Cheat Sheet C - Parte 1

Algoritmos e Estruturas de Dados - 2025/02

Última atualização: 19/09/2025 08:37

Este material foi inspirado no **Cheat Sheet Java - Parte 1**, de Igor Montagner.

Variáveis e Atribuições

No C, não existe inferência automática de tipos. Toda variável e todo valor de retorno precisam ter o tipo declarado explicitamente. A tabela a seguir mostra as equivalências de alguns tipos.

Pseudo código	C
int	int
float	float
long int	long
character	char
string	char[]/char*

Atenção

1. Divisão de **int** resultará sempre em um **int** (como a operação // em *Python*).
2. Operações que misturam **int** e **float** fazem conversão para **float**.
3. **Strings** em C são arrays de caracteres terminados obrigatoriamente com o caractere nulo **'\0'**. Sem esse terminador, funções de string não sabem onde a string acaba.
4. O tipo **char*** é um ponteiro para o primeiro caractere de uma string, e pode apontar para memória estática (literal de string) ou dinâmica (alocada com **malloc**).

Algoritmos

Em C, cada função precisa ter um nome e os tipos de **entrada** (argumentos) e **saída** (return) declarados de forma **explícita**. No exemplo abaixo, criamos uma função que recebe dois inteiros e devolve sua soma. Note que declaramos os tipos das variáveis em **Input** e o tipo da saída em **Output** e que descrevemos em **Result** o que o algoritmo faz.

Algorithm 1: Exemplo1

Result: Soma as variáveis a e b
Input : int a, int b
Output: int

1 int $c \leftarrow a + b$
2 **return** c

```c
int soma(int a, int b) {
    int c = a + b;
    return c;
}
```

Condicionais e Loops

Em C, blocos de código dentro de condicionais e loops são delimitados por chaves `{ }`.
No pseudocódigo abaixo, usamos indentação e linhas verticais para mostrar o conteúdo de cada bloco.

Algorithm 2: Exemplo2

Result: Calcula $\sum_i^n 2^i(-1)^i$
Input : int i, int n
Output: int

```
1  int total ← 0
2  int temp ← 2^i
3  while i <= n do
4  |  if i%2 = 0 then
5  |  |  total ← total + temp
6  |  else
7  |  |  total ← total − temp
8  |  end
9  |  i ← i + 1
10 |  temp ← temp × 2
11 end
12 return total
```

```c
int calcula(int i, int n) {
    int total = 0;
    int temp = pow(2, i);

    while (i <= n) {
        if (i % 2 == 0) {
            total += temp;
        } else {
            total -= temp;
        }
        i++;
        temp *= 2;
    }
    return total;
}
```

Loops `for` em C permitem inicializar uma variável, definir a condição de parada e indicar como ela será atualizada a cada iteração.

Algorithm 3: Exemplo de for

Result: Exemplo simples de loop for

```
1 for int i ← 5 to 10 do
2 |  ....
3 end
```

```c
// exemplo de for
for (int i = 5; i < 10; i++) {
    // ...
}
```

Ponteiros

Em C, um **ponteiro** é uma variável que armazena o endereço de memória de outra variável. Usamos o operador **&** para obter o endereço e o operador `*` para acessar ou modificar o valor armazenado nesse **endereço**. Ponteiros permitem manipular diretamente valores na memória e são fundamentais em C para trabalhar com struct, vetores e funções.

Algorithm 4: Exemplo2

Result: Altera o valor referenciado por p
Input : **int*** p
Output: **void**

```
1 *p ← 20
```

```c
// Função que recebe um ponteiro e altera o valor original
void alteraValor(int *p) {
    *p = 20;  // modifica o valor armazenado no endereço
}
```

Alocação estática e dinâmica

- **Estática**: o tamanho do array é fixado no código em tempo de compilação. A memória é reservada automaticamente e liberada quando a função termina.
  Exemplo: `int v[10];`

- **Dinâmica**: o tamanho pode ser decidido em tempo de execução. Para isso usamos `malloc`, que pede memória na heap.
  O operador `sizeof` garante que estamos reservando a quantidade correta de bytes para o `tipo`.
  A memória obtida precisa ser liberada manualmente com `free`.

Exemplos com `malloc`

Algorithm 5: Exemplo de malloc

Result: Alocação dinâmica de memória para um vetor de inteiros e uma string

1 **int** $n \leftarrow 5$
2 **int*** $v \leftarrow malloc(n \times sizeof(int))$
3 **char*** $s \leftarrow malloc(n \times sizeof(char))$
4 **return** v, s

```c
int n = 5;
int *v = malloc(n * sizeof(int));
char *s = malloc(n * sizeof(char));
```

Atenção!

- Sempre use `free` para liberar a memória alocada. Se esquecer, ocorre **vazamento de memória**.
- A memória alocada por `malloc` vem com **lixo**, então **prepare antes de usar** (por exemplo, inicializando com zeros ou valores desejados).

Arrays

Um array em C é um conjunto de elementos do mesmo tipo armazenados de forma contínua na memória.
O tamanho é fixado no momento da criação.
O primeiro elemento tem índice `0` e o elemento `i` é acessado com `A[i]`.
Para criar arrays dinamicamente, usamos `malloc` para reservar a quantidade de memória desejada.

Algorithm 6: Exemplo de média

Result: Calcula média de um array
Input : array **float** A, **int** n
Output: **float**

1 **float** $total \leftarrow 0$
2 **for** $i \leftarrow 0$ ***to*** $n-1$ **do**
3 | $total \leftarrow total + A[i]$
4 **end**
5 **return** $total/n$

```c
float media(float A[], int n) {
    float total = 0;
    for (int i = 0; i < n; i++) {
        total += A[i];
    }
    return total / n;
}
```

Strings em C

Em C uma *string* é representada como um array de `char` terminado pelo caractere especial `'\0'`.
O acesso a cada caractere é feito diretamente com `S[i]`, e o loop percorre até encontrar o `'\0'`.

Algorithm 7: Acesso aos caracteres de uma string

Result: Percorre e processa os caracteres de uma string
Input: **char** $S[]$

1 **int** $i \leftarrow 0$
2 **while** $S[i] \neq' \backslash 0'$ **do**
3 | // faz algo com $S[i]$
4 | $i \leftarrow i + 1$
5 **end**

```c
void percorreString(char S[]) {
    int i = 0;
    while (S[i] != '\0') {
        // faz algo com
        printf("%c\n", S[i]);
        i++;
    }
}
```

Cuidado com `char` em C

- `'a'` representa um único caractere (tipo `char`), sempre entre aspas simples.
- `"a"` representa uma *string*, que na prática é um array de dois caracteres: `'a'` e `'\0'`.

Struct em C

Em C, uma `struct` permite agrupar diferentes variáveis em um único tipo.
Cada campo dentro da `struct` pode ter um nome e um tipo, funcionando como uma "caixa" que guarda vários valores relacionados.
Isso é útil quando queremos representar um objeto que tem várias informações ou quando precisamos retornar vários resultados de uma função.

Algorithm 8: Uso de struct em C

Result: Exemplo de struct com soma e produto
Input : **int** a, **int** b
Output: **struct Resultado** com campos soma e produto

1 **struct Resultado** r
2 $r.soma \leftarrow a + b$
3 $r.produto \leftarrow a \times b$
4 **return** r

```c
// definição da struct
typedef struct {
    int soma;
    int produto;
} Resultado;

// função que retorna a struct
Resultado calcula(int a, int b) {
    Resultado r;
    r.soma = a + b;
    r.produto = a * b;
    return r;
}
```