Professor: Fabrício Varajão – varajao@gmail.com – www.varajao.com.br

Apostila

Revisado e atualizado em Ago/2017

ÍNDICE

1. Fundamentos em C++

Vamos relembrar rapidamente alguns fatores importantes para depois nos dedicar ao conteúdo específico para a disciplina de Pesquisa e Ordenação.

1.1. Princípios básicos

Toda aplicação em C++ possui uma estrutura elementar. Seguindo a tradição, mostrarei abaixo o programa "Alo, mundo!" que trata-se de um programa que simplesmente exibe esta mensagem na tela.

```
// Um programa elementar.
#include <stdio.h>
int main() {
      printf("Olá, Mundo!");
      return 0;
}
// Fim de main()
```

Neste momento alguns pontos importantes serão revistos de estudos anteriores e são eles: bibliotecas, variáveis, seus tipos, declaração. Também veremos como funcionam os operadores e as expressões.

1.1.1. Comentários

Comentários são linhas não compiladas, ou seja, não são comandos que devem ser executados pelo programa.

Então, se não são linhas executadas, qual a finalidade de adicionar um comentário?

A função dos comentários é deixar o código mais legível e para fazer anotações importantes dentro do próprio código, como por exemplo, anotar porque você usou uma lógica ao invés de outra, porque você nomeou aquela variável de tal forma, etc.

```
//Comentario de uma linha
/*
  Comentarios de mais de uma linha
  ou em bloco, devem ser feitos assim
*/
```

Os comentários de uma linha foram introduzidos em C++ e são definidos pelas barras barras duplas (//). Tudo que estiver nessa linha será ignorado pelo compilador.

Já os comentários de várias linhas têm em ambos, tanto C como C++. Você procede abrindo o comentário com /* e demonstrando onde termina o comentário com */. Tudo de estiver depois de /* e antes de */ será ignorado pelo compilador. Por isso nunca se esqueça de que quando abrir (/*), obrigatóriamente, você terá que fechar (*/). Isso é um erro muito comum de se cometer, principalmente para quem está começando.

1.1.2. Pré-processamento (Include e as Bibliotecas)

Uma característica marcante de C é o pré-processamento.

A linguagem C tem a capacidade de importar bibliotecas de funções que são, leigamente, comandos já prontos para usar.

Cada um desses comandos, que chamamos de FUNÇÕES, estão organizados em bibliotecas diferentes. Bastando a nós determinar em qual biblioteca o compilador irá encontrar tal função.

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

O pré-processamento é determinado pelo caracter #.

Para importar uma biblioteca, basta usar o comando include.

Quanto a inclusão de biblioteca, há diferenças entre C e C++ tanto na forma de inclusão quanto nas bibliotecas, que não são as mesmas.

Uma vez que a função já está pronta dentro da biblioteca, basta importar tal biblioteca e utilizar a função que queremos.

Por exemplo, se você quiser mostrar uma mensagem na tela, você não tem que produzir uma função inteira ou criar um comando novo, basta importar uma biblioteca de I/O (entrada e saída) e utilizar uma função dela. Quando for compilar, o compilador vai buscar nas bibliotecas tais funções para saber como utilizá-las.

O papel do pré-processamento é indicar, antes mesmo de compilar, os parâmetros necessários para ser criado o arquivo executável.

O pré-processamento é indicado pelo caracter sharp (#) no inicio da linha e deve ser usado no início da programação.

Importando uma biblioteca

A importação de uma biblioteca é dada pelo comando INCLUDE (incluir) seguido da biblioteca entre os sinais de menor (<) e maior (>).

Porém, devemos notar que existem diferenças entre a importação de bibliotecas de C e de C++.

Importando uma biblioteca em C

Em C, a importação de bibliotecas são mais simples, bastando acrescentar para cada biblioteca um include em uma linha diferente e o nome da biblioteca seguido de ponto H (.h) - .h é a extensão do arquivo da biblioteca que vem da palavra inglesa HEADER (cabeçalho) - se você esquecer de colocá-lo o programa não será compilado.

No exemplo abaixo, vamos incluir duas bibliotecas padrões de C.

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

Como podem ver, cada biblioteca foi incluída em uma linha.

Importando uma biblioteca em C++

Em C++, a importação de bibliotecas é um pouco diferente. O comando de importação continua sendo o include, mas nesse caso, não usaremos o ponto H (.h) no final das bibliotecas legitimamente* de C++.

**Muitos compiladores aceitam formas híbridas de inclusão de bibliotecas, podendo misturar bibliotecas de C e de C++. Veremos um exemplo mais a frente.*

Após importarmos uma biblioteca por linha, no final dessa lista devemos demonstrar que tipo de funções usaremos. Na grande maioria dos casos usaremos as funções padrões de cada bibloteca. Podemos fazer isso facilmente digitando a palavra reservada USING* indicando o espaço de nome (namespace) standard (std), que quer dizer padrão em inglês.

Não se esqueça que a palavra reservada USING necessita terminar a linha com ponto e vírgula (;).

Abaixo importaremos uma biblioteca padrão de entrada e saída de C++.

```
#include <iostream>
using namespace std;
```

No exemplo abaixo uma importação híbrida de bibliotecas de C e C++.

```
#include <stdlib.h>
//biblioteca de C

#include <iostream>
//biblioteca de C++
using namespace std;
```

As bibliotecas de C são diferentes das bibliotecas de C++. Apesar de muitos compiladores de C++ suportarem as bibliotecas de C, nenhum compilador exclusivamente de C suporta bibliotecas de C++.

1.1.3. Função Principal

Todo o programa em C está estrito dentro da função MAIN.

MAIN é a primeira função a ser executada por qualquer programa em C, mesmo que tenha outras funções escritas antes dela.

```
int main (void)
```

Há compiladores que interpretam esta função mesmo incompleta, ou seja, você também tem como começar esta função escrevendo:

```
main ()
int main ()
main (void)
```

Mas para manter a portabilidade e evitar erros desnecessários de compilação, o melhor a fazer é declarar ela inteira -> int main (void).

Devemos salientar também que deve ser tudo em letras minúsculas, pois a linguagem C é case sensite, trocando por miúdos, ele distingüi entre letras maiúsculas e minúsculas. Por exemplo:

MAIN não é a mesma coisa que Main, que não é a mesma coisa que MaiN, que por sua vez, não é a mesma coisa que MaIn, que também é diferente de main.

Então, de preferência, sempre use letras minúsculas.

1.1.4. Bloco de dados

Os blocos de dados são utilizados para determinar todas as linhas de código que comandos ou funções devem executar de uma vez.

```
{ }
```

O bloco de dados é determinado pelo abrir e fechar de chaves ({}).

Você abre chaves onde se deve começar a executar e fecha onde deve terminar a execução. Em um programa pode haver mais de uma chave e o mesmo número de chaves aberta deve ser o de chaves fechadas.

No caso do primeiro programa, apenas abre a execução da função principal (main) e fecha-se no final de sua execução.

1.1.5. Funções e Processos

Qualquer programa é determinado pelo que está dentro das chaves.

O que está dentro das chaves pode ser uma função, uma palavra reservada (como IF, ELSE, FOR, ...) ou processo (operação matemática).

```
printf ("Olá! Mundo!");
return 0;
```

Em C, toda chamada de função, processo ou algumas palavras reservadas devem terminar a linha com um ponto e vírgula (;).

Como você pode ver na chamada da função printf e da palavra reservada return.

1.1.6. Variáveis

Variáveis são espaços de memória para armazenar um dado, devem possuir um tipo e um nome, onde o nome identifica o espaço de memória e o tipo determina a natureza do dado.

Tabela de especificações gerais dos tipos de variáveis

Tipo	Tamanho	Menor valor	Maior valor
char	1 byte	-128	+127
unsigned char	1 byte	0	+255
short int (short)	2 bytes	-32.768	+32.767
unsigned short int	2 bytes	0	+65.535
int (*)	4 bytes	-2.147.483.648	+2.147.483.647
long int (long)	4 bytes	-2.147.483.648	+2.147.483.647
unsigned long int	4 bytes	0	+4.294.967.295
float	4 bytes	-10^{38}	$+10^{38}$
double	8 bytes	-10^{308}	$+10^{308}$

(*) depende da máquina, sendo 4 bytes para arquiteturas de 32 bits

Variáveis devem ser explicitamente declaradas e podem ser declaradas em conjunto:

```
int a; /* declara uma variável do tipo int */
int b; /* declara uma variável do tipo int */
float c; /* declara uma variável do tipo float */
int d, e; /* declara duas variáveis do tipo int */
```

Variáveis só armazenam valores do mesmo tipo com que foram declaradas:

```
int a; /* declara uma variável do tipo int */
a = 4.3; /* a armazenará o valor 4 */
```

Uma variável pode receber um valor quando é definida (inicializada), ou através de um operador de atribuição:

```
int a = 5, b = 10; /*declara e inicializa duas variáveis do tipo int*/
float c = 5.3; /* declara e inicializa uma variável do tipo float */
```

Uma variável deve ter um valor definido quando é utilizada:

```
int a, b, c; /* declara e inicializa duas variáveis do tipo int */
a = 2;  /* atribui o valor 2 na variável a */
c = a + b; /* ERRO: b contém "lixo" */
```

Variável global

Declarada fora do corpo das funções e é visível por todas as funções subsequentes. Não é armazenada na pilha de execução, não deixa de existir quando a execução de uma função termina e existe enquanto o programa estiver sendo executado.

Devem ser utilizadas com critério pois pode-se criar um alto grau de interdependência entre as funções e dificulta o entendimento e o reuso do código.

```
#include <stdio.h>
int s, p; /* variáveis globais */
void somaprod (int a, int b)
{
   s = a + b;
   p = a * b;
}
int main (void)
```

```
{
   int x, y;
   scanf("%d %d", &x, &y);
   somaprod(x,y);
   printf("Soma = %d produto = %d\n", s, p);
   return 0;
}
```

Variável estática

Declarada no corpo de uma função e é visível apenas dentro da função em que foi declarada. Não é armazenada na pilha de execução, armazenada em uma área de memória estática e continua existindo antes ou depois de a função ser executada.

