Algoritmos e Programação Estruturada

Algoritmo

Os algoritmos são as bases para criação de um programa de computador, onde diversas aplicações poderão ocorrer. Um algoritmo bem estruturado vai gerar um programa para solução de um problema que antes, parecia complexo. Todas as áreas estão voltadas para a tecnologia e são através de diversas formas de pensamentos que os algoritmos são realizados.

Algoritmo é uma sequência ordenada de passos que deve ser seguida para a realização de uma tarefa (Berg e Figueiró, 1998).

Os algoritmos nortearão a descobrir qual o melhor percurso para solucionar um problema computacional.

A elaboração de algoritmos é um passo importante para o desenvolvimento de um programa de computador (ou software), pois, a partir da construção de algoritmos para a resolução de algum problema, é possível traduzir o algoritmo para alguma linguagem de programação.

A seguir, veja alguns exemplos de algoritmo:

Algoritmo para efetuar o cozimento de um arroz:

- Acender o fogo.
- Refogar os temperos.
- Colocar o arroz na panela.
- Colocar a água.
- Cozinhar o arroz.
- Abaixar o fogo.
- Esperar o ponto.
- Desligar o fogo.
- Servir o arroz.

Algoritmo para efetuar o cozimento de um arroz (mais detalhado):

- Comprar o arroz.
- Analisar a qualidade.
- Realizar a pré-seleção para o cozimento.
- Preparar o tempero.
- Pegar a panela.
- Acender o fogo.
- Colocar os temperos na panela para refogar.
- Adicionar o arroz.
- Colocar a água na medida considerada ideal para a quantidade.
- Aguardar a água secar.
- Baixar o fogo.
- Fechar a panela com a tampa.
- Aguardar o ponto.
- Desligar o fogo.
- Servir o arroz.

Assim, percebemos que não existe somente uma forma de realizar um algoritmo, podem ser criadas outras formas e sequências para obter o mesmo resultado, ou seja, eles são independentes, porém, com a mesma finalidade de execução.

Representação

O algoritmo é representado em três partes:

- **Entrada:** Dados (ingredientes para o preparo do arroz).
- **Processamento:** Execução (cozimento do arroz).
- **Saída:** Solução/objetivo atingido (finalização do arroz – momento que será servido).

Linguagem Natural

A linguagem natural na definição geral é uma forma de comunicação entre as pessoas de diversas línguas, ela pode ser falada, escrita, gesticulada entre outras formas de comunicação. A linguagem natural tem uma grande contribuição quando vamos desenvolver uma aplicação computacional, pois ela pode direcionar de forma simples e eficiente as descrições dos problemas e suas soluções (Santos, 2001).

Para reforçar os conceitos de linguagem natural podemos ver como exemplo o cadastro de notas de alguns alunos de um curso.

Problema

O usuário deverá entrar com dois valores (as notas) e o computador retorna o resultado da média destes valores (média das notas).

Perceba que a linguagem natural é muito próxima da nossa linguagem.

Solução:

- Início.
- Entrar com o primeiro valor (nota do primeiro bimestre).
- Entrar com o segundo valor (nota do segundo bimestre).
- Realizar a soma do primeiro valor com o segundo.
- Realizar a divisão do total dos valores por dois (média das notas dos bimestres).
- Armazenar o valor encontrado.
- Mostrar na tela o resultado da média.
- Se a média do aluno for maior ou igual a seis.
- O aluno será considerado aprovado.
- Senão será reprovado.
- Fim.

Outro exemplo é o algoritmo para calcular o máximo divisor comum, o famoso “MDC”:

- Dividir um número “a” por “b”, onde o resto é representado por “r”.
- Substituir a por b.
- Substituir b por r.

- Continuar a divisão de a por b até que um não possa ser mais dividido, então “a” é considerado o MDC.

De acordo com a solução, o resultado fica: MDC(480, 130) = 10.

a	b	r
480	130	90
130	90	40
90	40	10
40	10	0
10	0	

Variáveis e Atribuições

As variáveis, como o próprio nome sugere, é algo que pode sofrer variações, ou seja, estão relacionadas a identificação de uma informação. Exemplos: valor1, nome.

A atribuição tem a função de indicar valores para as variáveis, ou seja, atribuir informação para variável. Exemplos:

valor1 ← 8

nome ← “márcio”

Significa que o número 8 está sendo atribuído para variável “valor1”, e que o texto “márcio” está atribuído para a variável “nome”.

Diagrama de Blocos (Fluxograma)

Diagrama de blocos é um conjunto de símbolos gráficos, onde cada um desses símbolos representa ações específicas a serem executadas pelo computador. Determina a linha de raciocínio utilizada pelo programador para resolver problemas. Os símbolos dos diagramas de bloco forma padronizados pela ANSI (Instituto Norte-Americano de Padronização).

A seguir, veja a descrição dos principais símbolos utilizados em um diagrama de blocos, de acordo com Manzano (2015):

- **Terminal:** Representa o início ou o fim de um fluxo lógico. Em alguns casos define as sub-rotinas.
- **Entrada Manual:** Determina a entrada manual dos dados, geralmente através de um teclado.
- **Processamento:** Representa a execução de ações de processamento.
- **Exibição:** Mostra o resultado de uma ação, geralmente através da tela de um computador.
- **Decisão:** Representa os desvios condicionais nas operações de tomada de decisão e laços condicionais para repetição de alguns trechos do programa.
- **Preparação:** Representa a execução de um laço incondicional que permite a modificação de instruções do laço.

- **Processo Definido:** Mostra o resultado de uma ação, geralmente através da tela de um computador.
- **Conector:** Representa pontos de conexões entre trechos de programas, que podem ser apontados para outras partes do diagrama de bloco.
- **Linha:** Representa os vínculos existentes entre os símbolos de um diagrama de blocos.

