avançados de sistemas, compiladores e programação genérica em C. Abaixo segue uma curadoria comentada com **livros consagrados, capítulos relevantes** e o tipo de conteúdo que cada um cobre:

## 1. The C Programming Language – Brian W. Kernighan & Dennis M. Ritchie

- 📌 **Clássico absoluto**, base da linguagem.
- 🔍 Explica a estrutura de memória das `struct`, ponteiros, `void *`, `union` e `typedef`, que são base para os conceitos que exploramos.
- ! Não cobre diretamente listas genéricas nem `vtables`, mas é **essencial para domínio do C puro**.

## 2. C Interfaces and Implementations – David R. Hanson

- 💡 **Livro essencial para entender como construir bibliotecas modulares e abstratas em C**.
- 📦 Ensina a construir TADs genéricos e abstratos (pilhas, filas, conjuntos, mapas).
- ⚙️ Usa técnicas com `void *`, separação entre interface `.h` e implementação `.c`, além de simular encapsulamento.
- ✅ Explica como construir *ADT-safe* abstrações reutilizáveis em C.

## 3. Advanced C Programming – Peter van der Linden

- 🦋 Foca em truques avançados e o que acontece "por baixo dos panos".
- Incluí tópicos como layouts de memória, ponteiros genéricos, alinhamento, conversão de tipos, simulações de OO em C.
- ⚠️ Livro com explicações muito didáticas e exemplos surpreendentes.

## 4. Linux Kernel Development – Robert Love

- 🛠 Embora voltado para o kernel do Linux, este livro é excelente para aprender estruturas como `list_head` (listas duplamente encadeadas com polimorfismo estrutural).
  - Mostra como o Linux usa técnicas de **container\_of**, structs com cabeçalho comum e macros para **listas genéricas e heterogêneas**.
  - 🖥 Ideal para ver essas técnicas em uso real de alta performance.
- 

## 5. Design Patterns in C – Adam Tornhill (ou sites baseados no livro de John Lakos, etc.)

- Aplica ideias de design patterns (como estratégia, estado, vtable) em C puro, usando ponteiros de função e estrutura organizacional modular.
  - Ajuda muito a transpor conceitos OO para C, com foco em **flexibilidade e extensibilidade**.
- 

## 6. The Art of Unix Programming – Eric S. Raymond

- Embora mais filosófico e histórico, mostra **como o C foi usado em grandes projetos Unix** com práticas como encapsulamento manual, simulação de objetos, etc.
  - 🎓 Inspira boas práticas e organização de código robusto com C puro.
- 

## 7. Algorithms in C – Robert Sedgewick


- Livro citado no artigo que você mandou.
  - Inclui implementações com **sentinelas, listas, árvores, filas**, etc., todas em C.
  - Importante para entender como a estrutura de dados pode ser implementada de forma genérica e robusta.
  - Excelente referência para entender **listas circulares com sentinela**, como a descrita no seu PDF.
- 

## 8. 21st Century C – Ben Klemens

- Moderniza a linguagem C com dicas contemporâneas.
  - Traz boas práticas, compatibilidade com C++ e exemplos de *design modular*, ponteiros de função, manipulação de dados genéricos, etc.
  - 🏠 Ideal para quem já conhece bem C e quer aplicar com mais segurança em projetos grandes e modernos.
- 

## Extras Online

- 📄 *Linux kernel source code* (`include/linux/list.h`): exemplo consagrado de "polimorfismo estrutural" real.

-  *glibc source code*: muitos exemplos de uso de `void *` e estruturas abstratas.

## Sugestão de Roteiro de Estudo

1. **K&R (Kernighan & Ritchie)** – dominar o básico da linguagem.
2. **Hanson** – aprender TADs genéricos e encapsulamento.
3. **van der Linden** – ponteiros, layout e hacks avançados.
4. **Sedgewick** – implementação eficiente de estruturas de dados.
5. **Love + código do kernel** – técnicas de alto desempenho em sistemas reais.
6. **21st Century C** – modernização, compatibilidade, modularidade.

Se desejar, posso montar um plano de estudos temático com capítulos, exercícios e projetos progressivos focados em estruturas genéricas e polimorfismo em C. Deseja isso também?

Sim, monte esse plano de estudos temático avançado com capítulos, exercícios e projetos focados em estruturas genéricas e polimorfismo em C.

Perfeito. A seguir, apresento um **plano de estudos temático avançado** com foco em **estruturas genéricas e polimorfismo em C**, organizado em módulos progressivos. O plano é prático e teórico, com indicações de leitura, exercícios e projetos.