Devem ser utilizadas quando for necessário recuperar o valor de uma variável atribuída na última vez que a função foi executada.

```
void imprime ( float a )
{
   static int n = 1;
   printf(" %f ", a);
   if ((n % 5) == 0) printf(" \n ");
   n++;
}
```

1.1.7. Constantes

Além das variáveis podemos utilizar outros espaços de memória para armazenar dados, porém nestes os dados são fixados.

```
#define PI 3.14159F /* declara uma constante do tipo float denominada
                       PI e atribui o valor 3.14159 */
```

1.1.8. Operadores e Expressões

Os operadores são classificados em aritméticos, atribuição, incremento e decremento, relacionais e lógicos.

Operadores aritméticos (+ , - , * , / , %):

Operações são feitas na precisão dos operandos onde o operando com tipo de menor expressividade é convertido para o tipo do operando com tipo de maior expressividade. Divisão entre inteiros trunca a parte fracionária:

```
int a
double b, c;
a = 3.5; /* a recebe o valor 3 */
b = a / 2.0; /* b recebe o valor 1.5 */
c = 1/3 + b; /* 1/3 retorna 0 pois a operação será sobre inteiros */
             /* c recebe o valor de b */
```

O operador módulo, "%", aplica-se a inteiros:

```
x % 2 /* o resultado será 0, se x for par; caso contrário, será 1 */
```

Precedência dos operadores: * , / , - , +

```
a + b * c / d é equivalente a (a + ((b * c) / d))
```

*Operadores de atribuição (= , += , -= , *= , /= , %=):*

Uma atribuição é como uma expressão, onde a ordem é da direita para a esquerda e oferece uma notação compacta para atribuições em que a mesma variável aparece dos dois lados, isto é:

var op = expr é equivalente a var = var op (expr)

```
i += 2; /* é equivalente a i = i + 2; */
x *= y + 1; /* é equivalente a x = x * (y +1); */
```

Operadores de incremento e decremento (++ , --):

Incrementa ou decrementa de uma unidade o valor de uma variável. Estes operadores não se aplicam a expressões e o incremento pode ser antes ou depois da variável ser utilizada:

n++ incrementa n de uma unidade, depois de ser usado

++n incrementa n de uma unidade, antes de ser usado

```
n = 5;
x = n++; /* x recebe 5; n é incrementada para 6 */
x = ++n; /* n é incrementada para 6; x recebe 6 */
a = 3;
b = a++ * 2; / b termina com o valor 6 e a com o valor 4 */
```

Operadores relacionais (< , <= , == , >= , > , !=):

O resultado será 0 ou 1 (não há valores booleanos em C):

```
int a, b;
int c = 23;
int d = c + 4;
c < 20 /* retorna 0 */
d > c /* retorna 1 */
```

Operadores lógicos (&& , ||)

A avaliação é da esquerda para a direita e para quando o resultado pode ser conhecido:

```
int a, b;
int c = 23;
int d = c + 4;
a = (c < 20) || (d > c); /* <ou> retorna 1 - as 2 sub-expressões são avaliadas */
b = (c < 20) && (d > c); /* <e> retorna 0 - apenas a 1ª sub-expressão é avaliada */
```

Expressões para conversão de tipo de variável

A conversão de tipo é automática na avaliação de uma expressão, mas pode ser requisita explicitamente:

```
float f;     /* valor 3 é convertido automaticamente para "float" */
float f = 3; /* passa a valer 3.0F, antes de ser atribuído a f */
```

```
int g, h;          /* 3.5 é convertido (e arredondado) para "int" */
g = (int) 3.5;     /* antes de ser atribuído à variável g */
h = (int) 3.5 % 2 /* e antes de aplicar o operador módulo "%" */
```

1.2. Entrada e Saída

Existem algumas funções muito utilizadas para coletar e apresentar dados, iremos relembrar as mais importantes.

1.2.1. printf

Possibilita a saída de valores segundo um determinado formato e pode ser utilizado para gerar saídas de texto:

Sintaxe: printf (formato, lista de constantes/variáveis/expressões...);

```
printf ("%d %g", 33, 5.3); /* tem como resultado a linha: */

33 5.3
```

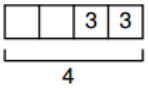
```
printf ("Inteiro = %d Real = %g", 33, 5.3); /* com saída: */

Inteiro = 33 Real = 5.3
```

```
printf ("Curso:\t BSI");   /* com saída: */

Curso:      BSI
```

Tabela de especificações de formatos

Formato	Descrição
%c	Especifica um char
%d	Especifica um int
%ld	Especifica um inteiro longo
%u	Especifica um unsigned int
%f	Especifica um double (ou float)
%e	Especifica um double (ou float) no formato científico
%s	Especifica uma cadeia de caracteres
%x	Especifica um número inteiro em formato hexadecimal

Pode-se especificar o tamanho do campo combinando a quantidade de dígitos no formato:

Formato	Resultado
%4d	

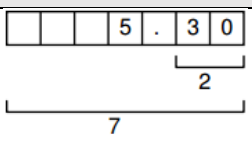
Formato	Resultado
%7.2f	

Tabela de especificações de caracteres de “escape”

Formato	Descrição
\n	Caractere de nova linha
\t	Caractere de tabulação
\r	Caractere de retrocesso
\"	Caractere “
\\	Caractere \

1.2.2. scanf

Captura valores fornecidos via teclado:

Sintaxe: scanf (formato, lista de endereços das variáveis...);

```
int n;
scanf ("%d", &n); /* valor inteiro digitado pelo usuário é armazenado
                     na variável n */
```

Tabela de especificações de formatos

Formato	Descrição
%c	Especifica um char
%d	Especifica um int
%u	Especifica um unsigned int
%f, %e, %g	Especificam um float
%lf, %le, %lg	Especificam um double
%s	Especifica uma cadeia de caracteres

Caracteres diferentes dos especificadores no formato servem para cercar a entrada. Espaço em branco dentro do formato faz com que sejam “pulados” eventuais brancos da entrada.

%d, %f, %e e %g automaticamente pulam os brancos que precederem os valores numéricos a serem capturados:

```
scanf ("%d:%d", &h, &m); /* valores (inteiros) fornecidos devem ser
                            separados pelo caractere dois pontos (:) */
```

1.2.3. Stream de saída - Objeto cout

O objeto cout representa o stream de saída. Este stream é uma espécie de sequência (fluxo) de dados a serem impressos na tela. Para realizar a impressão, usa-se o "operador de inserção" que "insere" dados dentro do stream.

<< Operador de Inserção

O operador << sobrecarregado executa a saída (imprime na tela) com streams. O objeto cout é usado em conjunto com ele para a impressão de dados.

```
#include <iostream>
```

```
using namespace std;
int main()
{
  cout << "Imprimindo o famoso HELLO WORLD!!!\n";

  // imprimindo uma linha usando múltiplos comandos
  cout << "Teste com ";
  cout << "dois couts\n";

  // usando o manipulador endl
  // endl gera um caractere nova linha, e também descarrega o buffer de saída
  cout << "Escrevendo uma linha..." << endl;

  cout << "Mais uma vez...\n";
  cout << flush; // agora apenas esvaziando o buffer de saída sem gerar nova linha
  return 0;
}
```

<< sempre retorna uma referência ao objeto cout. Desta forma, é possível encadear vários cout, permitindo uma combinação de constantes e variáveis, como mostra o exemplo abaixo:

```
int a = 10;
int b = 12;
// imprime uma string e o resultado da soma entre as variáveis a e b
cout << "Encadeando saídas: Somandos dois números " << a + b << endl;
```

A saída de caracteres também pode ser feita sem o uso do operador de inserção <<. Para isto, usa-se a função membro put, que retorna uma referência para o objeto cout. Exemplos:

```
cout.put('caractere');              // fazendo a saída de um único caractere
cout.put('caractere').put('\n');    // fazendo a saída encadeada
```

1.2.4. Stream de entrada - Objeto cin

O objeto cin representa o stream de entrada. Ele realiza a leitura de um seqüência de dados, sem espaços e sem tabulações, vindas do teclado. Para coletar estes dados armazenados, usa-se o "operador de extração" que "extrai" dados do stream.

>> Operador de Extração

O operador >> sobrecarregado executa a entrada com streams em C++, usando o comando cin para aquisição de dados. Variáveis podem ser usadas para o armazenamento das informações.

```
#include <iostream>
using namespace std;
int main()
{
  int Num1;
  int Num2;
  cout << "Lendo o primeiro número...: ";
  cin >> Num1;
  cout << endl;
  cout << "Lendo o segundo número....: ";
  cin >> Num2;
  cout << endl;
  return 0;
}
```

Da mesma forma que o operador de inserção, >> sempre retorna uma referência ao objeto cin, permitindo o encadeamento de vários cin. Exemplo:

```
float c;
float d;
cin >> c >> d;    // entra com um float, pressiona <ENTER>,
                  // entra com outro float, pressiona <ENTER>
```

Outras peculiaridades usando o operador de extração:

```
while(cin >> dado)    // repete enquanto o usuário não digitar fim de arquivo
                      // (Ctrl+Z ou Ctrl+D)

cin.eof();            // devolve 1 (verdadeiro) se chegou ao fim do arquivo

char c;
c = cin.get();
cin.get(vet, n, '\n');
char nome[100];
cin >> nome;            // lê até o 1° espaço em branco apenas
cin.get(nome, 100);     // lê até o 99° caractere ou <ENTER>
cin.get(nome, 100, 'x');

/* get sempre insere um NULL no final da seqüência lida.
   O delimitador não é retirado da entrada. Isto faz com que a próxima
   leitura devova este caractere (no caso, uma linha será devolvida. */

cin.getline(nome, TAM);    // remove o delimitador do buffer automaticamente
cin.putback();        // recoloca o último caractere lido de volta no buffer
```