Pseudocódigo

O pseudocódigo é considerado uma ferramenta que pode auxiliar a programação, ela pode ser escrita em palavras similares ao inglês ou português para facilitar a interpretação e desenvolvimento de um programa (Aguilar, 2011).

Exemplo de pseudocódigo que calcula a média das notas dos alunos de um curso:

```
Algoritmo "calculo_media"

Var
  valor1, valor2, soma, media: real
Inicio
  Escreva("Digite o valor da nota 1: ")
  Leia(valor1)
  Escreva("Digite o valor da nota 2: ")
  Leia(valor2)

  soma <- (valor1 + valor2)
  media <- (soma / 2)

  Escreval("A média do aluno é ", media)

  se media >= 6 entao
    Escreval("Aluno aprovado")
  senao
    Escreval("Aluno reprovado")
  fimse
Fimalgoritmo
```

Perceba que os parâmetros utilizados também são considerados um algoritmo do tipo português estruturado, ou seja, de fácil entendimento e interpretação.

Paradigmas de Programação

Após os estudos de algoritmos e as suas formas de construções, Manzano (2015) coloca em destaque os paradigmas de programação, que são caracterizados pelos paradigmas da:

- **Programação Estruturada:** Onde o algoritmo é construído como sequência linear de funções ou módulo.
- **Programação Orientada a Objetos:** Onde o programador abstrai um programa como uma coleção de objetos que interagem entre si.

Os algoritmos são as bases para criação de um programa de computador, onde diversas aplicações poderão ocorrer. Um algoritmo bem estruturado vai gerar um programa para solução de um

problema que antes, parecia complexo. Todas as áreas estão voltadas para a tecnologia e são através de diversas formas de pensamentos que os algoritmos são realizados.

Hoje podemos contar com a ajuda de softwares específicos para construção de diagrama de blocos (fluxogramas), entre eles, você pode usar o Lucidchart que é um gerador de fluxograma online e gratuito.

Outro software muito utilizado é o Dia. Você pode fazer o download no site de Dia Diagram Editor (2020).

Foguete Ariane 5

Você conhece o caso do foguete Ariante 5?

Trata-se de um projeto da Agência Espacial Europeia de um primeiro voo não tripulado em 4 de junho de 1996, que custou bilhões de dólares e levou 10 anos para ser construído. O foguete Ariane 5 explodiu 40 segundos após a decolagem em seu voo inaugural. Com isso, houve a destruição do foguete e de toda a carga que custava milhões de dólares, causando um enorme prejuízo financeiro e de tempo.

E tudo isso não aconteceu por uma falha mecânica, mas por um erro de programação.

O programa que convertia um valor em float (ponto flutuante) para um inteiro de 16 bits (long int) recebeu como entrada um valor fora da faixa que suportava (um estouro de inteiros). Com esse bug, os computadores principais, inclusive o de backup, desligaram ao mesmo tempo, alegando u merro de execução (run time error). Por isso, é importante destacar que a declaração de uma variável de forma incorreta ou com tipos incompatíveis pode trazer erros muitas vezes catastróficos. Esse desastre mostra a importância da declaração de variáveis corretamente.

Bug do Milênio (Y2K)

Você sabia que no fim do século XX houve uma grande mobilização no mundo e em uma comunidade de programadores, devido ao chamado “bug do milênio”?

Esse foi o nome dado para um provável acontecimento – que não se concretizou – na virada do ano de 1999 para 2000: Havia o receio de que os sistemas da época não reconhecessem as datas do ano 2000 e retornassem para 1900.

Mas por que isso?

Na época, recursos de memória eram caros e limitados – por exemplo, 1 MB de memória custava em torno de 700 Dólares. Por isso, havia uma constante necessidade, por parte dos desenvolvedores, de economizar e otimizar espaço em memória. Assim, na década de 1960, as datas eram armazenadas, porém, eram interpretadas apenas dois dígitos para o ano – os dois últimos dígitos.

No caso, o ano de 1999 seria interpretado apenas como “99” (dois últimos dígitos). Com isso, o receio era que, em vez de os sistemas reconhecerem o ano de 2000, o identificassem como ano de 1900, zerando os dois últimos dígitos na data.

De fato, o problema não causou tantos impactos, pois muitos sistemas desenvolvidos na época já consideravam o ano com quatro dígitos. Porém, problemas pontuais ocorreram: Alguns sites da

época não reconheciam a data e mostravam a data “01/01/19100”, e houve falhas em terminais de ônibus e equipamentos de medição de radiação. Houve, também, investimentos de bilhões de Dólares em medidas preventivas.
Caso realmente tivesse se concretizado, esse acontecimento causaria um enorme prejuízo para bancos, com juros negativos, investidores e empresas poderiam ir à falência. Os sistemas de aeroportos e usinas nucleares poderiam entrar em colapso, causando quedas de aviões e vazamentos radioativos. Atualmente é improvável que esse tipo de problema ocorra, pois já temos alternativas de armazenamento em nuvem e recursos de memória mais baratos.

Para exercitar um pouco, observe o código com o erro que se assemelha ao ocorrido no caos do foguete Ariane 5. Mostra que os erros de execução (run time error) são comuns quando se declara um vetor ou array com menor capacidade que o necessário para o problema. Nesse caso, o código é compilado sem erros, porém, ocorrem problemas na execução.

