

C Manual

Amanda Araujo

Resumo

Compilado de comandos básicos e estruturas da linguagem de programação C. O guia *4dummies* que eu precisava para relembrar conceitos, sintaxe e implementações em C, mantendo minha sanidade. Devemos sempre ser o nosso melhor colaborador e acho que a futura eu me agradecerá. É um processo em andamento e contínuo*.

Conteúdo

Básico da linguagem de programação C

1	Linguagem C	3
2	Ambiente de trabalho	3
3	Códigos simples: aspectos da linguagem	4
4	Variáveis	5
5	Operadores	7
6	Estruturas de controle	8
6.1	Estruturas condicionais	9
6.1.1	if else	9
6.1.2	Switch case	9
6.2	Estruturas de repetição: <i>Loops</i>	9
6.2.1	For	9
6.2.2	While	10
7	Vetores e matrizes	10
8	Ponteiros	11
9	Funções	12
9.1	Localização das funções no código-fonte	13
9.2	Passagem de parâmetros: por valor <i>vs.</i> por referência	14
9.3	Retorno de múltiplos valores em uma função	16

*Códigos disponíveis no GitHub nos repositórios: [lab-icc-I](#) e [estrutura-dados-I](#).

10 Estruturas (Struct)	16
10.1 Criando structs: typedef	17
10.2 Acessando e alterando campos de structs	18
<i>Estrutura de Dados</i>	
11 Funções recursivas	19
11.1 Recursão	19
11.2 O que acontece quando uma função é chamada e retornada?	20
11.3 Iteração <i>vs.</i> Recursão	21
12 Análise de algoritmos	21
13 Listas lineares	22
13.1 Listas lineares sequenciais	23
13.2 Listas ligadas (ou encadeadas)	24
13.2.1 Implementação estática	24
13.2.2 Implementação dinâmica	28
13.3 Listas dinâmicas com nó cabeça e circularidade	29
13.4 Listas dinâmicas duplamente encadeadas com nó cabeça e circularidade	29
13.5 Filas	29
13.5.1 Implementação estática	29
13.5.2 Implementação dinâmica	29
13.5.3 Filas com prioridade	29
13.6 Deques	29
13.7 Pilhas	29
13.7.1 Implementação estática	29
13.7.2 Implementação dinâmica	29
13.7.3 Representação de duas pilhas em um único vetor	29
13.8 Matrizes esparsas: representação por listas cruzadas	29
13.9 Listas generalizadas	29
A Ambiente de trabalho: minha máquina	30
B Comandos terminal Linux	30
B.1 Diretório de trabalho	30
B.2 Compilar, rodar e debugar	31

Parte I

Básico da linguagem de programação C

1 Linguagem C

A linguagem de programação C foi criada em 1972 por Dennis Ritchie na empresa AT&T Bell Labs. Características gerais de C:

- Compilada (*vs. interpretada*);
- Estruturada;
- Imperativa;
- Procedural;
- Nível baixo/médio;
- *Case sensitive*: interpreta como diferentes letras maiúsculas de minúsculas.

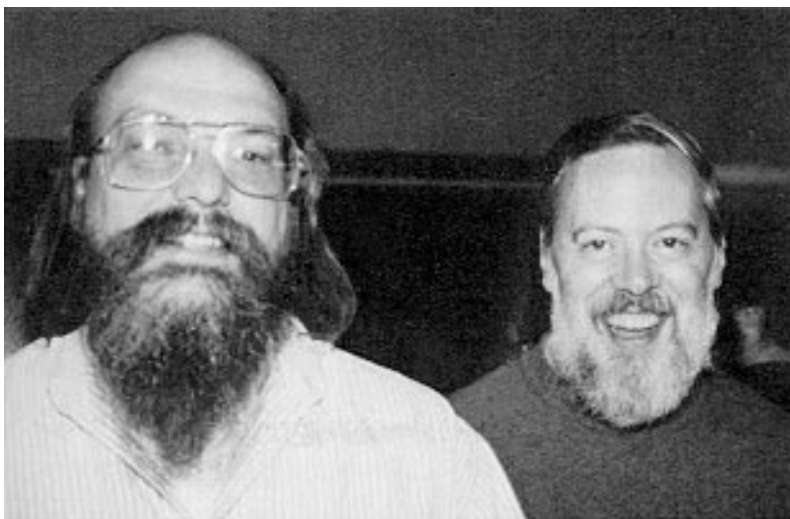


Figura 1: Ken Thompson e Dennis Ritchie (da esquerda para direita), os criadores das linguagens B e C, respectivamente.

Vantagens e desvantagens

A linguagem permite acesso direto à memória e aos recursos do sistema, além de ser capaz de gerar programas extremamente rápidos em tempo de execução.

Uma das principais características da linguagem C é que ela é fortemente tipada, o que significa que o tipo de dado que está sendo manipulado deve ser especificado de forma explícita.

2 Ambiente de trabalho

First things first: *set up* do ambiente de trabalho. Diferentemente de linguagens interpretáveis que rodam diretamente, a linguagem C precisa ser compilada. Ou seja,

mais uma etapa é adicionada, e para rodar códigos em C na sua máquina é preciso baixar e instalar um compilador. O código em si pode ser escrito em qualquer editor de texto, até mesmo no bloco de notas (não recomendável), desde que possua a extensão `.c`. O ideal é usar algum ambiente de desenvolvimento integrado (IDE), como o Visual Studio Code (VS Code), uma aplicação de software que ajuda os programadores a desenvolver código de software de maneira eficiente

Extra: *debugger*. Ao programar coisas mais complicadas em C, especialmente ao mexer com ponteiros e lidar com estrutura de dados, podem surgir erros mais difíceis de serem identificados. Nessas horas, um programa que te ajude a indicar e localizar os erros é essencial, e para isso temos os depuradores (*debugger*).

3 Códigos simples: aspectos da linguagem

Começando pelo clássico *Hello World!*, temos o primeiro exemplo de um código em C:

```
1  #include <stdio.h>
2
3  /* Primeiro código em C */
4
5  int main()
6  {
7      printf("Hello World!\n");
8      return 0;
9  }
```

Nele é possível notar alguns aspectos gerais da linguagem C, presentes em todos os seus códigos:

- **#include**: todo código em C começa tendo como cabeçalho a importação de bibliotecas que contém funções úteis, por meio do comando *include*. As bibliotecas a serem carregadas devem ser enunciadas dentro de `<>`. O padrão é a importação da biblioteca *stdio.h*, que contém as funções básicas de C, como *printf()* e *getchar()*.
- **Tipo da variável**: em C, toda variável e função deve ser declarada explicitamente com seu tipo: *int*, *double*, *char*, etc. Nesse caso, temos o tipo *int*, que indica um inteiro.
- **int main()**: indica ao compilador que o programa tem uma função chamada *main* - Todo programa em C tem uma função *main*. Ela é o local de início (*entry point*) da execução de um programa em C.
- **Colchetes { }**: indica que tudo dentro deles é parte da função, no caso, da função *main*.
- **Ponto e vírgula**: toda linha de comando em C deve terminar com uma indicação do seu fim, por meio da adição de ponto e vírgula: `;`.

- **return**: toda função em C necessariamente retorna algo e o comando **return** indica o fim da função. Nesse caso, como a **main** é declarada como uma função **int**, ela vai precisar de um inteiro para retornar quando o programa acabar. Um “0” indica que o programa terminou sem erros; qualquer outro número representa problemas.
- **Comentários**: parte não executada do código, essencial para anotar ou comentar o que foi feito. Tudo o que se encontra entre **/* */** é um comentário, lembrar de sempre abrir e fechá-lo (barras duplas **//** também é um indicativo de comentário, no entanto, apenas da linha em questão).

4 Variáveis

Em C, uma variável é um nome personalizado definido pelo usuário ou legível pelo usuário atribuído a um local de memória. Todas as variáveis em C são declaradas ou definidas antes de serem utilizadas. Uma *declaração* indica o tipo de uma variável. Se a declaração produz também armazenamento (se inicia), então é uma *definição*.