Entrada e Saída segura quanto ao tipo

Quando dados inesperados são processados pelos operadores << e >>, vários indicadores de erros podem ser ativados para indicar se houve sucesso ou falha em uma operação de E/S. O exemplo a seguir apresenta alguns deles:

```
int dado;
cin.eof();         // retorna verdadeio ser o fim do arquivo foi encontrado
cin >> dado;       // lê um inteiro
if(cin.bad()) {    // houve uma falha na leitura
  // ...
}
if(cin.fail()) {   // dado digitado incompatível com o dado esperado
  // ...
}
if(cin.fail()) {   // estado da stream está OK
  // ...
}
```

ou usando

```

ios::eofbit
ios::badbit
ios::failbit
ios::goodbit
cin.clear();      // reseta o estado de stream, ativando "goodbit"
```

1.2.5. Manipuladores de streams

Executam tarefas de formatação, por meio de diferentes recursos. Alguns deles estão abaixo destacados:

Base do stream de inteiros

```
#include <iomanip>
#include <iostream>
using namespace std;
// ...
cin >> n;
cout << hex << n << endl            // apresenta n em hexadecimal
     << oct << n << endl            // apresenta n em octadecimal
     << setbase(10) << n << endl;   // apresenta n na base 10
// ...
```

Precisão em ponto flutuante

```
double PI = 3.14159265;

cout.precision(5);    // define a precisão
cout << setiosflags (ios::fixed) << f << endl;
cout << PI;           // imprime PI com 5 casas decimais
ou
cout << setiosflags (ios::fixed) << setprecision(5) << PI;    // define e
imprime PI com 5 casas decimais
```

Para retornar ao formato de impressão padrão

```
  cout << setiosflags (ios::scientific);
  cout << f << endl;
```

Largura de campo

Para definir a largura da impressão de uma variável usa-se o setw(n), onde n é o número de caracteres desejado para a impressão desta variável. Note que o setw só afeta a próxima variáveldo cout.

```
int caixas = 45;
cout << "Número de caixas: " << setw(10) << caixas << endl;
```

Alinhamento da impressão

É possível alinhar os dados impressos tanto à esquerda quanto à direita, com os manipuladores left e right.

```
cout  << right <<  10;
cout  << left <<  10;
```

Entrada de Strings

Em determinadas ocasiões, deseja-se coletar dados que contenham strings com tabulações, espaços em branco e/ou novas linhas. Frases são exemplos onde este tipo de situação ocorre. No entanto, o operador >> sobrecarregado ignora tais caracteres.

Para englobar essas situações, C++ oferece o uso da função-membro getline. Esta função remove o delimitador do stream (isto é, lê o caractere e o descarta) e armazena o mesmo em uma variável definida pelo usuário. A seguir, um exemplo de sua utilização.

```
#include <iostream>
using namespace std;
int main()
{
  const TAM = 80;
  char guarda[TAM];
  cout << "Digite uma frase com espaços: " << endl;
  cin.getline(guarda, TAM);
  cout << "A frase digitada é: \n" << guarda << endl;
  return 0;
}
```

1.3. Controles de fluxo

A seguir um breve resumo dos comandos para controle de fluxo, dentre eles os de tomadas de decisão, construtores de laços e seleção.

1.3.1. if

Comando básico para codificar tomada de decisão. Se expr for verdadeira (≠ 0), executa o bloco de comandos 1, se expr for falsa (= 0), executa o bloco de comandos 2.

Sintaxe: if (expr) { bloco de comandos 1 }

else { bloco de comandos 2 }

ou

if (expr) { bloco de comandos }

```
if ( a > b )
  maximo = a;
else maximo = b;
```

Existe um operador condicional (?:) que em alguns casos pode substituir o if:

Sintaxe: condição ? expressão1 : expressão2;

```
maximo = a > b ? a : b ; /* equivale ao if anterior */
```

1.3.2. while

Enquanto expr for verdadeira, o bloco de comandos é executado, quando expr for falsa, o comando termina:

Sintaxe: while (expr)

{ bloco de comandos }

```
k = 1;
while (k <= n)
{
      f = f * k; /* a expressão "f = f * k" é equivalente a "f *= k" */
      k = k + 1; /* a expressão "k = k + 1" é equivalente a "k++" */
}
```

1.3.3. for

É uma forma compacta para exprimir laços:

Sintaxe: for (expressão_inicial; expressão_booleana; expressão_de_incremento)

{ bloco de comandos }

```
for (k = 1; k <= n; k=k+1)  /* a expressão "k = k + 1" é equivalente
{                            a "i++" */
      f = f * k; /* a expressão "f = f * k" é equivalente a "f *= k" */
}
```

1.3.4. do – while

O teste de encerramento é avaliado no final:

Sintaxe: do

{ bloco de comandos }

while (expr);

```
do
{  printf("Digite um valor inteiro nao negativo:");
   scanf ("%d", &n);
} while (n<0);
```

1.3.5. break

Termina a execução do comando de laço:

```
for (i = 0; i < 10; i++) {
    if (i == 5)
       break;
    printf("%d ", i);
} /* A saída deste programa será: 0 1 2 3 4 */
```

1.3.6. continue

Termina a iteração corrente e passa para a próxima:

```
for (i = 0; i < 10; i++ ) {
    if (i == 5)
       continue;
    printf("%d ", i);
}  /* gera a saída: 0 1 2 3 4 6 7 8 9  -- não listando o 5*/
```

1.3.7. switch

Seleciona uma entre vários casos. Que só pode ser operado com inteiro ou caracter.

Sintaxe: switch (expr)

{
case op1: bloco de comandos 1; break;
case op2: bloco de comandos 2; break;
default: bloco de comandos default; break;
}

```
switch (op)
{
   case '+': printf(" = %f\n", num1+num2); break;
   case '-': printf(" = %f\n", num1-num2); break;
   default: printf("Operador invalido!\n"); break;
}
```

Tabela resumo das sintaxes dos controles de fluxo

`if ( expr ) { bloco de comandos } else { bloco de comandos }`
`condição ? expressão1 : expressão2;`
`while (expr) { bloco de comandos }`

`for ( expr_inicial; expr_booleana; expr_de_incremento) {`
`  bloco de comandos`
`}`

`do { bloco de comandos } while ( expr );`

`switch ( expr ) {`
`  case op1: bloco de comandos 1; break;`
`  case op2: bloco de comandos 2; break;`
`  ...`
`  default: bloco de comandos default; break;`
`}`

1.4. Funções

Além das funções pré-existentes na linguagem podemos criar nossas próprias funções. Para isso devemos seguir a seguinte sintaxe:

tipo_retornado nome_da_função (lista de parâmetros...)

{ corpo da função }

```
#include <stdio.h> /* programa que lê um número e imprime seu fatorial */
int fat (int n);
int main (void)
{ int n, r;
  printf("Digite um número nao negativo:");
  scanf("%d", &n);
  r = fat(n);
  printf("Fatorial = %d\n", r);
  return 0;
}
```

"Protótipo" da função: deve ser incluído antes da função ser chamada

Chamada da função

"main" retorna um inteiro:
0 : execução OK
≠ 0 : execução não OK

```
/* função para calcular o valor do fatorial */
int fat (int n)
{ int i;
  int f = 1;
  for (i = 1; i <= n; i++)
    f *= i;
  return f;
}
```

Declaração da função: indica o **tipo da saída** e o **tipo e nome das entradas**

Retorna o valor da função

1.4.1. Funções recursivas

Recursividade é a chamada de uma função a partir de outra. A recursão pode se dar de duas formas:

- Direta: Quando uma função A chama a ela própria
- Indireta: Quando uma função A chama uma função B que, por sua vez, chama A

Quando uma função é chamada recursivamente, cria-se um ambiente local para cada chamada. As variáveis locais de chamadas recursivas são independentes entre si, como se estivéssemos chamando funções diferentes.

```
int fat (int n) /* Função recursiva para cálculo do fatorial */
{
  if (n==0) return 1;
  else return n*fat(n-1);
}
```

Nota: *Mais à frente falaremos novamente de recursividade.*

1.4.2. Funções macros

É como uma fusão entre uma constante e uma função. Sua sintaxe segue como uma definição de constante:

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
v = 4.5;
c = MAX(v, 3.0);
```

O pré-processador substituirá o trecho de código por:

```
v = 4.5;
c = ((v) > (3.0) ? (v) : (3.0));
```

Tome cuidado na definição de macros, pois erro de sintaxe pode ser difícil de ser detectado e o compilador indicará erro na linha em que se utiliza a macro e não na linha de definição da macro (onde efetivamente encontra-se o erro).

Envolva cada parâmetro, além da macro como um todo, entre parênteses, para evitar problemas:

```
#include <stdio.h>
#define DIF(a,b) a - b
int main (void)
{
   printf(" %d ", 4 * DIF(5,3));
   return 0;
}
```

O resultado impresso é 17 e não 8 pois o compilador processa

```
printf(" %d ", 4 * 5 - 3)
```

e a multiplicação tem precedência sobre a subtração

A solução é a inclusão de parênteses envolvendo a macro:

```
#define DIF(a,b) ( (a) – (b) )
```

1.5. Alocação Estática versus Dinâmica

Na alocação estática de vetor é necessário saber de antemão a dimensão máxima do vetor, a variável que representa o vetor armazena o endereço ocupado pelo primeiro elemento do vetor e o vetor declarado dentro do corpo de uma função não pode ser usado fora do corpo da função.

```
#define N 10
int v[N];
```

Na alocação dinâmica de vetor a dimensão do vetor pode ser definida em tempo de execução variável do tipo ponteiro recebe o valor do endereço do primeiro elemento do vetor e a área de memória ocupada pelo vetor permanece válida até que seja explicitamente liberada (através da função free), sendo assim o vetor alocado dentro do corpo de uma função pode ser usado fora do corpo da função, enquanto estiver alocado.