Analise o código e faça o teste:

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

int main() {
   setlocale(LC_ALL, "portuguese");

   char nome[4];
   int ano;

   printf("Digite o seu nome e sobrenome: ");
   scanf("%s", nome);
   printf("Digite o ano de nascimento: ");
   scanf("%d", &ano);

   printf("\nNome digitado: %s\n", nome);
   printf("Ano de nascimento: %d\n", ano);

   return 0;
}
```

Observe o erro “Stack smashing detected”, que consiste em um erro de overflow, a alocação do vetor “char” apresenta 4 posições e o nome e sobrenome na entrada ultrapassam 4 caracteres.

Sistemas Computacionais

Desde o momento em que você liga um computador (ou tablet, ou smartphone), centenas de processos são inicializados e passam a competir pelo processador para que possam ser executados e fazer a “mágica” do mundo digital acontecer.

Todos os resultados desses sistemas são obtidos através do processamento de dados e nesta aula começaremos a estudar os recursos que lhe permitirão implementar soluções com processamento.

Os sistemas computacionais são construídos para resolver os mais diversos problemas. Todos esses sistemas, independentemente da sua aplicação, são construídos em três partes: Entrada, processamento e saída.

Nos três casos, a leitura dos dados é feita para um único fim: Processamento e geração de informações e essa etapa é construída a partir da combinação de operações aritméticas, relacionais, lógicas e outras técnicas de programação.

Operadores Aritméticos

Vamos começara a aprimorar nossos algoritmos através das operações aritméticas. Os operadores aritméticos podem ser classificados em unários ou binários (Manzano, 2015).

Operadores aritméticos unários:

Operador	Descrição	Exemplo	Resultado
++	Pós-incremento	*x++*	*x + 1*
++	Pré-incremento	*++x*	*x + 1*
--	Pós-decremento	*y--*	*y - 1*
--	Pré-decremento	*--y*	*y - 1*

Quando trabalhamos com operadores, a ordem de procedência é muito importante. Segundo Soffner (2013), os operadores aritméticos possuem a seguinte ordem de execução:

1. Parênteses.
2. Potenciação e radiciação.
3. Multiplicação, divisão e módulo.
4. Soma e subtração.

Operadores relacionais

Faz parte do processamento fazer comparações entre valores e, a partir do resultado, realizar novas ações. Em programação, para compararmos valores usamos operadores relacionais. Os operadores relacionais são utilizados para construir expressões booleanas, ou seja, expressões que terão como resultado verdadeiro ou falso.

Operadores relacionais:

Operador	Descrição	Exemplo
==	Igual a	*x == y*
!=	Diferente de	*x != y*
>	Maior que	*x > y*
<	Menor que	*x < y*
>=	Maior ou igual que	*x >= y*
<=	Menor ou igual que	*x <= y*

Observe o código a seguir, de acordo com as entradas *n1 = 5*, *n2 = 10* e *n3 = 5*:

- A instrução *(n1 == n2) && (n1 == 3)*, mostrará se n1 é igual a n2 e (&&) é igual a n3. No caso será impresso o valor 0 (falso), pois n1 e n2 são diferentes.
- A instrução *(n1 == n2)* || *(n1 == n3)*, mostrará se n1 é igual a n1 ou (||) é igual a n3. No caso será impresso o valor 1 (verdadeiro), pois, os valores de n1 e n2 são iguais.
- A instrução *(n1 < n3)* || *(n1 > n2)*, mostrará se n1 é menor que n3 ou (||) n1 é maior que n2. No caso será impresso o valor 0 (falso), pois, os valores de n1 e n3 são iguais e n1 é maior que n2.

Código abaixo:

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

int main() {
   setlocale(LC_ALL, "portuguese");

   int n1 = 5, n2 = 10, n3 = 5;

   printf("(n1 == n2) && (n1 == n3) = %d\n", ((n1 == n2) && (n1 == n3)));
   printf("(n1 == n2) || (n1 == n3) = %d\n", ((n1 == n2) || (n1 == n3)));
   printf("(n1 < n3) || (n1 > n2) == %d\n", ((n1 < n3) || (n1 > n2)));

   return 0;
}
```

Operadores Lógicos

Além dos operadores relacionais, outro importante recurso para o processamento é a utilização de operadores lógicos, que possuem como fundamento a lógica matemática clássica e a lógica booleana (Gersting, 2017).

Operadores lógicos:

Operador	Descrição	Exemplo
!	Negação (not-não)	*!(x == y)*
&&	Conjunção (and-e)	*(x > y) && (a == b)*
||	Disjunção (or-ou)	*(x > y) || (a == b)*

Observe no código a seguir, que na linha 14 *if(strcmp(login, “admin”) == 0 && strcmp(senha, “123”) == 0)* utilizamos os operadores == e *&&*.

- O operador == compara se as strings de entrada em login é igual a “admin” e senha é igual a “123”.
- O operador && verifica se o login e a senha correspondem os valores de entradas:

Código abaixo:

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

```
#include <locale.h>
#include <string.h>

int main() {
   setlocale(LC_ALL, "portuguese");

   char login[10];
   char senha[10];

   printf("Entre com seu login: ");
   scanf("%s", login);
   printf("Digite a sua senha: ");
   scanf("%s", senha);

   if(strcmp(login, "admin") == 0 && strcmp(senha, "123") == 0) {
      printf("Bem vindo, %s.\n", login);
   }
   else {
      printf("Login falhou!\n");
   }

   return 0;
}
```

Funções Predefinidas

Para facilitar o desenvolvimento de soluções em software, cada linguagem de programação oferece um conjunto de funções predefinidas que ficam à disposição dos programadores. Entende-se por função “um conjunto de instruções que efetuam uma tarefa específica.” (Manzano, 2015).