Declarando e definindo variáveis em C. Sintaxe:

```
type variableName = value;
```

Tipos de dados/variáveis (e sua respectiva máscara de formatação):

- **char** (%c): caracteres (1 byte)
- **int** (%d): números inteiros (4 bytes/2 bytes*)
- **float** (%f): número real de precisão simples (4 bytes)
- **double** (%lf): número real de precisão dupla (8 bytes)
- **void**: valor vazio, usado principalmente ao definir funções (0 bytes)

```
1 // Create variables
2 int myNum = 15;           // Integer (whole number)
3 float myFloatNum = 5.99;  // Floating point number
4 char myLetter = 'D';      // Character
5
6 // Print variables
7 printf("%d\n", myNum);
8 printf("%f\n", myFloatNum);
9 printf("%c\n", myLetter);
```

OBS: Em C não há um tipo de variável especial hexadecimal, no entanto, para *printar* valores no formato hexadecimal é necessário utilizar a máscara **%x** como um especificador de formato no comando **printf()**. O mesmo vale para a leitura por meio de **scanf()** (**%x**: para alfabeto em letra minúscula a-f; **%X**: para alfabeto em letra maiúscula A-F).

*Depende do ambiente de execução do programa; ambiente 16-bit: 2 bytes, ambiente 32-bit: 4 bytes.

OBS2: A linguagem C não tem um tipo de dado específico para armazenar valores booleanos (V/F). Em C, o valor booleano é representado por um valor inteiro: 0 significa falso e qualquer $\text{int} \neq 0$ significa verdadeiro. Em geral, utiliza-se 1 para representar o valor verdadeiro.

```
1 //To make life easier, C Programmers typically define the terms "
  true" and "false" to have values 1 and 0 respectively.
2
3 //In the old days, this was done using #define:
4 #define true 1
5 #define false 0
6
7 //Today it is better to use const int instead of #define:
8 const int true = 1;
9 const int false = 0;
10
11 typedef int bool;
```

Estendendo um pouco mais... Diferença entre *variáveis* vs. *nomes*

De maneira contrastante, as variáveis em C são fundamentalmente diferentes das variáveis em Python. Python não tem variáveis: *Python has names, not variables*.

Ao declarar uma variável em C (`int x = 2337;`), diferentes passos são executados:

1. Alocação de memória suficiente para o tipo de variável, *p.ex.* um inteiro;
2. Atribuição do valor aquele local de memória;
3. Indicação de que o nome da variável aponta para o valor.

É possível alterar o valor de `x` (variável mutável), sobrescrevendo seu valor anterior no mesmo local de memória, *i.e.* `x` é o local de memória, não somente um nome para ele. Isso pode ser pensado como posse, em certo sentido, `x` possui o local de memória.

Em Python, por outro lado, não há variáveis, apenas nomes. O nome `x` em Python não possui diretamente nenhum endereço de memória - é criado um objeto cujo valor é definido e um nome `x` que lhe é dado -, enquanto a variável `x` em C possui um *slot* estático na memória. Ao se alterar o valor de `x`, o que ocorre em Python é a criação de uma referência a um novo objeto, que não possui o espaço de memória como antes. Além disso, o objeto anterior agora passa a possuir um *ref count* de 0, e será limpo, liberando memória, pelo *garbage collector*. Em resumo, em Python não se atribui valores a variáveis, ao invés disso, nomes são vinculados a referências.

Tudo isso pode parecer um pouco confuso e até mesmo abstrato, no entanto, essas ideias ficam mais claras ao se deparar com conceitos como o de ponteiros, presentes explicitamente em C, mas ausentes em Python. Essas diferenças surgem das propostas de cada linguagem de programação e com qual premissa elas foram criadas. C é tida como linguagem de “baixo nível” - visando tornar manipuláveis aspectos mais ligados ao hardware (memória, por exemplo) e tirar vantagem direta de sua arquitetura, permitindo um controle fino sobre os recursos do sistema -, enquanto Python foi projetada

para ser de “alto nível” - alta abstração, facilitando seu uso por programadores, que passam a estar mais livres para focar na resolução de problemas, ao invés de nos detalhes de implementação de baixo nível.

5 Operadores

Para realizar operações entre as variáveis, existem operadores de diferentes naturezas: aritméticos (1), relacionais (2) e lógicos (3), que representam as operações aritméticas básicas, estabelecem relações/comparações e representam operações baseadas na lógica matemática, respectivamente.

Operação	Operador
Adição	+
Subtração	-
Multiplicação	*
Divisão	/
Resto	%
Incremento	++
Decremento	--

Tabela 1: Operadores aritméticos

Operação	Operador
Igualdade	==
Diferença	!=
Maior	>
Maior ou igual	>=
Menor	<
Menor ou igual	<=

Tabela 2: Operadores relacionais

Operação	Operador
Negação	!
Conjunção (<i>e</i>)	&&
Disjunção (<i>ou</i>)	

Tabela 3: Operadores lógicos

Operador de referência (*): usado para acessar o conteúdo de uma posição de memória, cujo endereço está armazenado em um ponteiro.

Operador de endereço (&): usado para se referir ao endereço da variável valor e não

ao conteúdo da mesma ao se atribuir um endereço a um ponteiro.

Operador ponto (.): acessa um campo (ou membro) de uma *struct*.

Operador seta (->): acessa um campo (ou membro) de uma *struct* por meio de ponteiros. É uma desreferência do ponteiro contido no membro, indica que deve pegar o valor apontado pelo ponteiro no membro; um exemplo de *syntactic sugar**.

Outros tipos de operadores: operadores de manipulação de bits (*bitwise*), operadores de atribuição (= e combinação de = com outros para abreviar expressões), operador série (,), operador de conversão de tipos, *etc.*

Syntactic sugar

Um açúcar sintático (*syntactic sugar*), termo criado por Peter J. Landin em 1964, é uma sintaxe dentro da linguagem de programação cuja finalidade é tornar suas construções mais fáceis de serem lidas e expressas. Ela faz com que o uso da linguagem se torne "mais doce" para o uso humano, permitindo que suas funcionalidades sejam expressas mais claramente, mais concisamente ou, ainda, como um estilo alternativo preferido por alguns.

6 Estruturas de controle

Estruturas de controle são mecanismos que permitem controlar o fluxo de execução de um programa. Elas determinam a ordem em que as instruções são executadas com base em condições ou *loops*. Elas podem ser simples comandos (Figura 2) ou funções (Seção 9).

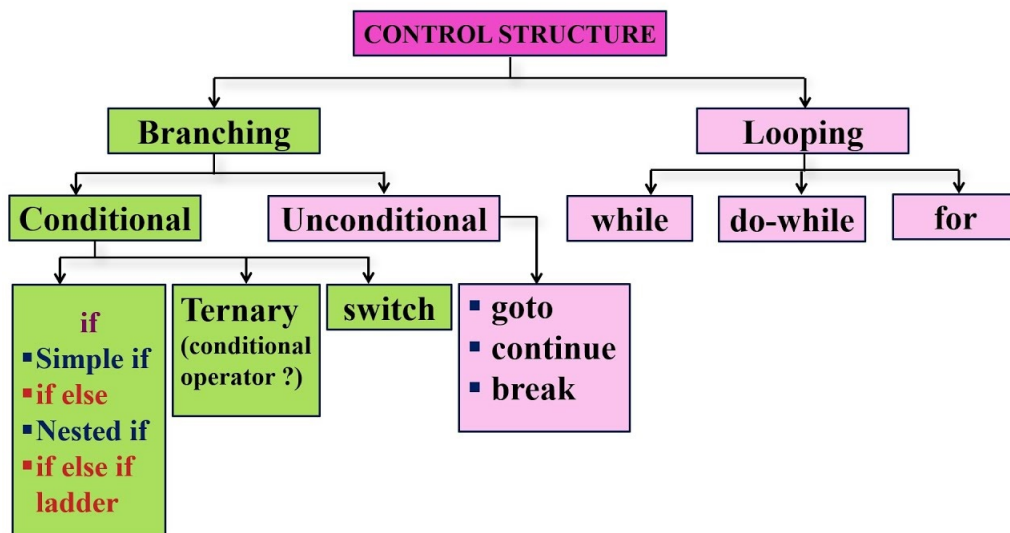


Figura 2: Tipos de estruturas de controle em C.