```
int* v;
…
v = (int*) malloc(n * sizeof(int));
```

1.5.1. sizeof

Retorna o número de bytes ocupado por um tipo.

```
v = (int*) malloc(n * sizeof(int));
```

1.5.2. malloc

Recebe como parâmetro o número de bytes que se deseja alocar. Retorna um ponteiro genérico para o endereço inicial da área de memória alocada, se houver espaço livre:

- Ponteiro genérico é representado por void*
- Ponteiro é convertido automaticamente para o tipo apropriado
- Ponteiro pode ser convertido explicitamente

Retorna um endereço nulo, se não houver espaço livre, representado pelo símbolo NULL.

```
v = (int *) malloc(10*sizeof(int));
```

1.5.3. free

Recebe como parâmetro o ponteiro da memória a ser liberada. Deve receber um endereço de memória que tenha sido alocado dinamicamente.

```
free(v);
```

1.5.4. realloc

A função realloc permite re-alocar um vetor preservando o conteúdo dos elementos, que permanecem válidos após a re-alocação. Exemplo: **m** representa a nova dimensão do vetor.

```
v = (int*) realloc(v, m*sizeof(int));
```

1.6. Trabalhando com arquivos texto

Muitas são as situações onde se é necessário utilizar de arquivos do tipo texto em programação, seja para armazenar dados, parâmetros para uma aplicação, auxiliar em testes contínuos com uma mesma massa de informação, para facilitar a manupilação de um conjunto de informações dentre outros. Veremos alguns passos básicos para manipular arquivos deste tipo.

1.6.1. Criação de arquivo

Para usar as funções de manipulação de arquivo você deve incluir a biblioteca **stdio.h** em sua aplicação.

Sempre que for utilizar um arquivo texto em C++ é necessário abrí-lo. Para tanto, a linguagem possui o comando **fopen** onde o primeiro parâmetro é o nome do arquivo, o segundo a forma de abertura:

```
arq = fopen("ArqGrav.txt", "a+");
```

Tabela de parâmetros da forma de abertura da função fopen

Parâmetro	Descrição
"r"	read: Abre arquivo para operações de entrada. O arquivo deve existir.
"w"	write: Cria um arquivo vazio para operações de saída. Se um arquivo com o mesmo nome já existe, seu conteúdo é descartado e o arquivo será tratado como um novo arquivo vazio.
"a"	append: Abrir arquivo para saída no final de um arquivo. Operações de saída sempre escrevem dados no final do arquivo, expandindo-o. Operações de reposicionamento (fseek, fsetpos, rewind) são ignorados. O arquivo é criado se ele não existe.
"r+"	read/update: Abre um arquivo de atualização (tanto para entrada e saída). O arquivo deve existir.
"w+"	write/update: Cria um arquivo vazio e o abre para atualização (tanto para entrada quanto saída). Se um arquivo com o mesmo nome já existir seu conteúdo é descartado e o arquivo será tratado como um novo arquivo vazio.
"a+"	append/update: Abre um arquivo de atualização (tanto para entrada e saída), com todas as operações de saída de gravação de dados no final do arquivo. Operações de reposicionamento (fseek, fsetpos, rewind) afetam as operações de entrada próximos, mas as operações de saída movem a posição de volta para o final do arquivo. O arquivo é criado se ele não existe.

A função fopen retorna um "apontador" para o arquivo caso consiga abrí-lo, caso contrário, retorna a constate NULL.

Exemplo:

```
FILE *arq;
int result;
char Str[50];
arq = fopen("ArqGrav.txt", "rt");
if (arq == NULL)
{   printf("Problemas na CRIACAO do arquivo\n");
    return; }
```

1.6.2. Leitura

Para leitura do conteúdo de arquivos texto pode-se usar a função **fgets** ou **fscanf**.

A função **fgets** lê uma linha inteira de uma vez.

Exemplo:

```
result = fgets(Linha, 100, arq); //lê até 99 caracteres ou até o '\n'
```

Se a função for executada com sucesso, fgets retorna o endereço da string lida, caso contrário retorna NULL.

```
#include <stdio.h>
#include <conio.h>

int main(void)
{
  FILE *arq;
  char Linha[100];
  char *result;
  int i;
  // Abre um arquivo TEXTO para LEITURA
  arq = fopen("ArqTeste.txt", "rt");
  if (arq == NULL)  // Se houve erro na abertura
  {  printf("Problemas na abertura do arquivo\n");
     return 0;  }
  i = 1;
  while (!feof(arq))
  {   // Lê uma linha (inclusive com o '\n')
      result = fgets(Linha, 100, arq);  // lê até 99 caracteres ou até o '\n'
      if (result)  // Se foi possível ler
        printf("Linha %d : %s",i,Linha);
      i++;
  }
  fclose(arq);
}
```

A função **fscanf** funciona como a função SCANF, porém, ao invés de ler os dados de teclado, estes dados são lidos de arquivo.

```
int i, result;
float x;
result = fscanf(arq, "%d%f", &i, &x);
```

Se result for igual à constante EOF, não há mais dados para serem lidos.

1.6.3. Gravação

Para gravação de arquivos texto usa-se as funções **fputs** e **fprintf**.

Exemplo de fputs:

```
result = fputs(Str, arq);
```

Se a função NÃO for executada com sucesso, fputs retorna a constante EOF.

```
char Str[100];
FILE *arq;
arq = fopen("ArqGrav.txt", "wt");  // Cria um arquivo texto para gravação
if (arq == NULL) // Se não conseguiu criar
{  printf("Problemas na CRIACAO do arquivo\n");
   return;
}
strcpy(Str,"Linha de teste");
result = fputs(Str, arq);
if (result == EOF)
    printf("Erro na Gravacao\n");
fclose(arq);
```

Exemplo de fprintf:

```
result = fprintf(arq,"Linha %d\n",i);
```

Se a função fprintf for executada com sucesso, devolve o número de caracteres gravados. Se a função NÃO for executada com sucesso, retorna a constante EOF.

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
  FILE *arq;
  int i;
  int result;
  arq = fopen("ArqGrav.txt", "wt");  // Cria um arquivo texto para gravação
  if (arq == NULL) // Se nào conseguiu criar
  {  printf("Problemas na CRIACAO do arquivo\n");
     return 0;
  }
  for (i = 0; i<10;i++)
  {   // A funcao 'fprintf' devolve o número de bytes gravados
      // ou EOF se houve erro na gravação
      result = fprintf(arq,"Linha %d\n",i);
      if (result == EOF)
        printf("Erro na Gravacao\n");
  }
  fclose(arq);
}
```

1.7. Valores aleatórios

Gerar números aleatoriamente é muito importante principalmente em jogos, para testes em algoritmos e aplicações. Veremos rapidamente como gerar valores de forma randômica.

Porém os computadores também usam algoritmos e códigos para gerarem esses números que devem ser, teoricamente, aleatórios. Porém, nunca serão totalmente aleatórios, pois são gerados por código, por funções e tarefas específicas.

Uma boa tentativa de deixarem os números mais aleatórios possíveis, 'alimentar' seu algoritmo com números diferentes. Obviamente, o código de uma função que gera números aleatórios é sempre o mesmo, mas se você fornecer números diferentes, ela vai gerar seqüencias diferentes de números aleatórios.

Esses números que fornecemos são chamados de semente, ou seed. E vamos usar o tempo (sim, o tempo cronológico), como semente.

1.7.1. rand

A função que gera números aleatórios em C++ é a rand(). Ela gera números entre 0 e **RAND_MAX**, onde esse RAND_MAX é um valor que pode variar de máquina pra máquina. Pra usar a função rand(), temos que adicionar a biblioteca **time.h** e para saber o valor de RAND_MAX, temos que usar a função **stdlib.h**.

Exemplo de um programa que gera 10 números aleatórios:

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

int main(void)
{   int i;
    printf("intervalo da rand: [0,%d]\n", RAND_MAX);
    for(i=1 ; i <= 10 ; i++)
        printf("Numero %d: %d\n",i, rand());
}
```

Esperimente rodar este programa e depois feche e rode o programa novamente. E de novo, de novo...notou diferença nos valores? Bom o programa não está errado, ele irá gerar os mesmos números aleatórios... Nem tanto aleatórios assim.

Para mudar isso, vamos alimentar a função rand() com uma semente, com um número, que é o tempo atual.

Assim, toda vez que rodarmos o programa, a rand() pega um número de tempo diferente e gera uma seqüencia diferente.

Para fazer isso, basta usar a função **srand**(), que será responsável por alimentar a rand(). Fazemos isso adicionando o comando: **srand((unsigned)time(NULL));** antes da rand().

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main()
{   int i;
    printf("intervalo da rand: [0,%d]\n", RAND_MAX);
    srand( (unsigned)time(NULL) );
    for(i=1 ; i <= 10 ; i++)
        printf("Numero %d: %d\n",i, rand());
}
```

Porém, seu código gera números entre 0 e RAND_MAX.

E se você quiser gerar entre 0 e 10? Ou entre 1 e mil? Ou entre 0.00 e 1.00?

Para escolher a faixa de valores vamos usar operações matemáticas, principalmente o operador de módulo, também conhecido como resto da divisão: %

Para fazer com que um número 'x' receba um valor entre 0 e 9, fazemos:

```
x = rand() % 10
```

Para fazer com que um número 'x' receba um valor entre 1 e 10, fazemos:

```
x = 1 + ( rand() % 10 )
```

Para fazer com que um número 'x' receba um valor entre 5 e 10, fazemos:

```
x = 5+(rand()%6)
```

Para fazer com que um número 'x' receba um valor entre 0.00 e 1.00, primeiro geramos números inteiros, entre 0 e 100:

```
x = rand() % 101
```

Para ter os valores decimais, dividimos por 100:

```
x = x/100;
```

1.8. Estruturas básicas

Vamos rapidamente relembrar algumas estruturas elementares, tal como vetor, matriz, estrutura, fila, pilha, lista e árvore, para depois vemos os algoritmos de ordenação.

