■ Tema 5: Argo (JSON Parser)

O objetivo é analisar (parse) uma string JSON de um arquivo em uma estrutura de dados `json` em C. Isso envolve `stdio`, gerenciamento de memória dinâmica (`malloc`/`realloc`) e recursão.

Funções-Chave: `malloc`, `realloc`, `free`, `isdigit`, `getc`, `ungetc`. (O `peek` é fornecido usando `getc`/`ungetc`).

Leitura Segura (Foco: `peek`, `accept`, `expect`)

- **Objetivo:** Entender as ferramentas de `stdio` para um parser.

Explicação:

- `getc(stream)`: Consome e retorna o próximo caractere.
- `ungetc(c, stream)`: Devolve `c` ao fluxo para ser lido novamente.
- `peek(stream)`: Lê um caractere sem consumi-lo (usando `getc` e `ungetc`).
- `accept(stream, c)`: Consome `c` se for o próximo caractere.
- `expect(stream, c)`: Exige que `c` seja o próximo caractere.

- **Dica:** Essas são suas ferramentas básicas de navegação.

O Roteador (Foco: `parse_value`)

- **Objetivo:** Criar a função que decide que tipo de JSON está por vir.
- **Dica:** A `parse_value` é o coração da recursão.

Tarefa: Implemente `parse_value`. Use `peek(stream)` para olhar o próximo caractere:

- Se for `"`: chame `parse_string()`.
- Se for `{`: chame `parse_map()`.
- Se for `-` ou `isdigit`: chame `parse_integer()`.
- Caso contrário: chame `unexpected(stream)` e retorne erro.

Analisando `parse_integer`

- **Objetivo:** Converter uma string de dígitos em um `int`.
- **Dica:** O código já está lá, mas entenda-o.

Tarefa:

1. Verifique um sinal opcional de `-`.
2. Verifique se o próximo caractere *é* um dígito (se não, erro).

Em um loop `while (isdigit(peek(stream)))`:

- `val = val * 10 + (getc(stream) - '0');` (a lógica `atoi`).

4. Atribua `dst->type = INTEGER` e `dst->integer = val * sign`.

Analisando `parse_string` (Parte 1: O Básico)

- **Objetivo:** Ler uma string simples `"hello"`.
- **Dica:** `expect(stream, '"')`. Em seguida, leia caracteres em um loop até o próximo `"`.
- **Teste:** Analise `"hello"`.

Analisando `parse_string` (Parte 2: Memória Dinâmica)

- **Objetivo:** Lidar com strings de qualquer tamanho.
- **Função-Chave:** `malloc`, `realloc`.
- **Dica:** Você não sabe o tamanho da string com antecedência.

Tarefa:

1. `malloc` um buffer inicial (ex: `cap = 16`).
2. Dentro do loop de leitura, verifique `if (len + 1 >= cap)`.
3. Se estiver cheio, dobre o `cap` e chame `buf = realloc(buf, cap);`.

- **Teste:** Analise uma string JSON muito longa para forçar o `realloc`.

Analisando `parse_string` (Parte 3: Escapes)

- **Objetivo:** Lidar com `"` e `\`.

Dica: Dentro do loop de leitura, se `c == '\'`:

1. Pegue o *próximo* caractere (`c = getc(stream);`).
2. Verifique se `c` é `"` ou `\`. Se não for, é um erro.
3. Armazene `c` no buffer.

- **Teste:** Analise `"hello\"world""`. A string resultante deve ser `hello"world`.

Analisando `parse_map` (Parte 1: Mapa Vazio)

- **Objetivo:** Lidar com `{}`.

Dica:

1. `expect(stream, '{')`.
2. `if (accept(stream, '}'))`: é um mapa vazio.
3. Defina `dst->type = MAP`, `.size = 0`, `.data = NULL`.

- **Teste:** Analise `{}`.

Analisando `parse_map` (Parte 2: Um Par Chave-Valor)

- **Objetivo:** Lidar com `{"key": "value"}`.
- **Dica:** Se não for `}`, então você *deve* estar em um loop `for (;;)` para ler pares.

Tarefa:

1. `malloc` um array inicial de `pair` (ex: `cap = 4`).
2. Chame `parse_string(&tmp_key)` (a chave *deve* ser uma string).
3. `expect(stream, ':')`.
4. Chame `parse_value(&tmp_val)` (o valor pode ser *qualquer* tipo de JSON).
5. Armazene: `arr[count].key = tmp_key.string;`, `arr[count].value = tmp_val;`, `count++`.

- **Teste:** Analise `{"id": 123}`.

Analisando `parse_map` (Parte 3: Múltiplos Pares)

- **Objetivo:** Lidar com `{"a": 1, "b": 2}`.
- **Dica:** O loop `for (;;)` no Exercício 8 continua.

Tarefa:

1. Após salvar o par, verifique `if (accept(stream, ','))`. Se sim, `continue` o loop para ler o próximo par.
2. Se não, `break` o loop (esperando por `}`).
3. **Importante:** Dentro do loop, antes de ler um novo par, verifique se o array está cheio (`if (count == cap)`) e use `realloc` para dobrar seu tamanho.

- **Teste:** Analise `{"a":1, "b":"dois", "c":{}}`.

Exercício Composto: Gerenciamento de Memória em Erros

- **Objetivo:** Entender o bloco `err:` em `parse_map`.
- **Dica:** O `parse_map` aloca memória para o `arr`, para `tmp_key.string`, e `parse_value` aloca memória para `tmp_val`.
- **Provocação:** Se `expect(stream, ':')` falhar após `parse_string(&tmp_key)` ter sucesso, o que acontece com a memória alocada para `tmp_key.string`? (É um vazamento de memória).
- **Tarefa:** Analise o bloco `err:`. Ele percorre todos os pares (`arr[i]`) que foram alocados *até agora* e chama `free(arr[i].key)` e `free_json(arr[i].value)` (que é recursiva!), e finalmente `free(arr)`. Estude esta lógica de limpeza; ela é crucial para um parser robusto.
- **Teste:** Analise `{"key": "value" , }` (vírgula extra). Verifique se o parser falha e se (usando `valgrind`) não há vazamentos.