6.1 Estruturas condicionais

6.1.1 if else

Sintaxe básica:

```
if (condição) {  
    // Código a ser executado se a condição for verdadeira;  
} else {  
    // Código a ser executado se a condição não for verdadeira;  
}
```

6.1.2 Switch case

O comando `switch case` é uma forma de reduzir a complexidade de vários `if & else` encadeados. É muito utilizado, principalmente para uso em estruturas de menu. O conteúdo de uma variável é comparado com um valor constante, e caso a comparação seja verdadeira, um determinado comando é executado.

```
switch (expression)  
{  
    case /* constant-expression */:  
        /* code */  
        break;  
  
    default:  
        break;  
}
```

Atenção: Não é possível declarar uma variável dentro de um caso do `switch`.

Um engano muito comum é as pessoas acharem que o `case` é um bloco de comandos e gera um novo escopo. Na verdade, o `case` é apenas um *label*. Então é apenas um nome para um endereço do código usado para provocar um desvio. Na verdade, um `switch` é apenas um `goto` baseado em um valor. Vale para o `switch`, assim como também não se pode declarar uma variável dentro de um `if`.

6.2 Estruturas de repetição: *Loops*

As estruturas de comando de repetição em C (`for`, `while` e `do-while`) são usadas para criar *loops* que executam um bloco de código repetidamente enquanto uma condição específica for verdadeira.

6.2.1 For

Sintaxe básica:

```
for (inicialização; condição; atualização) {  
    // Código a ser executado repetidamente;  
}
```

6.2.2 While

Sintaxe geral:

```
while (condição) {  
    // Código a ser executado repetidamente;  
}
```

7 Vetores e matrizes

Vetores e matrizes em C são coleções de **variáveis de um mesmo tipo**, acessíveis com um único nome e armazenados contiguamente na memória. Um vetor é uma estrutura de dados indexada, *i.e.* as variáveis armazenadas (itens) são acessadas por meio de um índice próprio. Vetores são matrizes de uma dimensão.

Atenção: Em C o index começa no 0.

OBS: O último elemento de um vetor pode sempre ser acessado pelo índice -1.

Sintaxe da declaração de um vetor/matriz:

```
tipo nomeVetor[TAM];  
tipo nomeMatriz[NLIN] [NCOL];
```

Ao se declarar um vetor/matriz em C, é preciso explicitar, além do tipo de variável a ser armazenada (`int`, `float`, etc.), a quantidade de memória a ser reservada para tal, indicada entre colchetes [].

Preenchimento e acesso aos elementos: realizado com o uso de índices, indicado entre colchetes [] após seu nome:

```
1  int Vetor[5];    // declara um vetor de 5 posicoes  
2  int Matriz[5][3]; // declara uma matriz de 5 linhas e 3 colunas  
3  
4  Vetor[0] = 9; // coloca 9 na primeira posicao do vetor  
5  Vetor[-1] = 30 // coloca 30 na ultima posicao do vetor  
6  Matriz[0][1] = 15; // coloca 15 na 1a linha e 2a coluna da matriz  
7  
8  //Colocar os numeros de 1 a 5 em Vetor  
9  for(int i = 0; i < 5; i++){  
10     Vetor[i] = i + 1;  
11 }
```

Uso de constantes para definir o tamanho de um vetor ou matriz (`#define`):

```
1  #define NLIN 10  
2  #define NCOL 5  
3  int Matriz[NLIN][NCOL];  
4  
5  //Preencher uma matriz com um dado  
6  for(int i = 0; i < NLIN; i++) {  
7      for(int j = 0; j < NCOL; j++) {
```

```

8         Matriz[i][j] = 30;
9     }
10 }

```

É possível já no momento de criação do vetor/matriz passar sua lista de valores, entre colchetes { } (declaração e definição em conjunto). Nesse caso, os colchetes retos [] aparecem não preenchidos:

```
int myNumbers[] = {25, 50, 75, 100};
```

Atenção: Cuidado com os limites de vetores e matrizes! A linguagem C não faz nenhum teste de verificação dos índices usados para acessar os elementos de um vetor. Isto significa que, se estes limites não forem respeitados, resultados indesejados serão obtidos, serão cuspidos valores espúrios sem aviso de erro. Exemplo (vetor de tamanho 5 e *print* de 10 elementos):

```

1  #include <stdio.h>
2
3  int main(){
4
5      int Vetor[5];
6
7      for(int i = 0; i < 5; i++) Vetor[i] = i + 1;
8      for(int i = 0; i < 10; i++) printf("%d ", Vetor[i]);
9
10     return 0;
11 }

```

```

1  1 2 3 4 5 32766 50827776 1247028395 0 0

```

8 Ponteiros

Um ponteiro (ou apontador) é uma variável capaz de armazenar um **endereço de memória** ou o endereço de outra variável. Ponteiros podem apontar para qualquer tipo de variável, portanto temos ponteiros para `int`, `float`, `char`, etc.

Utilidade dos ponteiros: úteis quando uma variável precisa ser acessada em diferentes partes do programa. Situações exemplo:

- Alocação dinâmica de memória;
- Manipulação de arrays;
- Para retornar mais de um valor em uma função;
- Referência para listas, pilhas, árvores e grafos.

O uso de ponteiros permite uma maior eficiência no código.

Sintaxe de declaração de um ponteiro:

```
tipo *nome_Ponteiro;
```

Note que: importantíssimo, é o asterisco * que indica que se trata de um ponteiro.

Após a criação de um ponteiro, é necessário dizer para quem ele aponta, *i.e.* realizar a atribuição do endereço da variável de interesse, de mesmo tipo, ao ponteiro:

```
nome_Ponteiro = &valor;
```

Note que: importantíssimo, é o operador de endereço & que indica que se trata do endereço da variável, e não seu valor.

```
1  #include <stdio.h>
2  void main()
3  {
4      int a;
5      int b;
6      int c;
7      int *ptr; // declara um ponteiro para um inteiro
8                // um ponteiro para uma variavel do tipo inteiro
9      a = 90;
10     b = 2;
11     c = 3;
12     ptr = &a;
13     printf("Valor de ptr: %p, Conteudo de ptr: %d\n", ptr, *ptr);
14     printf("B: %d, C: %d", b, c);
15 }
```

Ponteiros permitem uma manipulação a um nível mais profundo, lidando diretamente com a memória, o que torna a linguagem C altamente flexível e poderosa, mas também pode abrir a porta para erros mais críticos. Embora os ponteiros sejam poderosos e versáteis, eles também podem ser fontes de erros difíceis de depurar, como vazamentos de memória e falhas de segmentação, os infames e rechaçáveis “*memory leak*” & “*segmentation fault*”.

9 Funções

Uma função em um código de programação é uma sub-rotina de um programa. Funções são blocos de código que realizam tarefas específicas e podem ser definidas e chamadas pelo programa principal (`main()`) ou por outras funções. O seu uso permite a separação do programa em partes (blocos) que possam ser logicamente compreendidos de forma isolada, facilitando sua leitura. Além disso, funções permitem o reaproveitamento de códigos e contribuem para evitar repetições de um mesmo trecho de código repetidamente para execução de uma mesma tarefa.

Sintaxe básica:

```
tipo nome_função(parâmetros) {
    //Corpo da função;
    return valorRetorno;
}
```

Exemplo de uma função simples:

```
1  int soma(int a, int b) {  
2      int resultado;  
3      resultado = a + b;  
4      return resultado;  
5  }
```

Elementos de uma função:

- Tipo (**tipo**): uma função sempre apresenta um tipo, referente ao tipo esperado do seu valor de retorno (`int`, `float`, `void` *etc.*).
- Nome da função: nomeia a função e indica como ela será chamada para ser executada futuramente.
- Declaração de parâmetros: entre parêntesis, uma listagem explícita de todos os parâmetros necessários para a execução da função, com seu respectivo tipo. É opcional.
- Colchetes { } : delimitam a função, indica que tudo dentro deles é parte da função.
- Corpo da função: contém as instruções e comandos a serem realizados como tarefa específica da função.
- **return**: toda função em C necessariamente retorna algo (a menos que se trate de uma função do tipo `void`) e o comando **return** indica o fim da função, juntamente com o valor a ser retornado. Uma função em C retorna apenas **um único valor**, porém isso pode ser contornado de maneiras indiretas (Seção 9.3).