1.8.1. Vetores

É uma estrutura de dados definindo um conjunto enumerável.

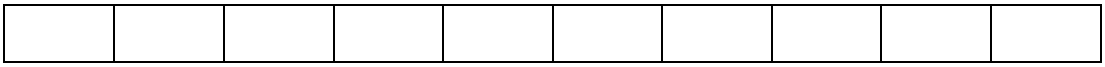
vetor = 

Figura de um vetor de 10 posições (da posição 0 a posição 9)

Exemplo: v = vetor de inteiros com 10 elementos.

```
int v[10];
```

Um vetor pode ser inicializado.

```
int v[5] = { 5, 10, 15, 20, 25 };

// ou simplesmente:

int v[] = { 5, 10, 15, 20, 25 };
```

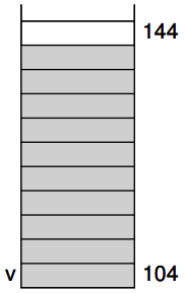
O acesso a cada elemento é feito através de indexação da variável.

```
int v[10];  // declara o vetor com 10 posições
v[0] = 0;   // acessa o primeiro elemento de v
...
v[9] = 9;   // acessa o último elemento de v
v[10] = 10; // ERRADO (invasão de memória)
```

O vetor é alocado em posições contíguas de memória.

Exemplo: v = vetor de inteiros com 10 elementos

Espaço de memória de v = 10 x valores inteiros de 4 bytes = 40 bytes



1.8.2. Matrizes (ou vetor bidimensional)

Uma matriz é um vetor bidimensional e são indexados de forma dupla: m[i][j], onde i indexa a linha e j indexa a coluna. A indexação se inicia em zero (0) e a variável pode ser inicializada na declaração.

Sendo assim m[0][0] é o elemento da primeira linha e primeira coluna.

```
float mat[4][3];
```

1.8.3. Estruturas

É um tipo de dado com campos compostos por tipos mais simples. Os elementos são acessados através do operador de acesso "ponto" (.).

1.8.3.1. struct

```
struct ponto /* declara ponto do tipo struct */
{ float x;
  float y;
};
struct ponto p; /* declara p como variável do tipo struct ponto */
p.x = 10.0; /* acessa os elementos de ponto */
p.y = 5.0;
```

1.8.3.2. typedef

Permite criar nomes de tipos, útil para abreviar nomes de tipos e para tratar tipos complexos.

```
typedef float Vetor[4];
Vetor v; /* exemplo de declaração usando Vetor */
v[0] = 3;
```

Vetor neste caso, é um tipo que representa um vetor de quatro elementos.

Exercício – Tipos e estruturas

Criar uma tabela com dados de alunos, organizada em um vetor. Os dados de cada aluno são: matrícula: número inteiro; nome: cadeia com até 80 caracteres; endereço: cadeia com até 120 caracteres; telefone: cadeia com até 20 caracteres.

```
struct aluno {
   int mat;
   char nome[81];
   char end[121];
   char tel[21];
};
```

```
typedef struct aluno Aluno;
#define MAX 100
Aluno tab[MAX];
```

Nesta solução a estrutura **Aluno** ocupa pelo menos 4+81+121+21 = 227 Bytes e **tab** é um vetor de Aluno e representa um desperdício significativo de memória, se o número de alunos é bem inferior ao máximo estimado e utilizando da forma a seguir isso não ocorre.

```
Aluno* tab[MAX];
```

Criar uma função para inicializar a tabela. Ela recebe um vetor de ponteiros (parâmetro deve ser do tipo "ponteiro para ponteiro") e atribui NULL a todos os elementos da tabela.

```
void inicializa (int n, Aluno** tab)
{ int i;
  for (i=0; i<n; i++)
    tab[i] = NULL;
}
```

Criar uma função para armazenar novo aluno na tabela. Ela recebe a posição onde os dados serão armazenados, os dados são fornecidos via teclado e se a posição da tabela estiver vazia, função aloca nova estrutura, caso contrário, função atualiza a estrutura já apontada pelo ponteiro.

```
void preenche (int n, Aluno** tab, int i)
{ if (i==n)
      { printf("Indice fora do limite do vetor\n");
        exit(1); /* aborta o programa */
      }
  if (tab[i]==NULL)
      tab[i] = (Aluno*) malloc(sizeof(Aluno));
  printf("Entre com a matricula:");
  scanf("%d", &tab[i]->mat); // ... podemos acrescentar os demais atributos
}
```

Criar uma função para remover os dados de um aluno da tabela. Ela recebe a posição da tabela a ser liberada e libera espaço de memória utilizado para os dados do aluno.

```
void retira (int n, Aluno** tab, int i)
{ if (i==n)
      { printf("Indice fora do limite do vetor\n");
        exit(1); /* aborta o programa */
      }
  if (tab[i] != NULL)
     { free(tab[i]);
       tab[i] = NULL; /* indica que na posição não mais existe dado */
     }
}
```

Criar uma função para imprimir os dados de um aluno da tabela. Ela recebe a posição da tabela a ser impressa.

```
void imprime (int n, Aluno** tab, int i)
{ if (i==n)
      { printf("Indice fora do limite do vetor\n");
        exit(1); /* aborta o programa */
      }
  if (tab[i] != NULL)
      { printf("Matrícula: %d\n", tab[i]->mat);
       printf("Nome: %s\n", tab[i]->nome);
       printf("Endereço: %s\n", tab[i]->end);
       printf("Telefone: %s\n", tab[i]->tel);
     }
}
```

Criar uma função para imprimir todos os dados da tabela. Ela recebe o tamanho da tabela e a própria tabela.

```
void imprime_tudo (int n, Aluno** tab)
{ int i;
  for (i=0; i<n; i++)
      imprime(n,tab,i);
}
```

Programa teste:

```
int main (void)
{ Aluno* tab[10];
  inicializa(10,tab);
  preenche(10,tab,0);
  preenche(10,tab,1);
  preenche(10,tab,2);
  imprime_tudo(10,tab);
  retira(10,tab,0);
  retira(10,tab,1);
  retira(10,tab,2);
  return 0;
}
```

1.8.4. Enumeração

O tipo enum declara uma enumeração, ou seja, um conjunto de constantes inteiras com nomes que especifica os valores legais que uma variável daquele tipo pode ter. Oferece uma forma mais elegante de organizar valores constantes.

```
enum bool { TRUE = 1, FALSE = 0 };
typedef enum bool Bool;
Bool resultado;
```

Neste caso acima, **bool** - declara as constantes FALSE e TRUE, associando TRUE ao valor 1 e FALSE ao valor 0. **Bool** - declara um tipo cujos valores só podem ser TRUE (1) ou FALSE (0) e **resultado** - variável que pode receber apenas os valores TRUE ou FALSE

1.8.5. Listas

É uma sequência encadeada de elementos, chamados de nós da lista. Cada nó da lista é representado por dois campos: a informação armazenada e o ponteiro para o próximo elemento da lista. A lista é representada por um ponteiro para o primeiro nó e o ponteiro do último elemento é NULL.

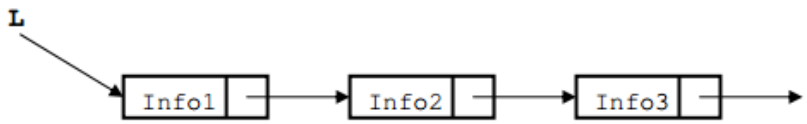


Figura demonstrando uma estrutura de lista

```
struct lista { int info;
               struct lista* prox; };
typedef struct lista Lista;

/* função de criação: retorna uma lista vazia */
Lista* lst_cria (void)
{     return NULL; }
```

```
/* inserção no início: retorna a lista atualizada */
Lista* lst_insere (Lista* l, int i)
{     Lista* novo = (Lista*) malloc(sizeof(Lista));
      novo->info = i;
      novo->prox = l;
      return novo;
}

/* função imprime: imprime valores dos elementos */
void lst_imprime (Lista* l)
{     Lista* p;
      for (p = l; p != NULL; p = p->prox)
            printf("info = %d\n", p->info);
}

/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int lst_vazia (Lista* l)
{     return (l == NULL); }

/* função busca: busca um elemento na lista */
Lista* busca (Lista* l, int v)
{     Lista* p;
      for (p=l; p!=NULL; p = p->prox)
            { if (p->info == v)
                  return p;
      }
      return NULL; /* não achou o elemento */
}

/* função retira: retira elemento da lista */
Lista* lst_retira (Lista* l, int v)
{     Lista* ant = NULL; /* ponteiro para elemento anterior */
      Lista* p = l; /* ponteiro para percorrer a lista */
      /* procura elemento na lista, guardando anterior */
      while (p != NULL && p->info != v)
      {    ant = p;
           p = p->prox;
      }
      /* verifica se achou elemento */
      if (p == NULL)
           return l; /* não achou: retorna lista original */
      /* retira elemento */
      if (ant == NULL)
      { /* retira elemento do inicio */
            l = p->prox;
      } else
      { /* retira elemento do meio da lista */
            ant->prox = p->prox;
      }
      free(p);
      return l; }

/* destrói a lista, liberando todos os elementos alocados */
void lst_libera (Lista* l)
{     Lista* p = l;
      while (p != NULL)
      {     Lista* t = p->prox; /* guarda referência p/ próx.elemento */
           free(p); /* libera a memória apontada por p */
           p = t; /* faz p apontar para o próximo */
      }
}

/* cria uma lista inicialmente vazia e insere novos elementos */
```