Algumas bibliotecas e funções na linguagem C:

Biblioteca	Função	Descrição
<stdio.h>	*printf()*	Imprime na tela.
	scanf()	Faz a leitura de um dado digitado.
	fgets(variavel, tamanho, fluxo)	Faz a leitura de uma linha digitada.
<math.h>	*pow(base, potencia)*	Operação de potenciação.
	sqrt(numero)	Calcula a raiz quadrada.
	sin(angulo)	Calcula o seno de um ângulo.
	cos(angulo)	Calcula o cosseno de um ângulo.
<string.h>	*strcmp(string1, string2)*	Verifica se duas strings são iguais.
	strcpy(destino, origem)	Copia uma string da origem para o destino.

<stdlib.h>	*malloc(tamanho)*	Aloca dinamicamente espaço na memória.
	realloc(local, tamanho)	Modifica um espaço já alocado dinamicamente.
	free(local)	Libera um espaço alocal dinamicamente.

Com essa aula, exploramos as formas de armazenar temporariamente os dados em diversos tipos de variáveis e como podemos utilizar os operadores para realizar o processamento dos dados. Não se esqueça de recorrer ao livro didático para aprofundar o seus estudos.

Estrutura de Decisão Condicional if/else (se/senão)

Para a solução de um problema que envolva situações podemos utilizar a instrução “if”, em português “se”, onde sua função é tomar uma decisão e criar um desvio dentro do programa, desta forma, podemos chegar a uma condição verdadeira ou falsa. Lembrando que a instrução pode receber valores em ambos os casos (Manzano, 2013).

Estrutura Condicional Simples

Sintaxe da instrução “if” (se) utilizada na linguagem C:

```
if(condicao) {
   // Comandos
}
```

No exemplo de condicional simples, será executado um teste lógico, onde, se o resultado for verdadeiro então ele trará uma resposta, caso contrário não retornará nada.

No exemplo a seguir, não é considerado o senão (else), simplesmente se a condição não for verdadeira ela não exibirá nada como resposta:

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

int main() {
   setlocale(LC_ALL, "portuguese");

   float idade;

   printf("Digite sua idade: ");
   scanf("%f", &idade);

   if(idade >= 18) {
      printf("Você já pode tirar sua carteira de habilitação! Você é maior de 18 anos!\n");
   }

   return 0;
}
```

Estrutura Condicional Composta

Sintaxe da instrução if/else (se/senão):

```
if(condicao) {
   // Primeiro conjunto de comandos
}
else {
  // Segundo conjunto de comandos
}
```

No exemplo de estrutura condicional composta a seguir, Maria e João estão se preparando para uma viagem, porém, se o orçamento final deles for igual ou maior que R$ 10.000,00 eles farão uma viagem internacional, senão deverão ficar com uma viagem nacional.

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

int main() {
   setlocale(LC_ALL, "portuguese");

   float orcamento;

   printf("Digite o valor do orçamento: ");
   scanf("%f", &orcamento);

   if(orcamento >= 10000) {
      printf("João e Maria possuem orçamento para uma viagem internacional, pois seu orçamento
é de R$ %.2f.\n", orcamento);
   }
   else {
      printf("João e Maria irão optar por uma viagem nacional, seu orçamento ficou em R$ %.2f.\
n", orcamento);
   }

   return 0;
}
```

Estrutura Condicional de Seleção de Casos “switch-case”

A estrutura condicional de seleção de casos “switch-case” testa sucessivamente o valor de uma expressão contra uma lista de constantes inteiras ou de caractere. Quando os valores são avaliados o comando é executado.

Sintaxe da instrução “switch-case” (seleção de casos):

```
switch(variavel) {
   case constante1:
      // Bloco 1
      break;
```

```
    case constante2:
        // Bloco 2
        break;
    default:
       // Bloco default
       break;
}
```

No exemplo de estrutura condicional de seleção de casos a seguir, é aplicado um descondo de acordo com a escolha de uma cor específica pelo cliente.

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

int main() {
    setlocale(LC_ALL, "portuguese");

    char x;
    float valor, desc, total;

    printf("Digite o valor da compra: ");
    scanf("%f", &valor);
    printf("Digite a letra que representa o seu desconto de acordo com a cor!\n\n");

    printf("A - Azul\nV - Vermelho\nB - Branco\n\n");

    printf("Digite sua opção: ");
    scanf("\n%c", &x);

    switch(x) {
        case 'a':
        case 'A':
            printf("Você escolheu azul, seu desconto será de 30%%.\n");

            desc = valor * 0.30;
            total = valor - desc;

            printf("O valor da sua compra é de R$ %.2f\n", total);
            break;
        case 'v':
        case 'V':
            printf("Você escolheu vermelho, seu desconto será de 20%%.\n");

            desc = valor * 0.20;
            total = valor - desc;

            printf("O valor da sua compra é de R$ %.2f.\n", total);
            break;
        case 'b':
        case 'B':
            printf("Você escolheu branco, seu desconto será de 10%%.\n");
```

```
            desc = valor * 0.10;
            total = valor - desc;

            printf("O valor da sua compra é de R$ %.2f.\n", total);
            break;
        default:
            printf("Opção Inválida!");
            break;
    }

    return 0;
}
```

Estrutura Condicional Encadeada

Conhecida como ifs aninhados. É um comando if que é o objeto de outros if e eles. Ou seja, sempre um comando eles estará ligado ao comando if de seu nível de aninhamento (Schildt, 1997).

Sintaxe:

```
if(condicao) {
    // Comandos
else {
    if(condicao2) {
        // Comandos
    }
    else {
        // Comandos
    }
}
```

No exemplo de estrutura condiciona encadeada a seguir, será analisado os tipos de triângulo, partindo da premissa que ele deverá ser testado antes, para ver se forma ou não um triângulo:

Código:

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

int main() {
    setlocale(LC_ALL, "portuguese");

    int a, b, c;

    printf("Classificação do triângulo: Informe a medida dos lados apertando a tecla Enter para cada medida: ");
    scanf("%d%d%d", &a, &b, &c);

    if(a < b + c && b < a + c && c < a + b) {
        printf("\nDadas as medidas: %d, %d, %d, temos um triângulo!\n", a, b, c);
```

```
        if(a == b && a == c) {
            printf("Este é um triângulo equilátero!\n");
        }
        else {
            if(a == b || a == c || b == c) {
                printf("Este é um triângulo isósceles!\n");
            }
            else {
                printf("Este é um triângulo escaleno!\n");
            }
        }
    }
    else {
        printf("\nAs medidas fornecidas, %d, %d, %d não formam um triângulo!\n", a, b, c);
    }

    return 0;
}
```

Vimos as estruturas condicionais e de seleção. Pense nas possibilidades que você pode ter usando essas estruturas de tomadas de decisão “if-else”, “if-else-if” e “switch-case”. Lembre-se que para cada caso poderá ter uma particularidade diferente em desenvolver um programa.

Estruturas de Repetição Condicional

Nesta aula vamos ver sobre as estruturas de repetição condicional.

Estrutura de Repetição Condicional While/Do While

Assim como nas estruturas de decisão, as estruturas de repetição desenvolvem aplicações para a otimização do pensamento computacional e ao mesmo tempo, a agilidade nas soluções dos problemas de repetição.

Estrutura de Repetição com Teste no Início – While

Neste caso algo será repetidamente executado enquanto uma condição verdadeira for verificada, somente após a sua negativa essa condição será interrompida.

Na realização dessa condição, o comando iterativo “while”, que significa “enquanto” em português, realiza o teste no início, antes de executar as ações programadas.

Quando utilizamos teste no início, pode ocorrer o famoso loop (laço) infinito, que é quando um processo é executado repetidamente. Para que isso não aconteça, você poderá utilizar os seguintes recursos:

- **Contador:** É utilizado para controlar as repetições, quando esta é determinada.
- **Incremento e Decremento:** Trabalham o número do contador, seja ele, aumentando ou diminuindo.
- **Acumulador:** Irá somar as entradas de dados de cada iteração da repetição, gerando um somatório a ser utilizado quando da saída da repetição.

- **Condição de Parada:** Utilizada para determinar o momento de parar quando não se tem um valor exato desta repetição.

No exemplo de repetição condicional a seguir, utilizando o comando while com teste no início, mostrará a palavra “Programa” dez vezes:

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

int main() {
   setlocale(LC_ALL, "portuguese");

   int cont = 0;

   while(cont < 10) {
      printf("Programa!\n");
      cont++;
   }

   return 0;
}
```

Estrutura de Repetição Condicional com Teste no Final – Do While

O laço “Do While” analisa a condição ao final do laço, ou seja, os comandos são executados antes do teste de condição.

Neste caso, em específico, o usuário tem a possibilidade de digitar novamente uma nova informação (Schildt, 1997).

O exemplo a seguir, realiza um programa que calcula a metragem quadrada de um terreno usando o teste no final para criar a opção de digitar novos valores sem sair do programa:

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

int main() {
   setlocale(LC_ALL, "portuguese");

   float metr1, metr2, resul;
   int resp;

   metr1 = 0;
   metr2 = 0;
   resul = 0;

   do {
      printf("=== CÁLCULO DE METROS QUADRADOS ===\n\n");
      printf("Digite a primeira metragem do terreno: ");
      scanf("%f", &metr1);
```

```
        printf("Digite a segunda metragem do terreno: ");
        scanf("%f", &metr2);

        resul = metr1 * metr2;

        printf("\nO terreno tem %.2f M².\n\n", resul);

        printf("Digite 1 para continuar ou 2 para sair: ");
        scanf("%d", &resp);

        printf("\n");
    }
    while(resp == 1);

    return 0;
}
```

O exemplo a seguir, realiza um programa que simula uma conta bancária (com tela de opções das transações. Ele repete uma entrada de dados até que determinada condição de saída seja atingida e, em seguida, acumule os valores digitados. Observe que foi utilizado o laço do while para implementar o menu do programa, uma estrutura de repetição usando comparativo:

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

int main() {
    setlocale(LC_ALL, "portuguese");

    float soma = 0;
    float valor;
    int opcao;

    do {
        printf("Digite uma operação!\n\n");
        printf("1 - Depósito\n");
        printf("2 - Saque\n");
        printf("3 - Saldo\n");
        printf("4 - Sair\n\n");
        printf("Opção: ");
        scanf("%d", &opcao);

        printf("\n");

        switch(opcao) {
            case 1:
                printf("Valor do depósito: ");
                scanf("%f", &valor);
                soma += valor;
                break;
            case 2:
                printf("Valor do saque: ");
```

```
            scanf("%f", &valor);
            soma -= valor;
            break;
        case 3:
            printf("Saldo atual: R$ %.2f.\n", soma);
            break;
        default:
            if(opcao != 4) printf("Opção Inválida!\n");
            break;
      }

      printf("\n");
   }
   while(opcao != 4);

   printf("Fim das Operações!\n");

   return 0;
}
```

O comando do while pode ter várias aplicações.

Estruturas de Repetição Determinísticas

Nesta aula vamos ver a aplicação do laço “for”, ou seja, a estrutura de repetição com variáveis de controle.

Estrutura de Repetição com Variáveis de Controle – For

O comando iterativo “for”, que em português significa “para”, é geralmente usado para repetir uma informação por um número fixo de vezes, isto é, podemos determinar quantas vezes acontecerá a repetição (Mizrahi, 2008).

Na aplicação do comando “for” há três expressões separadas por ponto e vírgula: Inicialização, condição final e incremento.

- **Inicialização:** Neste momento, coloca-se a instrução de atribuição. A inicialização é executada uma única vez antes de começar o laço.
- **Condição Final:** Realiza-se um teste que determina se a condição é verdadeira ou falsa, se for verdadeira, permanece no laço e, se for falsa, encerra o laço e passa para a próxima instrução.
- **Incremento:** Parte das nossas explicações anteriores, em que é possível incrementar uma repetição de acordo com um contador específico, lembrando que o incremento é executado depois dos comandos.

A seguir, veremos alguns exemplos de utilização do comando for.

Como primeiro exemplo, iremos criar uma contagem regressiva de um número qualquer, digitado pelo usuário:

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