Importante: escopo de variáveis, *i.e.* bloco de código onde as variáveis são válidas. Note que é possível declarar variáveis dentro das funções. Nesse caso, elas recebem o nome de **variáveis locais** (em contraposição às variáveis globais). Variáveis locais definidas dentro de uma função *não* são acessíveis em outras funções, mesmo que as variáveis apresentem nomes idênticos.

9.1 Localização das funções no código-fonte

Localização das funções no código-fonte

Via de regra geral:

Toda função deve ser *declarada* antes de ser usada.

Lembrando, declaração é diferente de definição. A declaração envolve anunciar qual o *tipo* da variável ou função, enquanto na definição ocorre a atribuição de um valor. No caso das funções, no momento em que definimos uma função, está implícita sua declaração (`tipo nome_função()`). Como funções são blocos de código independentes, úteis para encurtar e deixar o código principal simples e enxuto, sua definição pode (e deve) ficar localizada fora da `main()`.

Como o lema diz, a linguagem C exige que toda função seja *declarada* antes do seu

uso, de modo que é possível apresentar um código-fonte claro com o bloco correspondente à função `main()` primeiro, seguida da *definição* de todas as funções auxiliares empregadas. Para tal, basta anunciar acima da `main()` as funções a serem usadas, como um pequeno *spoiler* do que virá. Esta declaração é feita através do **protótipo** da função, que nada mais é do que um trecho de código (uma linha) que especifica o nome e os parâmetros da função.

```

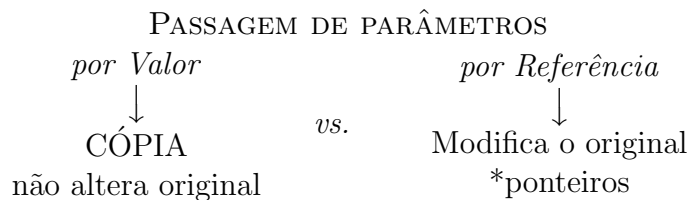
1  #include <stdio.h>
2
3  int SOMA(float a, int b); // Prototipo da funcao SOMA
4
5  int main()
6  {
7      SOMA(16.7,15); // Chamada da funcao SOMA antes de sua
                        // definicao, mas apos sua prototipacao
8
9      return 0;
10 }
11
12 int SOMA(float a, int b) // Definicao da funcao SOMA
13 {
14     float result; // a declaracao de variaveis igual ao que
                    // se faz na funcao main
15     result = a+b;
16     printf("Soma de %f com %d: %6.3f\n", a, b, result);
17     return 0;
18 }

```

9.2 Passagem de parâmetros: por valor *vs.* por referência

Passagem de parâmetros para funções

A finalidade dos parâmetros é fazer a comunicação entre as funções e a função principal. Chamamos de passagem de parâmetros a passagem de valores entre as funções. Existem duas formas de passar parâmetros para uma função:



A passagem de parâmetros *por valor* é o que tem sido feito até então nos exemplos, onde os parâmetros declarados são variáveis. Ao serem usados na função, é feita uma *cópia* do valor da variável, que é usada e modificada, enquanto a variável original se mantém inalterada. Já a passagem de parâmetros *por referência* é feita utilizando *ponteiros*, que contém o endereço de memória da variável e permite a alteração do conteúdo da variável original usando-se esta referência. Ou seja, o valor da variável passada como argumento é modificado não somente no bloco da função, mas afetando todo o código.

Passar por referência é recomendado no caso de tipos complexos, como é o caso de

uma **struct**. Embora a passagem da cópia gaste mais memória, esta permite um acesso mais rápido dos valores, atualmente priorizado, dado que normalmente memória não é mais um problema.

Exemplo de passagem de parâmetros por valor e por referência:

```
1  #include <stdio.h>
2
3  void soma(int a);      // Prototipo Passagem por valor
4  void somaref(int *a);  // Prototipo Passagem por referencia
5
6  int main(void)
7  {
8      int numero;
9      printf("Digite um numero: ");
10     scanf("%d", &numero);
11
12     printf("O numero digitado: %d \n", numero);
13
14     soma(numero);      //chamada da funcao
15     printf("Agora o numero vale (por valor): %d \n", numero);
16
17     somaref(&numero);  //chamada da funcao com ponteiro como
18     parametro
19     printf("Agora o numero vale (por referencia): %d \n", numero);
20     return 0;
21 }
22
23 // Passagem por valor
24 void soma(int a)
25 {
26     a = a + 10;
27     printf("Soma por valor: %d\n", a);
28 }
29
30 // Passagem por referencia
31 void somaref(int *a)
32 {
33     *a = *a + 10;
34     printf("Soma por referencia: %d\n", *a);
35 }
```

```
1  Digite um numero: 10
2  O numero digitado: 10
3  Soma por valor: 20
4  Agora o numero vale (por valor): 10
5  Soma por referencia: 20
6  Agora o numero vale (por referencia): 20
```

Note que: a função `soma` que recebe o parâmetro por valor realiza sua operação e ao final o valor do número original se mantém o mesmo. Em contraste, a função `somaref` recebe o argumento por referência e ao final sua operação altera o valor da variável original, que é perdido.

Providências para se passar o endereço de uma variável para uma função:

1. Na chamada da função deve-se usar o operador de endereço & antes do nome da variável;
2. No cabeçalho da função, declarar o parâmetro como um ponteiro;
3. Dentro da função, deve-se usar o operador de derreferência * para alterar o conteúdo da variável.

9.3 Retorno de múltiplos valores em uma função

A linguagem C permite o retorno direto de apenas **um único valor** para uma dada função. No entanto, há maneiras de se contornar essa limitação, por meio de métodos indiretos:

- Uso de ponteiros;
- Uso de estruturas;
- Uso de arrays.

Retorno de múltiplos valores usando ponteiros

É simples: basta passar o argumento para função *por referência* (endereço de memória) e fazer mudanças utilizando ponteiros (Seção 9.2). Dessa forma, os valores são alterados no argumento original, e são mantidas mesmo fora do ambiente da função.

10 Estruturas (Struct)

Até então, a linguagem de programação C já vem equipada com diversos tipos de dados: `char`, `int`, `float`, *etc.* Contudo, muitas vezes desejamos poder usar construtos que misturem tipos de dados distintos, e que agreguem informações sobre um mesmo ente. Nesses casos, é possível criar um tipo complexo de dado (dado derivado) ou uma (nova) estrutura de dados em C. Uma delas[†] são as **estruturas**.

As estruturas (ou registros) são coleções de dados, normalmente de tipos diferentes, que atuam como um todo, permitindo o armazenamento de uma só vez dos dados de uma mesma entidade. Podem conter tipos de dados simples (caractere, float, array e enumerado) ou compostos (estruturas, array ou uniões). As variáveis internas contidas no registro são denominadas *membros* da **struct**. Sintaxe geral:

```
1  struct NomeDaEstrutura {
2      tipoDeDado membro1;
3      tipoDeDado membro2;
4      // Pode ter mais membros...
5  };
```

[†]Outras estruturas de dados, além das estruturas, seriam as arrays, uniões, cadeias e campos de bits.