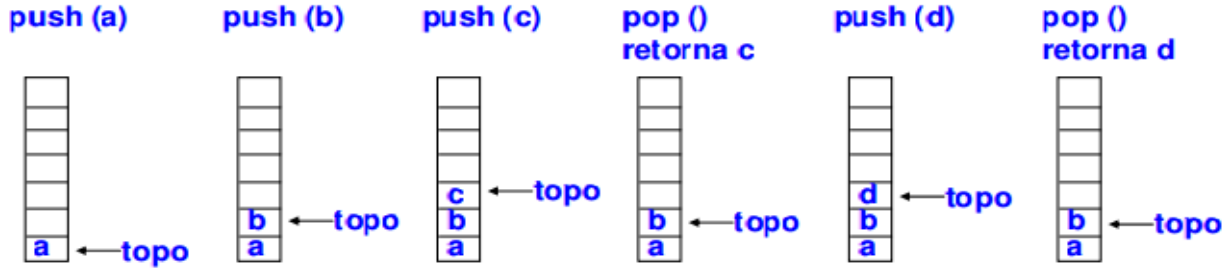
```
int main (void)
{     Lista* l; /* declara uma lista não inicializada */
      l = lst_cria(); /* cria e inicializa lista como vazia */
      l = lst_insere(l, 23); /* insere na lista o elemento 23 */
      l = lst_insere(l, 45); /* insere na lista o elemento 45 */
      lst_imprime(l); /* imprime a lista */
      return 0;
}
```

1.8.6. Pilhas

Nas pilhas, o novo elemento é inserido no topo e acesso é apenas ao topo, isto é, o primeiro que sai é o último que entrou (LIFO – last in, first out). As operações básicas das pilhas são:

• empilhar (push) um novo elemento, inserindo-o no topo;

• desempilhar (pop) um elemento, removendo-o do topo.




```
/* definição de tipos */
typedef struct pilha Pilha;
Pilha* pilha_cria (void);
void pilha_push (Pilha* p, float v);
float pilha_pop (Pilha* p);
int pilha_vazia (Pilha* p);
void pilha_libera (Pilha* p);

#define N 50 /* número máximo de elementos */

/* Implementação de pilha com vetor */
struct pilha {
      int n; /* vet[n]: primeira posição livre do vetor */
      float vet[N]; /* vet[n-1]: topo da pilha */
      /* vet[0] a vet[N-1]: posições ocupáveis */
};

/* aloca dinamicamente um vetor */
Pilha* pilha_cria (void)
{     Pilha* p = (Pilha*) malloc(sizeof(Pilha));
      p->n = 0; /* inicializa com zero elementos */
      return p;
}

/* insere um elemento na pilha */
void pilha_push (Pilha* p, float v)
{     if (p->n == N) { /* capacidade esgotada */
           printf("Capacidade da pilha estourou.\n");
           exit(1); /* aborta programa */
      }
      /* insere elemento na próxima posição livre */
      p->vet[p->n] = v;
      p->n++; /* equivalente a: p->n = p->n + 1 */
}
```

```
/* retira o elemento do topo da pilha, retornando o seu valor */
float pilha_pop (Pilha* p)
{     float v;
      if (pilha_vazia(p)) { printf("Pilha vazia.\n");
      exit(1); } /* aborta programa */
      /* retira elemento do topo */
      v = p->vet[p->n-1];
      p->n--;
      return v;
}

/* destrói a pilha, liberando todos os elementos alocados */
void pilha_libera (Pilha* p)
{   free(p); /* libera a memória apontada por p */
}

/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int pilha_vazia (Pilha* p)
{      if(p->n == 0) return 1;
       return 0;
}
```

1.8.7. Filas

Em filas, um novo elemento é inserido no final da fila e um elemento é retirado do início da fila, isto é, o primeiro que entra é o primeiro que sai (FIFO – first in, first out).

```
/* definição de tipos */
typedef struct fila Fila;
Fila* fila_cria (void);
void fila_insere (Fila* f, float v);
float fila_retira (Fila* f);
int fila_vazia (Fila* f);
void fila_libera (Fila* f);

/* Implementação de fila com vetor */
#define N 100 /* número máximo de elementos */
struct fila {
      int n; /* número de elementos na fila */
      int ini; /* posição do próximo elemento a ser retirado da fila */
      float vet[N];
};

/* aloca dinamicamente um vetor */
Fila* fila_cria (void)
{     Fila* f = (Fila*) malloc(sizeof(Fila));
      f->n = 0; /* inicializa fila como vazia */
      f->ini = 0; /* escolhe uma posição inicial */
      return f;
}

/* insere um elemento no final da fila */
void fila_insere (Fila* f, float v)
{     int fim;
      if (f->n == N) { /* fila cheia: capacidade esgotada */
            printf("Capacidade da fila estourou.\n");
            exit(1); } /* aborta programa */
      /* insere elemento na próxima posição livre */
      fim = (f->ini + f->n) % N;
      f->vet[fim] = v;
      f->n++;
}
```

```
/* retira o elemento do início da fila, retornando o seu valor */
float fila_retira (Fila* f)
{     float v;
      if (fila_vazia(f)) {
            printf("Fila vazia.\n");
            exit(1); /* aborta programa */
      }
      /* retira elemento do início */
      v = f->vet[f->ini];
      f->ini = (f->ini + 1) % N;
      f->n--;
      return v;
}

/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int fila_vazia (Fila* f)
{     if (f->n == 0) return 1;
      return 0; }

/* destrói a fila, liberando todos os elementos alocados */
void fila_libera (Fila* f)
{   free(f); } /* libera a memória apontada por p */
```

1.8.8. Árvores

Árvores são estruturas de dados extremamente úteis em muitas aplicações. Uma árvore é formada por um conjunto finito *T* de elementos denominados vértices ou nós de tal modo que se T = 0 a árvore é vazia, caso contrário temos um nó especial chamado raiz da árvore (*r*), e cujos elementos restantes são particionados em *m*>=1 conjuntos distintos não vazios, as subárvores de *r*, sendo cada um destes conjuntos por sua vez uma árvore.

A seguir vemos duas figuras. A forma convencional de representar uma árvore está indicada na figura da esquerda. Esta árvore tem nove nós sendo *A* o nó raiz. Os conjuntos das subárvores tem de ser disjuntos, portanto a estrutura indicada na figura da direita não é uma árvore.

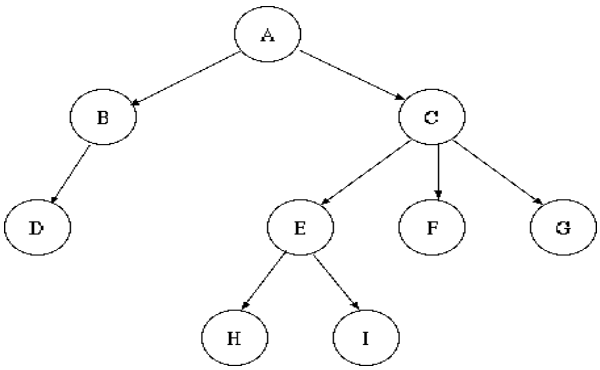


Figura representando uma árvore

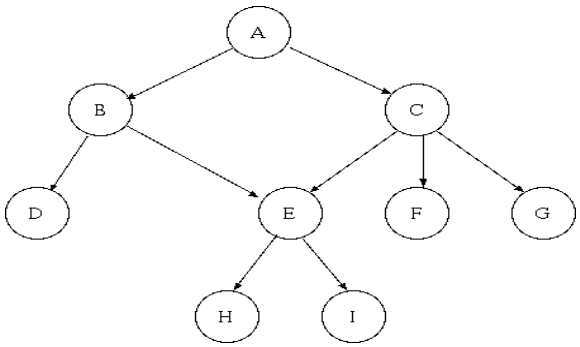


Figura de uma estrutura que não representa uma árvore

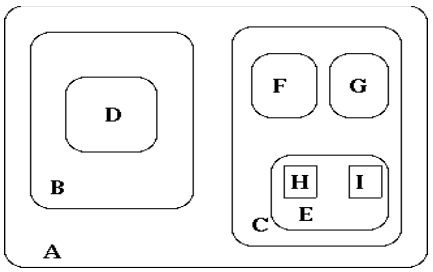
Os nós *B* e *C* são filhos de *A* e nós irmãos. Nós sem filhos como os nós *D, H, I, F e G* são chamados de folhas. A subárvore da esquerda do nó *A* tem raiz em *B* e a subárvore da direita tem raiz em *C*, isto está indicado pelos dois ramos saindo de *A*. A ausência de um ramo na árvore indica uma subárvore vazia, como a subárvore da direita do nó *B*. O número de filhos de um nó é chamado de grau de saída deste nó. Por exemplo, o nó *C* tem grau de saída 3 e o nó *E* grau 2.

Um caminho da árvore é composto por uma sequência de nós consecutivos (n_1, n_2, ..., n_{k-1}, n_k) tal que existe sempre a relação n_i é pai de n_{i+1}. Os *k* vértices formam *k-1* pares e um caminho de comprimento igual a *k*-1. O comprimento do caminho entre o nó *A* e o nó ***H*** é 3.

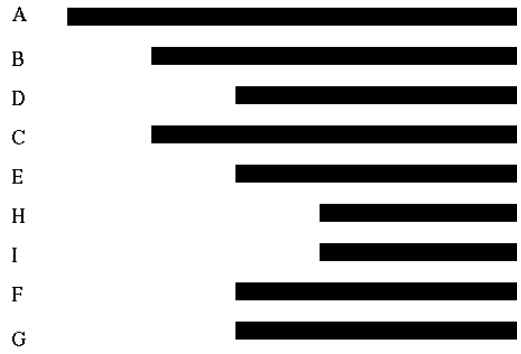
O nível de um nó n pode ser definido do seguinte modo: o nó raiz tem nível 0, os outros nós têm um nível que é maior uma unidade que o nível de seu pai. Na árvore da figura anterior temos nós nos seguintes níveis:

nível 0 = A
nível 1 = B, C
nível 2 = D, E, F, G
nível 3 = H, I

Existem diversas maneiras de representar árvores. Uma representação que reflete a ideia de árvores como conjuntos aninhados é mostrado na figura ao lado. A figura mostra o mesmo conjunto da árvore anterior.



Outra notação que encontramos a toda hora, e que está representada na figura ao lado, é a forma endentada ou de diagrama de barras. Notar que esta representação lembra um sumário de livro. Os sumários dos livros são representações da árvore do conteúdo do livro.