## Plano de Estudos Avançado: Estruturas Genéricas e Polimorfismo em C

### Objetivo Geral

Desenvolver domínio completo sobre técnicas avançadas em C puro para implementar estruturas genéricas, listas heterogêneas, polimorfismo de dados e simulação de orientação a objetos.

## Módulo 1 – Fundamentos Técnicos da Linguagem

### Tópicos

- Layout de memória em structs
- Ponteiros, `void *`, `union`
- Casting seguro e type punning

### Leitura

- **The C Programming Language** – Capítulos 5 e 6

- **Advanced C Programming** – Capítulo 3 (“Tricks com ponteiros e unions”)

## Exercícios

- Crie uma **union** com três tipos e experimente acessá-la de formas diferentes.
  - Escreva uma função que recebe **void \*** e imprime qualquer tipo básico (usando enum **tag**).
- 

## Módulo 2 – Pilhas e Filas Genéricas com **void \***

### Tópicos

- Alocação dinâmica com cópia de elementos
- Uso de **malloc**, **memcpy** e **sizeof**
- Abstração de TAD via ponteiros

### Leitura

- **C Interfaces and Implementations** – Capítulos 1 a 3
- **21st Century C** – Capítulo 7 (Generic Design)

## Exercícios

- Implemente uma pilha genérica (como **gstack**) que armazena inteiros, doubles e structs.
- Adicione funções para imprimir e esvaziar.

## Projeto 1

- **TAD Genérico de Fila Circular com Buffer Dinâmico**
    - Interface **.h** com funções **enqueue**, **dequeue**, **peek**.
    - Suporte a qualquer tipo (**void \***, com **elem\_size**).
- 

## Módulo 3 – Listas Heterogêneas com Tagged Union

### Tópicos

- **union**, **enum**, casting e segurança
- Implementação de listas com dados de múltiplos tipos
- Impressão e liberação correta de memória

### Leitura

- **Advanced C Programming** – Capítulo 4
- Código fonte da **glibc** ou **glib** (estruturas com **GValue**)



## Exercícios

- Implemente uma lista que aceite `int`, `char *`, `double` usando `enum` + `union`.
- Implemente função de busca por tipo e valor.

## Projeto 2

- **Lista Heterogênea para Eventos de Log**
    - Eventos podem ser `timestamp`, `mensagem`, `código de erro`, etc.
    - Iteração, filtragem e exportação.
- 

## Módulo 4 – Polimorfismo com Structs e Campos Iniciais Comuns

### Tópicos

- Simulação de herança via struct "base"
- Conversão entre tipos por layout comum
- Listas duplamente encadeadas polimórficas

### Leitura

- **Algorithms in C** – Capítulo de listas encadeadas com sentinela (Sedgewick)
- Código-fonte do Kernel Linux (`list_head`)

## Exercícios

- Implemente uma lista com struct base `node_s` e structs derivadas com diferentes payloads.
- Use casting de `node_s *` para acessar os campos reais.

## Projeto 3

- **Gerenciador de Objetos Polimórficos**
    - Lista encadeada com objetos de diferentes tipos.
    - Cada nó implementa função `print()` e `free()`.
- 

## Módulo 5 – Simulação de VTables e Métodos Virtuais

### Tópicos

- Struct de função virtual (vtable)
- Ponteiros para função como comportamento
- Polimorfismo em tempo de execução

## Leitura

- **C Interfaces and Implementations** – Capítulo 7
- **Design Patterns in C** (ou materiais online)

## Exercícios

- Implemente `struct animal` com função virtual `falar()`.
- Faça gato e cachorro implementarem diferentes comportamentos.

## Projeto 4

- **Sistema de Plugins em C**
  - Cada plugin implementa uma função virtual `executar()`.
  - Lista de objetos `plugin_s`, cada um com sua `vtable`.

---

# Módulo 6 – Projeto Final: Framework de Estruturas Genéricas

## Desafio

Construir um pequeno framework de estruturas de dados genéricas em C com:




- Pilha e fila genéricas (`void *`)
- Lista heterogênea com `enum + union`
- Lista polimórfica com `struct base`
- Simulação de orientação a objetos via `vtables`
- Interface e implementação separadas

## Entregáveis

- Biblioteca compilável com headers e `.c` files
- Testes unitários (simples, com `assert`)
- Exemplo de uso para cada estrutura

---

## Extras e Expansão

-  Reimplementação de padrões GoF (comportamento) em C.
-  Benchmarks de eficiência com listas Linux-style vs listas manuais.
-  Conversão para C++ com herança real e comparação de performance.

---

Se quiser, posso empacotar esse plano num arquivo `.pdf` completo, ou criar uma estrutura de repositório Git com os exercícios e projetos já organizados. Gostaria disso?