```
#include <locale.h>

int main() {
   setlocale(LC_ALL, "portuguese");

   int cont;

   printf("Digite um número para contagem regressiva: ");
   scanf("%d", &cont);

   for(cont; cont >= 1; cont--) {
      printf("%d\n", cont);
   }

   return 0;
}
```

Pode-se usar o comando “break” dentro de um laço para uma determinada condição, forçando assim o término do laço:

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

int main() {
   setlocale(LC_ALL, "portuguese");

   int w;

   for(w = 1; w <= 15; w++) {
      if(w == 8) {
         break;
      }

      printf("%d\n", w);
   }

   printf("A condição parou no número %d.\n", w);

   return 0;
}
```

No exemplo a seguir, temos um programa que mostra uma sequência de números, onde x vai de 10 a 0 e y vai de 0 a 10.

Na primeira expressão “x” tem o seu valor iniciado em 10 e “y” iniciado em 0.

Na segunda expressão o laço se repetirá enquanto n for maior ou igual a n e enquanto y for menor ou igual a 10.

Ao final da execução dos comandos do laço de repetição, x será decrementado de 1 e y será incrementado de 1:

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

int main() {
   setlocale(LC_ALL, "portuguese");

   int x, y;

   for(x = 10, y = 0; x >= 0, y <= 10; x--, y++) {
      printf("x = %2d, y = %2d.\n", x, y);
   }

   return 0;
}
```

Aplicações com Vetores

Vetor (array) é um tipo especial de variável capaz de armazenar diversos valores “ao mesmo tempo” usando um mesmo endereço na memória.

O exemplo a seguir mescla o comando for com while. O programa encontra a primeira posição para um determinado número inserido pelo usuário:

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

int main() {
   setlocale(LC_ALL, "portuguese");

   int numero;
   int i;
   int posicao = 0;
   int vetor[10];

   printf("Entre com o número de até três casas, diferente de zero, a ser procurado em um vetor de
10 posições: ");
   scanf("%d", &numero);

   printf("\n");

   for(i = 0; i < 10; i++) {
      printf("Entre com o número para a posição %02d: ", i);
      scanf("%d", &vetor[i]);
   }

   while(vetor[posicao] != numero) {
      posicao++;
   }
```

```
    printf("\n");

    for(i = 0; i < 10; i++) {
        printf("%03d\n", vetor[i]);
    }

    return 0;
}
```

Instrução Continue

Uma instrução continue dentro de um laço possibilita que a execução de comandos corrente seja terminada, passando à próxima iteração do laço.

No exemplo a seguir, temos um programa que percorrerá os números de 1 a 30 e neste percurso, irá testar se for digitado algum número ímpar, caso seja ímpar o programa continua o teste até o fim do laço:

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

int main() {
    setlocale(LC_ALL, "portuguese");

    int i;

    for(i = 1; i <= 100; i++)
        if(i == 30)
            break;
        else
            if(i % 2 == 1)
                continue;
            else
                printf("%2d\n", i);

    printf("Término do laço!\n");

    return 0;
}
```

Aplicações com Matrizes

Matrizes são arranjos de duas ou mais dimensões.

O exemplo a seguir monta uma matriz 3 x 3, onde os valores são lançados de acordo com a linha e coluna:

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

```
int main() {
   setlocale(LC_ALL, "portuguese");

   int linha, coluna;
   int matriz[3][3];

   for(linha = 0; linha < 3; linha++) {
      for(coluna = 0; coluna < 3; coluna++) {
         printf("Digite os valores da matriz para [%d][%d]: ", linha, coluna);
         scanf("%d", &matriz[linha][coluna]);
      }
   }

   printf("\nVeja a sua matriz:\n\n");

   for(linha = 0; linha < 3; linha++) {
      for(coluna = 0; coluna < 3; coluna++) {
         printf("%d ", matriz[linha][coluna]);
      }
      printf("\n");
   }

   return 0;
}
```

Procedimentos e Funções

Um software deve ser construído de forma organizada onde cada funcionalidade deve ser colocada em um “local” com uma respectiva identificação, para que o requisitante possa encontrá-la. Uma das técnicas de programação utilizada para construir programas dessa forma é a construção de funções. Assim, nesta aula vamos ver a criação de funções na linguagem C bem como seu tipo de retorno.

Funções da Linguagem C

printf() e *scanf()* são exemplos de funções que fazem parte das bibliotecas da linguagem C.

Na imagem a seguir, o comando na linha 2, *int main()* especifica uma função que chama *main* e que irá devolver para que a requisitou um valor inteiro, nesse caso, zero:

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

int main() {
   setlocale(LC_ALL, "portuguese");

   printf("Olá, Mundo!\n");

   return 0;
}
```

A ideia de criar programas com blocos de funcionalidades vêm de uma técnica de projeto de algoritmos chamada dividir para conquistar (Manzano, Matos, Lourenço, 2015). A ideia é simples, dado um problema, este deve ser dividido em problemas menores, que facilitem a resolução e organização. A técnica consiste em três passos:

- **Dividir:** Quebrar um problema em outros subproblemas menores.
- **Conquistar:** Usar uma sequência de instruções separada, para resolver cada subproblema.
- **Combinar:** Juntar a solução de cada subproblema para alcançar a solução completa do problema original.