Um exemplo clássico:

```
1 struct Aluno {
2     char nome[50];
3     int idade;
4     float nota;
5 };
```

10.1 Criando structs: typedef

O comando `typedef` permite ao programador definir um novo nome para um determinado tipo. Sua forma geral é:

```
typedef antigo_nome novo_nome;
```

Como exemplo vamos dar o nome de inteiro para o tipo `int`:

```
typedef int inteiro;
```

Agora podemos declarar o tipo inteiro. O comando `typedef` também pode ser utilizado para dar nome a tipos complexos, como as estruturas. As estruturas criadas poderiam ser definidas como tipos através do comando `typedef`. O exemplo ficaria:

```
1  #include <stdio.h>
2
3  typedef struct tipo_endereco {
4      char rua [50];
5      int numero;
6      char bairro [20];
7      char cidade [30];
8      char sigla_estado [3];
9      long int CEP;
10     } TEndereco;
11
12     typedef struct ficha_pessoal {
13         char nome [50];
14         long int telefone;
15         TEndereco endereco;
16     } TFicha;
17
18     void main(void){
19         TFicha *ex;
20         //...
21     }
```

A sentença `typedef` é utilizada para atribuir um nome novo a um tipo de dado derivado ou básico. `typedef` não define um novo tipo, mas simplesmente um nome novo para um tipo existente:

```
1 typedef struct {
2     float x;
3     float y;
4 } PONTO;
```

A declaração de uma variável de tipo PUNTO: Ponto de origem = {0.0, 0.0}.

OBS: Os exemplos de `struct` lembram um pouco a funcionalidade dos **dicionários em Python** ... Na linguagem de programação C, não existe uma estrutura de dados nativa chamada “dicionário” como em Python. Em Python, um dicionário é uma coleção associativa que mapeia chaves para valores, permitindo que você armazene e recupere valores com base em suas chaves.

Em C, você não tem um tipo de dado nativo chamado “dicionário” com a mesma facilidade de uso e funcionalidade integrada. No entanto, você pode implementar um dicionário ou uma estrutura de mapeamento semelhante usando outras estruturas de dados disponíveis na linguagem, como *arrays*, *structs* e até mesmo ponteiros para funções. Isso requer mais código e trabalho manual em comparação com a simplicidade de um dicionário em Python.

10.2 Acessando e alterando campos de structs

Os campos (ou membros) de uma `struct` podem ser acessados e alterados por meio do uso do operador ponto (`.`), caso se trate de uma variável do tipo da `struct` em questão, ou operador seta (`->`), caso se trate de um ponteiro:

```
1  typedef struct ficha_pessoal
2      {
3      char nome [50];
4      long int telefone;
5      TEndereco endereco;
6      }TFicha;
7
8  TFicha ficha1; //Variavel do tipo TFicha
9  TFicha *ficha2; //Ponteiro do tipo TFicha
10
11  ficha1.telefone = 35353035; //Operador ponto
12  ficha2 -> telefone = 35353035; //Operador seta (ponteiro)
13  (*ficha2).telefone = 35353035; //Operador ponto + dereferencia
14                                     //equivalente ao comando de cima
```

Parte II

Estrutura de Dados

Programa

Funções Recursivas. Introdução a análise de algoritmos: notação assintótica, análise do pior caso, melhor caso. Tipos abstratos de dados. Listas lineares: sequenciais, simplesmente e duplamente encadeadas, estáticas e dinâmicas. Pilhas, filas, filas de prioridade. Aplicações: matrizes esparsas e grandes números. Listas generalizadas: polinômios. Listas não-lineares: árvores, árvores binárias, operações básicas sobre árvores. Árvores binárias de busca, árvores binárias de busca balanceadas (AVL).

11 Funções recursivas

11.1 Recursão

Recursividade é o ato de chamar a si mesmo; propriedade de função, programa ou afim que se pode invocar a si próprio; do latim *recursio*, -onis, corrida de volta, retorno revolução* (Seção 11.1).



Figura 3: Ilustração de recursividade.

*Dicionário Priberam da Língua Portuguesa

Na lógica matemática e em ciência da computação, uma *definição recursiva* é usada para definir um objeto em termos de si próprio. Em computação, recursão pode ser empregada como um método de resolução de problemas, que consiste em quebrá-lo em casos menores do mesmo problema, ou, em outras palavras, em subproblemas (instâncias) do mesmo tipo do problema original (Fig 4): *dividir para conquistar*. No contexto de funções recursivas e algoritmos, pode-se separar as recursões em duas partes igualmente importantes:

- **Casos base:** *inputs* para o qual a resposta da função é trivial;
- **Casos recursivos:** *inputs* para os quais a função chama a si própria.

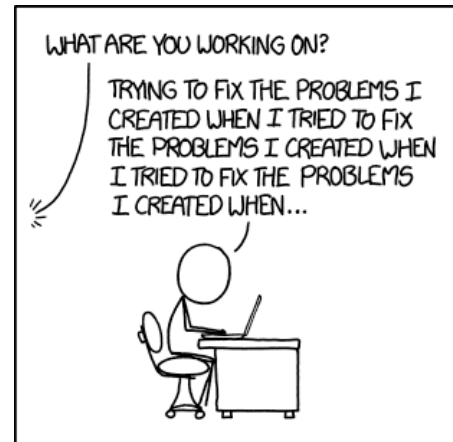


Figura 4: Resolvendo problemas por recursão.

Para garantir que não se caia em uma circularidade perigosa, desprovidos de esperança de sair de um *loop* eterno - que consumiria toda a memória disponível, é fundamental, além da lei da função de recursão, se definir um (ou mais) *casos base*. São eles os responsáveis por garantir que tenhamos uma solução para um problema pequeno o suficiente. Como os casos base quebram a cadeia de recursão, são também designados por “terminating case” (casos de encerramento).

Um dos exemplos matemáticos mais famosos e práticos, envolvem o fatorial de um número, a sequência de Fibonacci e a torre de Hanói.

11.2 O que acontece quando uma função é chamada e retornada?

Etapas do processo... por dentro do computador...