Uma outra forma interessante de representar uma árvore é a representação por parênteses aninhados. A sequência de parênteses representa a relação entre os nós da estrutura. O rótulo do nó é inserido à esquerda do abre parênteses correspondente.

$$(A(B(D))\left(C(E(H)(I))(F)(G)\right))$$

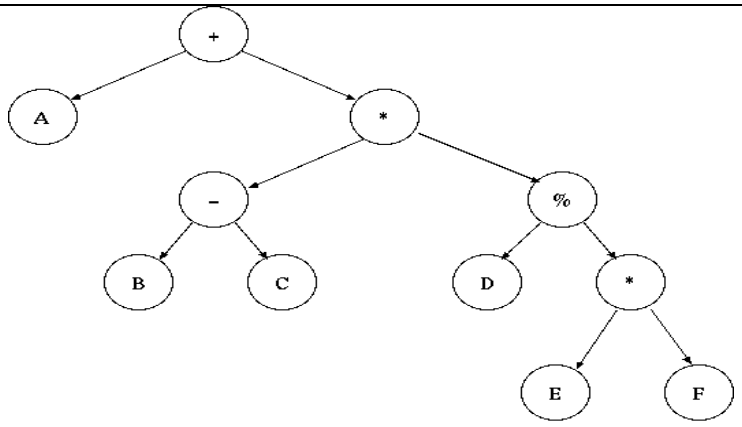
Esta representação tem importância, por exemplo, no tratamento de expressões aritméticas, já que toda expressão aritmética pode ser colocada nesta forma. Se colocarmos uma expressão nesta forma podemos então representá-la como uma árvore, mostrando como ela seria calculada. Para colocarmos uma expressão em forma de árvore devemos considerar cada operador como um nó da árvore e os seus operandos como as duas subárvores. Considere a expressão seguinte:

$$A + (B - C) * D\%(E * F)$$

Que após receber todos os parênteses fica da seguinte maneira:

$$(A + \left((B - C) * \left(D\%(E * F)\right)\right))$$

A figura ao lado mostra como fica esta expressão representada por uma árvore.



(A+((B−C)*(D%(E*F))))

Árvores Binárias

A figura abaixo mostra um importante tipo de árvore que é a árvore binária. Árvores binárias podem ser vistas em diversas situações do cotidiano. Por exemplo, um torneio de futebol eliminatório, do tipo das copas dos países, como a Copa do Brasil, em que a cada etapa os times são agrupados dois a dois e sempre são eliminados metade dos times é uma árvore binária.

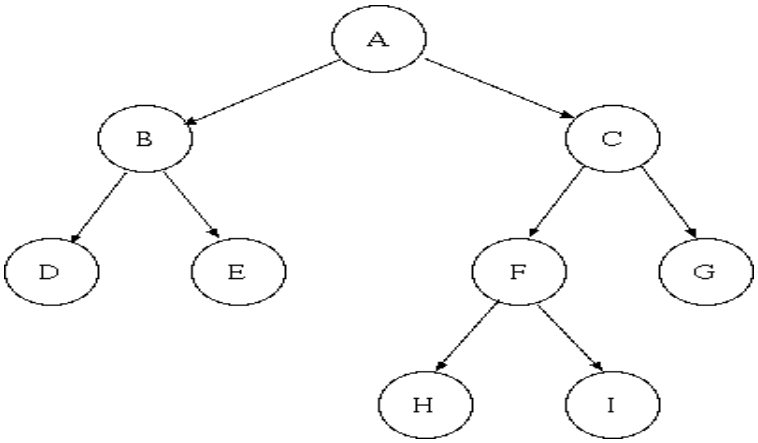


Figura representa uma árvore binária

Considere a figura a seguir, note que são duas árvores idênticas, mas são duas árvores binárias diferentes. Isto porque uma delas tem a subárvore da direita vazia e a outra a subárvore da esquerda.

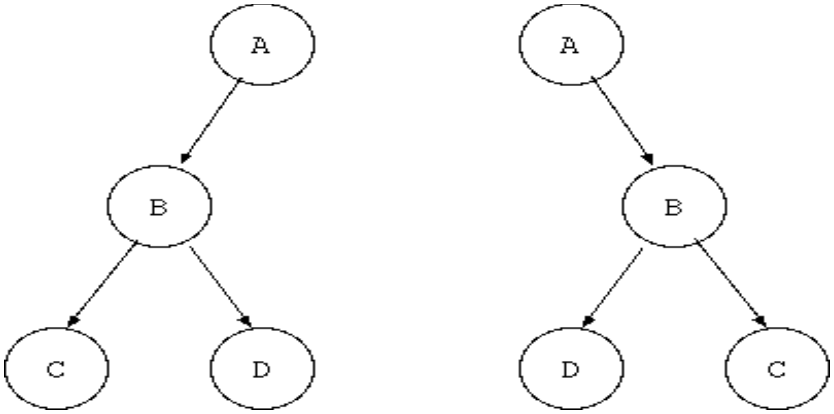
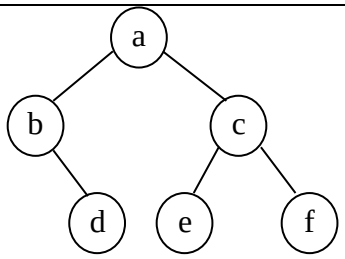


Figura representa árvores binárias diferentes

```
/* Representação de uma árvore */
struct arv {
      char info;
      struct arv* esq;
      struct arv* dir;
};

typedef struct arv Arv;
Arv* arv_criavazia (void);
Arv* arv_cria (char c, Arv* e, Arv* d);
Arv* arv_libera (Arv* a);
int arv_vazia (Arv* a);
int arv_pertence (Arv* a, char c);
void arv_imprime (Arv* a);

/* cria uma árvore vazia */
Arv* arv_criavazia (void)
{     return NULL;
}

/* cria um nó raiz dadas a informação e as sub-árvores (esquerda, direita) */
Arv* arv_cria (char c, Arv* sae, Arv* sad)
{     Arv* p=(Arv*)malloc(sizeof(Arv));
      p->info = c;
      p->esq = sae;
      p->dir = sad;
      return p;
}

/* libera memória alocada pela estrutura da árvore */
Arv* arv_libera (Arv* a)
{     if (!arv_vazia(a)){
            arv_libera(a->esq); /* libera sae */
            arv_libera(a->dir); /* libera sad */
            free(a); /* libera raiz */
      }
      return NULL;
}

/* indica se uma árvore é ou não vazia */
int arv_vazia (Arv* a)
{ return a==NULL;
}

/* verifica a ocorrência de um caractere c em um de nós */
int arv_pertence (Arv* a, char c)
{     if (arv_vazia(a))
            return 0; /* árvore vazia: não encontrou */
      else
            return a->info==c ||
            arv_pertence(a->esq,c) ||
            arv_pertence(a->dir,c);
}

/* varre recursivamente árvore, visitando os nós e imprimindo a informação */
void arv_imprime (Arv* a)
{     if (!arv_vazia(a))
      {     printf("%c ", a->info); /* mostra raiz */
            arv_imprime(a->esq); /* mostra sae */
            arv_imprime(a->dir); /* mostra sad */
      }
}
```

Criando uma árvore como está ao lado, as chamadas de arv_cria ficariam assim:

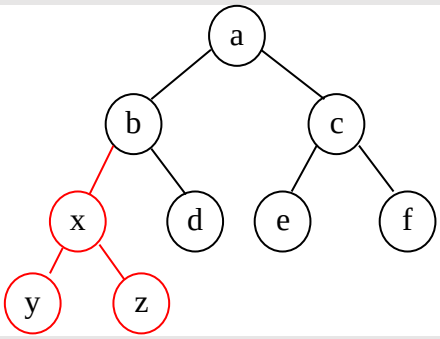



```
/* sub-árvore 'd' */
Arv* a1= arv_cria('d',arv_criavazia(),arv_criavazia());
/* sub-árvore 'b' */
Arv* a2= arv_cria('b',arv_criavazia(),a1);
/* sub-árvore 'e' */
Arv* a3= arv_cria('e',arv_criavazia(),arv_criavazia());
/* sub-árvore 'f' */
Arv* a4= arv_cria('f',arv_criavazia(),arv_criavazia());
/* sub-árvore 'c' */
Arv* a5= arv_cria('c',a3,a4);
/* árvore 'a' */
Arv* a = arv_cria('a',a2,a5);
```

Ou assim:

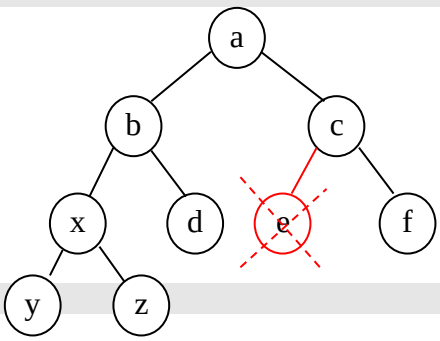
```
Arv* a = arv_cria('a',
            arv_cria('b',
                        arv_criavazia(),
                        arv_cria('d', arv_criavazia(), arv_criavazia()
                                )
                        ),
            arv_cria('c',
                      arv_cria('e', arv_criavazia(), arv_criavazia()),
                       arv_cria('f', arv_criavazia(), arv_criavazia())
                    )
           );
```

Acrescentando nós na árvore como esta ao lado, as chamadas ficariam assim:




```
a->esq->esq = arv_cria('x',
                       arv_cria('y',
                             arv_criavazia(),
                             arv_criavazia()),
                       arv_cria('z',
                             arv_criavazia(),
                             arv_criavazia())
                       );
```

Liberando nós na árvore como esta ao lado, as chamadas ficariam assim:




```
a->dir->esq = arv_libera(a->dir->esq);
```

Recursividade

Recursividade não é um comando, mas uma "habilidade" de uma função chamar a si mesma. Mas é importante entender este conceito, pois ele é muito útil e pode ajudar a deixar seu algoritmo muito mais simples.