Função

Uma função é um trecho de código escrito para solucionar um subproblema. Esses blocos são escritos tanto para dividir a complexidade de um problema maior, quanto para evitar a repetição de códigos. Essa técnica também pode ser chamada de modularização, ou seja, um problema será resolvido em diferentes módulos.

Em cada declaração da função alguns parâmetros são obrigatórios e outros opcionais. Veja-os a seguir:

- **Tipo de Retorno:** Obrigatório. Esse parâmetro indica qual o tipo de valor a função irá retornar. Pode ser um valor inteiro (int), decimal (float ou double), caractere (char), etc. Quando a sub-rotina faz um processamento e não retorna nenhum valor, usa-se o parâmetro void e, nesse caso, é chamado de procedimento (Manzano, 2015).
- **Nome:** Obrigatório. Parâmetro que especifica o nome que identificará a função. É como o nome de uma pessoa, para você convidá-la para sair você precisa “chamá-la pelo nome”. O nome não pode ter acento, nem caractere especial e nem ser nome composto (mesmas regras para nomes de variáveis).
- **Parênteses Depois do Nome:** Obrigatório. Toda função ou procedimento, sempre terá o nome acompanhado de parênteses. Por exemplo, *main()*, *printf()*, *somar()*, etc.
- **Parâmetros:** Opcional. Estudaremos mais adiante.
- **Comandos da Função:** Obrigatório. Só faz sentido criar uma função se ela tiver um conjunto de comandos para realizar.
- **Retorno:** Quando o tipo de retorno for void esse parâmetro não precisa ser usado, porém, quando não for void é obrigatório. O valor a ser retornado tem que ser compatível com o tipo de retorno, senão o programa dará um erro de compilação em algumas linguagens, em outras retornará um valor errôneo. Na linguagem C, irá ser retornado um valor de acordo com o tipo.

Local da Função

Em qual parte do código a função deve ser programada?

Veja a seguir, um exemplo de programa que utiliza uma função para calcular a soma entre dois números:

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

int somar();
```

```
int main() {
    setlocale(LC_ALL, "portuguese");

    int resultado = 0;

    resultado = somar();

    printf("O resultado da função é %d.\n", resultado);

    return 0;
}

int somar() {
    return 2 + 3;
}
```

Outra característica da utilização de funções é que estas “quebram” a linearidade de execução, pois a execução pode “dar saltos” quando uma função é invocada (Soffner, 2013).

Para entender melhor como funciona esse mecanismo, veja a seguir uma função que solicita um número para o usuário, calcula o quadrado desse número e retorna o resultado:

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

float calcular();

int main() {
    setlocale(LC_ALL, "portuguese");

    float resultado = 0;

    resultado = calcular();

    printf("O quadrado do número digitado é %.2f.\n", resultado);

    return 0;
}

float calcular() {
    float num;

    printf("Digite um número: ");
    scanf("%f", &num);

    return num * num;
}
```

O Uso de Funções com Ponteiros

Uma função pode retornar um número inteiro, um real e um caractere, assim como também pode retornar um vetor. Para isso, devemos utilizar ponteiros (ou apontador). A única forma de retornar um vetor é por meio de um ponteiro, pois não é possível criar funções como por exemplo, *int[10] calcular()*, onde *int[10]* quer dizer que a função retorna um vetor com 10 posições (Manzano, 2015).

A seguir veja um exemplo de uso desse recurso através de uma função, que cria um vetor de dez posições e os preenche com valores aleatórios, imprime os valores, e posteriormente passa esse vetor para “quem” chamar a função:

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

int *gerarRandomico();

int main() {
   setlocale(LC_ALL, "portuguese");

   int *p;
   int i;

   p = gerarRandomico();

   for(i = 0; i < 10; i++) {
      printf("p[%d] = %d\n", i, p[i]);
   }

   return 0;
}

int *gerarRandomico() {
   static int r[10];
   int a;

   for(a = 0; a < 10; a++) {
      r[a] = rand() % 100;

      printf("r[%d] = %d\n", a, r[a]);
   }

   return r;
}
```

Nessa aula vimos como criar funções que após um determinado conjunto de instruções retorna um valor para “quem” chamou a sub-rotina. Esse conhecimento permitirá criar programas mais organizados e também evitar repetição de códigos.

Escopo e Passagem de Parâmetros

Nessa aula vamos ver sobre escopo de variáveis, bem como a passagem de parâmetros em uma função.

Escopo de Variáveis

As variáveis são usadas para armazenar dados temporariamente na memória, porém o local onde esse recurso é definido no código de um programa determina seu escopo e sua visibilidade.

Veja a seguir um exemplo de variáveis em funções:

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

int testar();

int main() {
   setlocale(LC_ALL, "portuguese");

   int x = 20;

   printf("Valor de x na função main(): %d\n", x);
   printf("Valor de x na função testar(): %d\n", testar());

   return 0;
}

int testar() {
   int x = 10;

   return x;
}
```

No exemplo, há duas variáveis chamadas “x”, mas isso não acarreta erro, pois mesmo as variáveis tendo o mesmo nome elas são definidas em lugares diferentes, uma está dentro da função *main()* e outra dentro da *testar()* e cada função terá seu espaço na memória de forma independente. Na memória, as variáveis são localizadas pelo seu endereço, portanto mesmo sendo declaradas com o mesmo nome, seus endereços são distintos.

O escopo é dividido em duas categorias: Local ou global:

- **Variáveis Locais:** Elas existem e são “enxergadas” somente dentro do corpo da função onde foram definidas.
- **Variáveis Globais:** Elas existem e são “enxergadas” por todas as funções do programa.