11.3 Iteração *vs.* Recursão

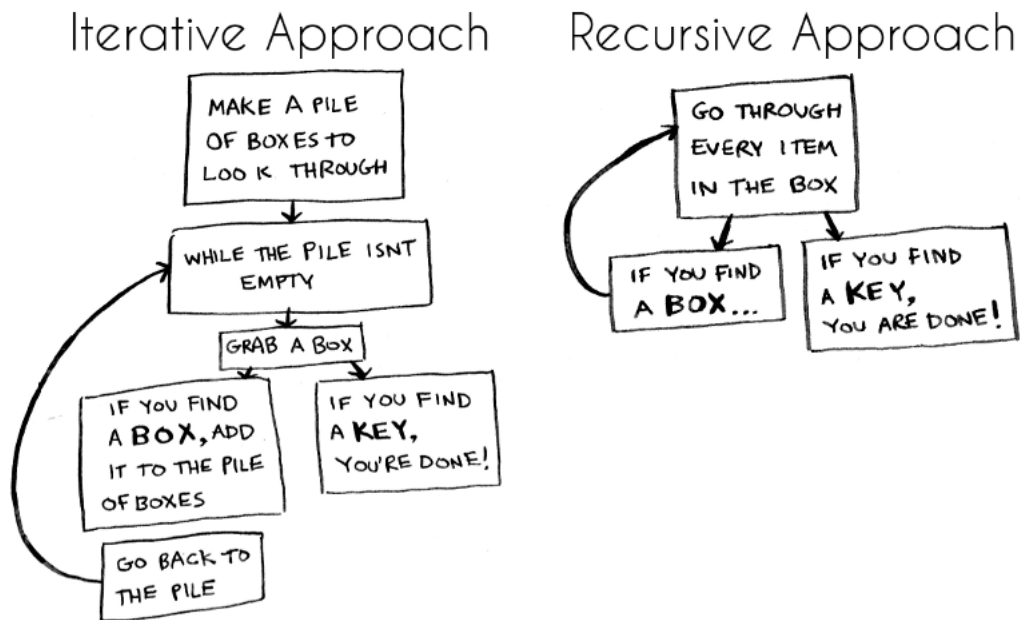


Figura 5: Programa iterativo *vs.* recursivo

12 Análise de algoritmos

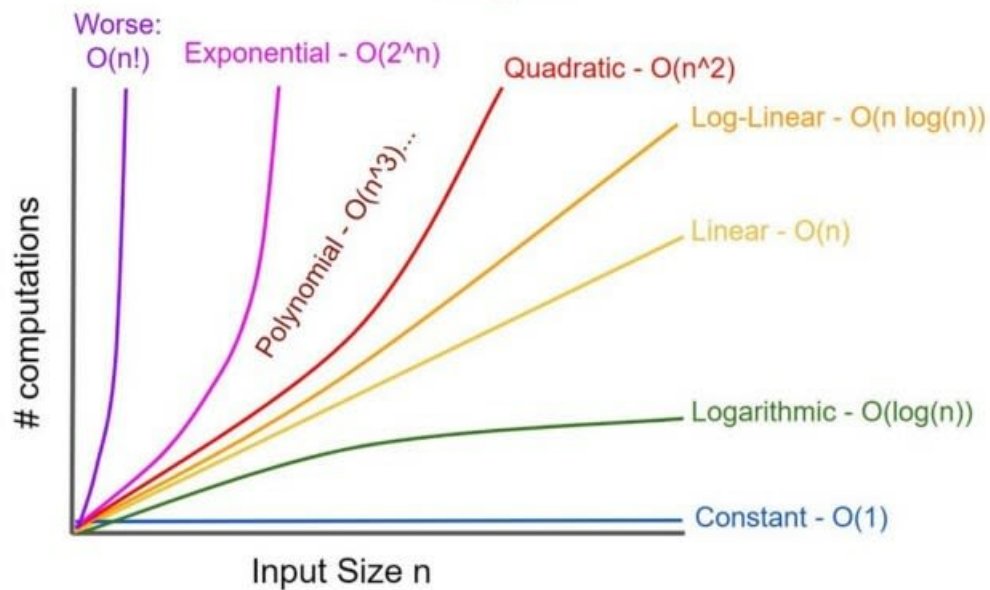


Figura 6: Complexidade de algoritmos - Big Oh

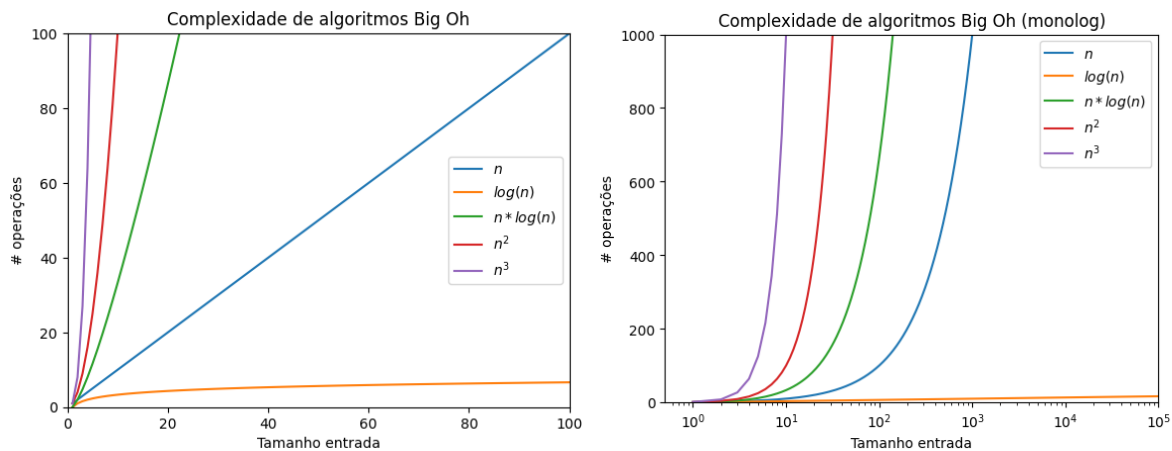


Figura 7: Complexidade de algoritmos - Big Oh, representação do comportamento das curvas em escala linear (à esquerda) e mono-log à direita.

13 Listas lineares

Lista linear é uma estrutura de dados na qual elementos de um mesmo tipo de dado estão organizados de maneira sequencial. Não necessariamente, estes elementos estão fisicamente em sequência, mas a ideia é que exista uma ordem lógica entre eles. Cada elemento, exceto o primeiro, possui um e apenas um antecessor, e cada elemento, exceto o último, possui um e apenas um sucessor.

Alocação de uma lista na memória

1. Sequencial (ou contígua): ordem lógica = ordem física
2. Encadeada: ordem lógica \neq ordem física

Operações com listas

- Criação de uma lista
- Remoção de uma lista
- Inserção de um elemento da lista
- Remoção de um elemento da lista
- Acesso de um elemento da lista
- Alteração de um elemento da lista
- Combinação de duas ou mais listas
- Classificação da lista
- Cópia da lista
- Localizar nodo[‡] através de info

As operações com a lista são implementadas separadamente, por meio de funções de gerenciamento (modularização do código).

[‡]Cada elemento da lista é chamado de nó, ou nodo.

Implementação de listas lineares

A implementação de listas lineares é feita por meio da criação e definição de estruturas: uso de `struct`'s. Existem diversos modelos de listas lineares, cada uma apresentando características próprias que devem ser respeitadas e incluídas ao se definir a estrutura em questão.

13.1 Listas lineares sequenciais

Lista cuja alocação na memória ocorre de maneira contígua, assim como a ordem lógica dos elementos. Ou seja, elementos adjacentes da lista ocupam posições vizinhas na memória física: ordem lógica = ordem física.

Vantagens

- ✓ Acesso direto pelo índice
- ✓ Tempo é constante

Desvantagens

- × Exige grandes movimentações para inclusão e exclusão de elementos

A forma mais comum de implementação: uso de um vetor de elementos estático do tipo `REGISTRO` de tamanho `MAX`:

```
1 // Definicao da estrutura: lista linear sequencial
2 #define MAX 50 // Tamanho vetor
3 #define false 0
4 #define true 1
5
6 typedef int bool;
7
8 typedef struct{
9     int chave;
10    // outros campos...
11 } REGISTRO;
12
13 typedef struct {
14     REGISTRO A[MAX]; // Vetor de elementos
15     int nElem;        // Numero de posicoes efetivamente ocupadas
16 } LISTA_LINEAR_SEQ;
```

- A estrutura `REGISTRO` é a que define e guarda informação de cada nó. Nesse caso, um nó contém apenas sua *chave*, que é o valor a ser armazenado;
- A lista sequencial é definida de fato em uma outra `struct`: `LISTA_LINEAR_SEQ`. Nessa estrutura, temos no caso duas informações:
 - Um vetor `A[MAX]` do tipo `REGISTRO`, que armazenará os múltiplos nós que comporão a lista;
 - Um inteiro `nElem` para indicar quantos nós a lista tem de fato, *i.e.* estão preenchidos.

13.2 Listas ligadas (ou encadeadas)

Lista linear na qual a ordem lógica dos elementos da lista (“nós”) não necessariamente coincide com sua posição física na memória: ordem lógica \neq ordem física.

O diferencial das listas ligadas (ou encadeadas) é que elas surgem para lidar melhor com a tarefa de inserção e exclusão de elementos em listas. Se em uma lista sequencial a adição ou retirada de um elemento em uma posição n implica no reposicionamento de todos os outros elementos seguintes para a casa vizinha, ação que exige muitas movimentações e operações, a lista encadeada contorna o problema com uma simples solução: permite que os dados ocupem qualquer posição em termos do endereço de memória, para tal criando um sistema de gerenciamento dos nós que guarda a informação da sua ordem e quais nós estão livres e ocupados.

Note que esse tipo de estrutura só é possível em linguagens que permitam a manipulação de memória, como C.

13.2.1 Implementação estática

A implementação estática de uma lista ligada (ou encadeada) se baseia no uso de vetores e inteiros. Consiste em uma **struct** para o nó e outra para lista em si. Elementos essenciais:

→ Lista ligada:

- Vetor de **REGISTRO**: guarda os nós presentes na lista;
- Campo **inicio**: indica o início da estrutura;
- Campo **dispo**: indica o início das posições livres da lista.