Fazer a função chamar ela mesma é simples, basta escrever no código da função, a função que está sendo criada como se ela já tivesse sido criada antes. Parece estranho isso, mas vamos imaginar um exemplo simples: Temos um programa - sabemos que o código do programa está todo dentro da função MAIN - se quisermos reiniciar o programa, basta chamarmos a função MAIN novamente.

Muito provavelmente deve estar imaginando: No que isso pode ser usado? Bem, existem casos onde é necessário realizar um processo ou uma conta com um resultado anterior da sequência. Um algoritmo muito pedido é a famosa sequência de Fibonacci. Essa sequência consiste nos dois primeiros elementos possuírem valor fixo 1, e os termos seguintes são, cada um, a soma dos dois termos anteriores. Ou seja, o 3º termo é a soma de 1+1 que é 2; o 4º termo é a soma de 1+2 que é 3; o 5º termo é a soma de 2+3 que é 5 e assim sucessivamente.

Isso resulta em: 1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - 34 - 55 - 89 ...

Para fazer a conta acima usaremos uma função que chamaremos de fibonacci e esta utilizará de recursividade:

```
#include <stdio.h>
#include <conio.h>
int fibonacci(int);
main()
{
  int n,i;
  printf("Digite a quantidade de termos da sequencia de Fibonacci: ");
  scanf("%d", &n);
  printf("\nA sequencia de Fibonacci e: \n");
  for(i=0; i<n; i++)
     printf("%d ", fibonacci(i+1));
  getch(); // retorna imediatamente após uma tecla ser pressionada
}

int fibonacci(int num)
{
  if(num==1 || num==2)
     return 1;
  else
     return fibonacci(num-1) + fibonacci(num-2);
}
```

Exemplo do algoritmo para gerar a série de Fibonacci de forma recursiva em linguagem C

Em uma função recursiva, a cada chamada é criada na memória uma nova ocorrência da função com comandos e variáveis “isolados” das ocorrências anteriores.

A função é executada até que todas as ocorrências tenham sido resolvidas.

Porém um problema que surge ao usar a recursividade é como fazê-la parar. Caso o programador não tenha cuidado é fácil cair num loop infinito recursivo o qual pode inclusive esgotar a memória.

Toda recursividade é composta por um **caso base** e pelas **chamadas recursivas**.

Caso base: é o caso mais simples. É usada uma condição em que se resolve o problema com facilidade.

Chamadas recursivas: procuram simplificar o problema de tal forma que convergem para o caso base

Vantagens da recursividade

- Torna a escrita do código mais simples e elegante, tornando-o fácil de entender e de manter.

Desvantagens da recursividade

- Quando o loop recursivo é muito grande é consumida muita memória nas chamadas a diversos níveis de recursão, pois cada chamada recursiva aloca memória para os parâmetros, variáveis locais e de controle.
- Em muitos casos uma solução iterativa gasta menos memória, e torna-se mais eficiente em termos de performance do que usar recursão.

```c
#include <stdio.h>
#include <conio.h>
int fatorial(int n);
int main(void)
{
  int numero;
  double f;
  printf("Digite o numero que deseja calcular o fatorial: ");
  scanf("%d",&numero);
  printf("Fatorial de %d = %d",numero, fatorial(numero));
  getch();
  return 0;
}

//Função recursiva que calcula o fatorial de um numero inteiro n
int fatorial(int n)
{
  int vfat;
  if ( n <= 1 )
     return (1); //Caso base: fatorial de n <= 1 retorna 1
  else
  {
     vfat = n * fatorial(n - 1);  //Chamada recursiva
     return (vfat);
  }
}
```

Exemplo do algoritmo para calcular fatorial de forma recursiva em linguagem C

No programa acima, se o número n for menor ou igual a 1 o valor 1 será retornado e a função encerrada, sem necessidade de chamadas recursivas. Caso contrário dá-se início a chamadas recursivas até cair no caso mais simples que é resolvido e assim, as chamadas retornam valores de forma a solucionar o cálculo.

Veja a figura a seguir que representa as chamadas recursivas para o cálculo de 5!

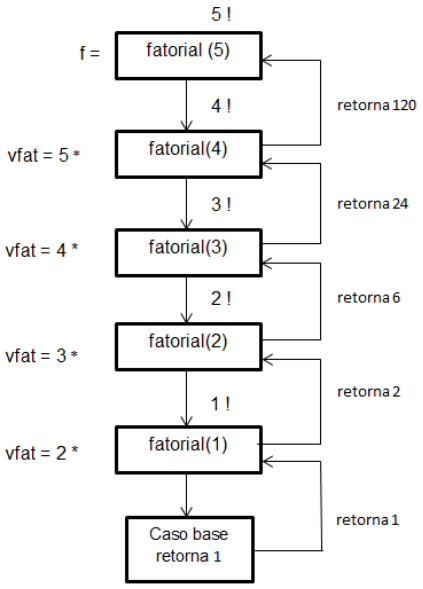


Figura demonstrando a execução do fatorial com recursividade

Exercício – Soma recursiva

Utilizando recursividade, crie um programa em C que peça um número inteiro ao usuário e retorne a soma de todos os números de 1 até o número que o usuário introduziu, ou seja: $1+2+3+...+n$

```
#include <stdio.h>
 int soma(int n)
{   if(n == 1) return 1;
    else return ( n + soma(n-1) );
}
int main()
{   int n;
    printf("Digite um inteiro positivo: ");
    scanf("%d", &n);
    printf("Soma: %d\n", soma(n));
}
```

Exemplo do algoritmo para calcular soma de forma recursiva em linguagem C

Exercício – Torre de Hanói

A lenda diz que num templo perdido na Ásia uns monges estão tentando mover 64 discos de tamanhos diferentes de um pino para outro, usando um terceiro como auxiliar, de tal forma que nunca um disco maior é colocado sobre um menor. De acordo com a lenda, o mundo se acaba no momento que esta tarefa for completada.

A tarefa é escrever um programa que calcula o movimento de n discos de acordo com as regras estabelecidas. Se o valor fornecido para o programa for 3, então a sequência de chamadas e as saídas geradas são:

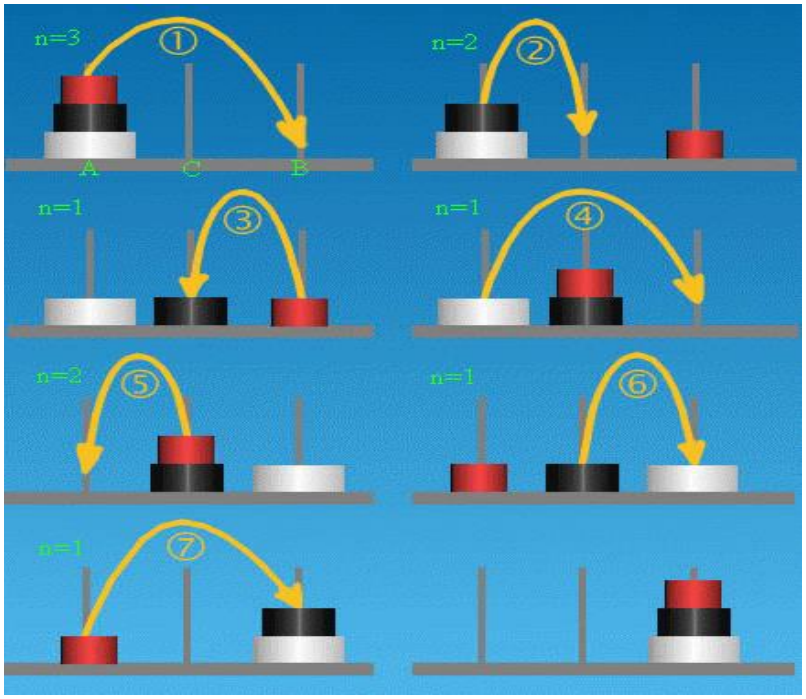


Figura demonstrando a execução da torre de Hanói para 3 discos

```
#include <stdio.h>
int movim;
// mova n discos do pino a para o pino b usando o pino c como intermediario
void hanoi(int n, char a, char b, char c) {
  movim++;
  if (n == 1)
    printf("mova o disco %d de %c para %c\n", n, a, b);
  else {
    hanoi(n - 1, a, c, b);
    printf("mova o disco %d de %c para %c\n", n, a, b);
    hanoi(n - 1, c, b, a);
  }
}

int main(void)
{
  int numDiscos;
  printf("Quantos discos?");
  scanf("%d", &numDiscos);
  hanoi(numDiscos, 'A', 'B', 'C');
  printf("realizados %d movimentos",movim);
  return 0;
}
```

Exemplo do algoritmo para calcular os movimentos da torre de Hanói em linguagem C

O programa vai efetuar 2^n-1 chamadas. Para n=64 o número de chamadas e consequentemente de movimentos é 18446744073709551615. Se os monges movem um disco a cada segundo e se eles não errarem nenhum movimento, então o mundo se acaba em aproximadamente 585 bilhões anos após o movimento do primeiro disco.

REFERÊNCIAS

ASCENCIO, Ana Fernanda Gomes. Aplicações das estruturas de dados em DELPHI. São Paulo: Pearson, 2005

GUIMARÃES, A. M., LAGES, N. A. Algoritmos e estrutura de dados. Rio de Janeiro: LTC, 2007.

MARKENZON. L. e SZWAR-CFITER, J. L., Estruturas de dados e seus algoritmos. Rio de Janeiro: Campus, 2009.

PEREIRA, Sílvio Lago; Estrutura de dados fundamentais. 12.ed. São Paulo: Érica. 2009.

ZIVIANI, Nivio. Estruturas de dados com implementação em C e PASCAL. São Paulo: Pearson, 2008.

ZIVIANI, Nivio. Estruturas de dados com implementação em JAVA e C++ São Paulo: CENGAGE 2011.