→ **REGISTRO**:

- Campo **chave** (e outros);
- Campo **prox**: indica o sucessor do elemento.

Na prática, **inicio** e **dispo** são as entradas de duas listas que compartilham o mesmo vetor, sendo uma para os elementos efetivos da lista, e a outra para armazenar as posições livres.

```
1 // Definicao da estrutura: lista ligada estatica
2 #define MAX 50
3
4 typedef struct{ // NO
5     int chave; // Valor do no
6     int prox; // Sucessor do no
7 } REGISTRO;
8
9 typedef struct{ // LISTA
10     REGISTRO A[MAX]; // Vetor de nos
11     int inicio; // Indica primeiro n0 efetivo (ocupado)
12     int dispo; // Inicio posicoes livres
13 } LISTA_LIGADA_ESTATICA;
```


Inicialização da lista

Função void: tarefa que exige a passagem da lista por referência, alteração da lista original. Para inicializar uma lista, seja ela qual for, é preciso, após declará-la, preencher todos os seus campos. No caso da lista ligada estática, isso implica em associar um valor para os campos `inicio`, `dispo`, e, como a lista será apenas criada (mas não preenchida), encadear os nós, preenchendo o campo `prox` do vetor de `REGISTRO`.

Campo	Valor inicial
<code>inicio</code>	-1
<code>dispo</code>	0
<code>A[i].prox</code>	$i + 1$
<code>A[MAX - 1].prox</code>	-1

Tabela 4: Valores de inicialização dos campos da lista ligada estática

Na criação da lista, não há nós preenchidos, portanto, `inicio = -1` (índice fora do vetor), indicando que a lista está vazia. Por outro lado, a lista estar vazia implica que todos os nós a partir do primeiro (índice 0) estão disponíveis, logo `dispo = 0`. É preciso encadear os nós, ligá-los entre si, sendo isso feito através da associação do campo `prox` de cada elemento (nó) do vetor de `REGISTRO` ao índice do elemento sucessor. Como estamos implementando com o uso de vetor, `A[i].prox` é simplesmente a posição do índice seguinte do elemento: $i + 1$, para todos, exceto para o último, que não apresenta sucessor: `A[MAX - 1].prox = -1` (fim da lista). E *voilà*: *fiat* lista ligada estática!

Exibição da lista

É importantíssimo ter em mente, que o importante é a todo custo manter a *ordem lógica* entre as informações e valores armazenados na lista. Ao se exibir a lista ligada, deseja-se os seus elementos em ordem lógica (não a ordem conforme foram adicionados à memória).

```
1 // * Exibicao da lista
2 void exibirListaLigEstat(LISTA_LIGADA_ESTATICA lista){
3     int i = lista.inicio;
4     while (i > -1){
5         printf("%d ", lista.A[i].chave);
6         i = lista.A[i].prox;
7     }
8     printf("\n");
9 }
```

Função void com passagem por valor (cópia): a lista é percorrida partindo do elemento na posição do vetor dada pelo campo `inicio` da lista, e seguindo o índice apontado pelo campo `prox` de cada nó, até que se chegue ao nó do último elemento preenchido da lista, que não apresenta sucessor, portanto `prox = -1`.

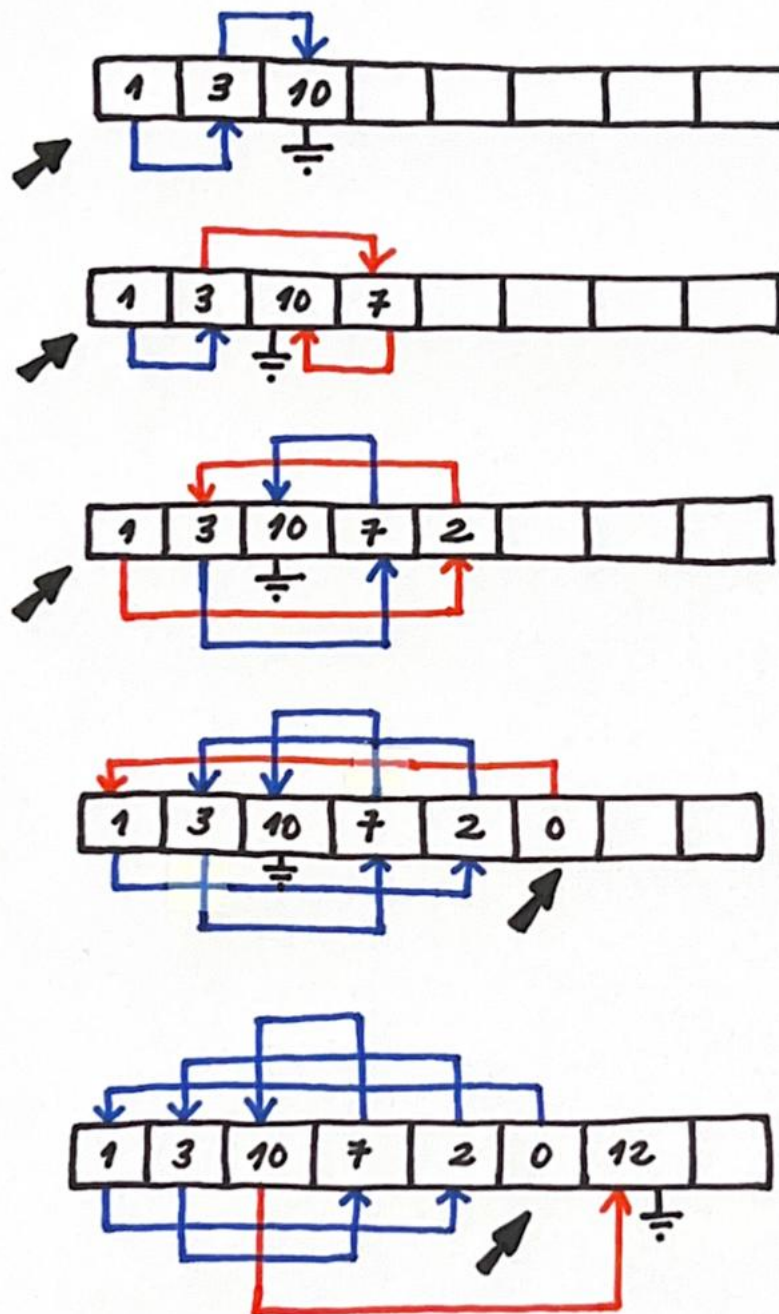


Figura 8: Esquema visual implementação lista ligada estática, múltiplas inserções de elementos. A seta (\rightarrow) indica o início da lista, enquanto o símbolo de aterramento indica o final dos elementos da lista. Ao se adicionar mais um elemento à lista, uma série de ações deve ser feita: obtenção de um nó livre para a nova chave, atribuição do campo ponteiro **prox** para o novo nó, além da reconexão do campo **prox** do nó antecessor, de modo a manter a ordem lógica da lista encadeada. Note que, ao se inserir um elemento na lista que ocupe a primeira posição na ordem lógica, o campo **inicio** da estrutura da lista deve ser alterado, de modo a indicar a posição do elemento inicial da lista (não necessariamente o índice 0 do vetor).

Inserção ordenada de elementos sem duplicações

Checkouts necessários (**return false**):

- ✓ Se há espaço na lista, *i.e.* **dispo** $\neq -1$;
- ✓ Busca para checar se o elemento já não está presente.

Inserção de elemento (**return true**):

- 1 - Associar um nó para o novo valor (pegar um nó na lista dos disponíveis);
- 2 - Realizar as devidas ligações dos ponteiros **prox** para garantir a ordem lógica.

Exclusão de elemento da lista

Checkout necessário (**return false**):

- ✓ Busca para checar se o elemento está presente.

Exclusão de elemento (**return true**):

- 1 - Devolver o nó para a lista dos nós disponíveis (alteração **dispo**);
- 2 - Realizar as devidas ligações dos ponteiros **prox** para garantir a ordem lógica.

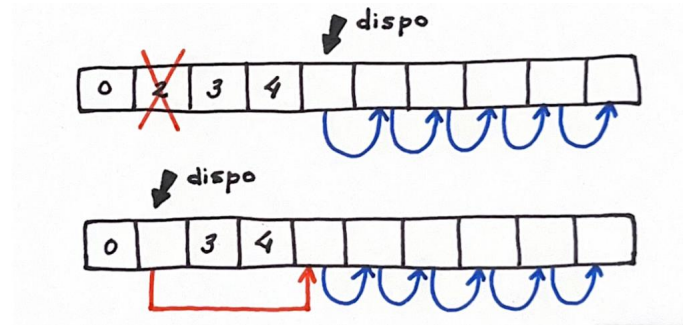


Figura 9: Esquema visual implementação lista ligada estática, exclusão de um elemento. A seta (\rightarrow) indica o início da lista de *nós disponíveis*: campo **dispo** da estrutura de lista ligada.

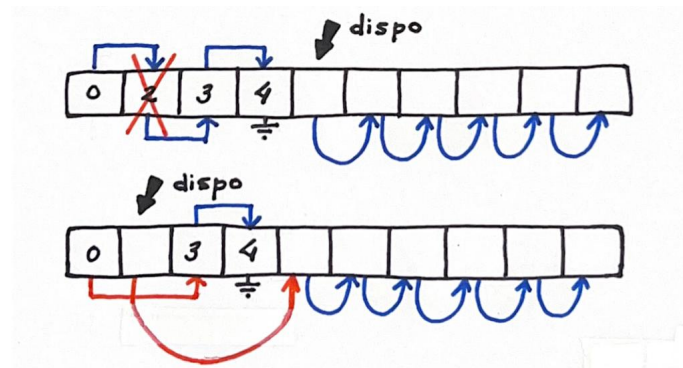


Figura 10: Esquema visual implementação lista ligada estática, exclusão de um elemento, mostrando conexões dos ponteiros **prox** tanto entre nós preenchidos da lista, quanto entre os nós disponíveis. A tarefa de exclusão de um elemento deixa claro como, de fato, há duas listas sendo mantidas em uma: nós ocupados (começo indicado por início) e nós disponíveis (indicada por **dispo**).

13.2.2 Implementação dinâmica

A implementação dinâmica de uma lista ligada (ou encadeada) se baseia no uso dos recursos de alocação dinâmica (<malloc.h>), *i.e.* ponteiros. Evita a necessidade da definição antecipada do tamanho máximo da lista, como acontece na implementação estática, sendo sua grande vantagem o efeito sanfona: usa tanta memória quanto necessário. No entanto, o controle do ponteiro é essencial; a perda do ponteiro implica na perda do acesso à lista.

Consiste em uma **struct** para os nós e outra para lista em si. Elementos essenciais:

→ NÓ:

- Campo **chave** (e outros);
- Campo ***prox**: indica o sucessor do elemento, note que temos um ponteiro de fato, autorreferente, pois aponta para outros nós.

→ Lista ligada dinâmica:

- Campo **inicio**: indica o início da estrutura.

Sintaxe básica:

```
typedef struct estrutura {
    TIPOCHAVE chave;
    int info;
    struct estrutura *prox;
} NO;
```

simplesmente um ponteiro **inicio** apontando para o primeiro nó da estrutura

“A única diferença significativa entre as implementações estática e dinâmica de listas ligadas está no fato de que a implementação estática “simula” uma lista ligada em vetor, e nos obriga a gerenciar as posições livres e ocupadas. Isso deixa de ser necessário no caso da alocação dinâmica “real”.”

- 13.3 Listas dinâmicas com nó cabeça e circularidade
- 13.4 Listas dinâmicas duplamente encadeadas com nó cabeça e circularidade
- 13.5 Filas
 - 13.5.1 Implementação estática
 - 13.5.2 Implementação dinâmica
 - 13.5.3 Filas com prioridade
- 13.6 Deques
- 13.7 Pilhas
 - 13.7.1 Implementação estática
 - 13.7.2 Implementação dinâmica
 - 13.7.3 Representação de duas pilhas em um único vetor
- 13.8 Matrizes esparsas: representação por listas cruzadas
- 13.9 Listas generalizadas

A Ambiente de trabalho: minha máquina

Na minha máquina...

- **Ambiente de trabalho:** Visual Studio Code (VS Code) - IDE;
- **Compilador:** gcc - instalado no terminal Ubuntu, mas com *set up* já pronto para rodar no terminal do próprio VS Code;
- **Depurador:** gdb.

Como programar: No VS Code, abrir pasta de interesse - salva na seção do Linux (Ubuntu) no meu computador, já conectada via WSL. Abrir ou criar arquivo com extensão *.c* para começar a programar em C.

B Comandos terminal Linux

B.1 Diretório de trabalho

Ao usar o terminal, é preciso ter em mente que você trabalha de dentro de uma pasta específica em seu computador. É sempre possível acessar itens de outras pastas em seu computador, mas o terminal controlará a pasta na qual você está atualmente. Isso é conhecido como seu **diretório de trabalho**.

Comando	Ação
<code>pwd</code>	Informa pasta do diretório de trabalho atual (<i>print working directory</i>)
<code>ls</code>	Lista todos os arquivos e pastas em seu diretório de trabalho (<i>list</i>)
<code>cd</code>	Altera diretório de trabalho, desce um nível (<i>change directory</i>)
<code>cd ..</code>	Altera diretório de trabalho, sobe para a pasta anterior

Tabela 5: Escrever os comandos no terminal e pressionar Enter

Para compilar e rodar um programa, é preciso estar no seu respectivo diretório de trabalho. Estando no lugar correto, em seguida é possível usar com sucesso as linhas de comando para compilar, rodar e debugar o arquivo de interesse.

```
● amanda_araujo@cubo-redondo:~$ pwd
/home/amanda_araujo
● amanda_araujo@cubo-redondo:~$ ls
estrutura-dados-I lab-icc
● amanda_araujo@cubo-redondo:~$ cd estrutura-dados-I
● amanda_araujo@cubo-redondo:~/estrutura-dados-I$ ls
bla bla.c funcoes.c lista1 lista1.c
● amanda_araujo@cubo-redondo:~/estrutura-dados-I$
```

Figura 11: Acessando diretório de trabalho de interesse, onde os códigos em C estão salvos.

Dica: Seta para cima no teclado (↑) permite navegar pelos comandos anteriormente escritos no terminal. Evita o trabalho de reescrever o mesmo comando novamente.

B.2 Compilar, rodar e debugar

Linha de comando (terminal dentro do VS Code)

Compilar		gcc nomedoarquivo.extensão -o nomedoarquivo
Rodar		./nomedoarquivo
Debug		gdb ./nomedoarquivo

Atalho de teclado (shortcut)

Compilar + rodar		<i>Ctrl + Shift + B</i>
Rodar + Debug		<i>F5</i>
Comentário linha (//)		<i>Ctrl + ;</i>
Comentário bloco (/**/)		<i>Shift + Alt + A</i>

Referências

1. “Pointers in Python: What’s the Point?”, *Real Python*, <https://realpython.com/pointers-in-python/>
2. “Ponteiros em C”, <https://linguagemc.com.br/ponteiros-em-c/>
3. “Manual de Sintaxe da Linguagem C”, <https://www.feg.unesp.br/>
4. “Estrutura de Dados e Algoritmos - COS-121”, Ricardo Farias, https://www.cos.ufrj.br/~rfarias/cos121/aula_10.html
5. “ACH2023 - Algoritmos e Estruturas de Dados I”, Willian Yukio Honda & Ivandré Paraboni, EACH-USP, <http://each.uspnet.usp.br/digiampietri/ACH2023/ACH2023.pdf>
6. “Como navegar em arquivos e pastas em um terminal”, *Terminal Cheat Sheet*, <https://terminalcheatsheet.com/pt-BR/guides>
7. “Recursion (Computer Science)”, *Wikipedia*, [https://en.wikipedia.org/wiki/Recursion_\(computer_science\)](https://en.wikipedia.org/wiki/Recursion_(computer_science))