No exemplo anterior, ambas as variáveis são locais.

No exemplo a seguir, veja a declaração e uso da variável global.

Sua inclusão é feita após a biblioteca de entrada e saída padrão. Na função principal não foi definida nenhuma variável com o nome de “x” e mesmo assim pode ser impresso seu valor na linha 10, pois é acessado o valor da variável global. Já na linha 12 é impresso o valor da variável global modificado pela função *testar()*, que retorna o dobro do valor:

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

int x = 10;

void testar();

int main() {
   setlocale(LC_ALL, "portuguese");

   printf("Valor de x global: %d\n", x);

   testar();

   printf("Valor de x global alterado em testar(): %d\n", x);

   return 0;
}

void testar() {
   x *= 2;
}
```

Vamos ver um exemplo de utilização do escopo global de uma variável, um programa que calcula a média entre duas temperaturas distintas. Na linha 2 foram declaradas duas variáveis. O programa inicia a execução na linha 8. Na 9 é solicitado ao usuário digitar duas temperaturas, as quais são armazenadas dentro das variáveis globais criadas. Na linha 11 a função *calcularMedia()* é invocada para fazer o cálculo da média usando os valores das variáveis globais:

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

float t1, t2;

float calcularMedia();

int main() {
   setlocale(LC_ALL, "portuguese");

   printf("Digite as duas temperaturas: ");
   scanf("%f%f", &t1, &t2);

   printf("\nA temperatura média é de %.1f°C\n", calcularMedia());

   return 0;
}

float calcularMedia() {
   return (t1 + t2) / 2;
}
```

Variável Global e Local com Mesmo Nome

Na linguagem C, para conseguirmos acessar o valor de uma variável global, dentro de uma função que possui uma variável local com mesmo nome, devemos usar a instrução *extern* (Manzano, 2015). Veja no exemplo, que foi necessário criar uma nova variável chamada “b”, com um bloco de instruções, que atribui a nova variável o valor “externo” de x:

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

int x = 10;

int main() {
   setlocale(LC_ALL, "portuguese");

   int x = -1;
   int b;

   {
      extern int x;
      b = x;
   }

   printf("Valor de x: %d\n", x);
   printf("Valor de b (x global): %d\n", b);

   return 0;
}
```

Passagem de Parâmetros para Funções

Ao definir uma função, pode-se também estabelecer que ela receberá informações “de quem” a invocou, ou seja, “quem chamar” a função deve informar os valores, sobre os quais o cálculo será efetuado. Ao criar uma função que recebe parâmetros é preciso especificar qual o tipo de valor que será recebido. Uma função pode receber parâmetros na forma de valor ou de referência.

Passagem por Valor

Na passagem parâmetros por valores, a função cria variáveis locais automaticamente para armazenar esses valores e após a execução da função essas variáveis são liberadas. Veja a seguir, um exemplo de definição e chamada de função com passagem de valores:

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

void testar(int n1, int n2);

int main() {
```

```
    setlocale(LC_ALL, "portuguese");

    int n1 = 10;
    int n2 = 20;

    printf("Valores antes de chamar a função testar(): %d e %d\n", n1, n2);

    testar(n1, n2);

    printf("Valores depois de chamar a função testar(): %d e %d\n", n1, n2);

    return 0;
}

void testar(int n1, int n2) {
    n1 = -1;
    n2 = -2;

    printf("Valores dentro da função testar(): %d e %d\n", n1, n2);
}
```

Ao utilizar variáveis como argumentos, na passagem de parâmetros por valores, essas variáveis não são alteradas, pois é fornecido uma cópia dos valores armazenados para a função (Soffner, 2013).

Passagem por Referência

A utilização de funções com passagem de parâmetros por referência está diretamente ligada aos conceitos de ponteiro e endereço de memória. A ideia da técnica é análoga a passagem por valores, ou seja, a função será definida de modo a receber certos parâmetros de “quem” faz a chamada do método que deve informar esses argumentos. Entretanto, o comportamento e o resultado são diferentes.

Na passagem por referência não será criada uma cópia dos argumentos passados, na verdade, será passado o endereço da variável e função irá trabalhar diretamente com os valores ali armazenados (Soffner, 2013).

Como a função utiliza o endereço (ponteiros), na sintaxe serão usados os operadores * e & (Manzano, 2015).

Na definição da função, os parâmetros a serem recebidos devem ser declarado com *.

Na chamada da função os parâmetros devem ser passados com o &.

Veja um exemplo:

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

void testar(int *n1, int *n2);

int main() {
```

```
    setlocale(LC_ALL, "portuguese");

    int n1 = 10;
    int n2 = 20;

    printf("Valores antes de chamar a função testar(): %d e %d\n", n1, n2);

    testar(&n1, &n2);

    printf("Valores depois de chamar a função testar(): %d e %d\n", n1, n2);

    return 0;
}

void testar(int *n1, int *n2) {
    *n1 = -1;
    *n2 = -2;

    printf("Valores dentro da função testar(): %d e %d\n", *n1, *n2);
}
```

Passagem de Vetor

Esse recurso pode ser utilizado para criar funções que preenchem e imprimem o conteúdo armazenado em um vetor, evitando a repetição de trechos de código. Na linguagem C, a passagem de um vetor é feita implicitamente por referência. Isso significa que mesmo não utilizando os operadores “*” e “&”, quando uma função que recebe um vetor é invocada, o que é realmente passado é o endereço da primeira posição do vetor.

Na definição da função, os parâmetros a serem recebidos devem ser declarados com colchetes sem especificar o tamanho.

Na chamada da função os parâmetros devem ser passados como se fossem variáveis simples.

No exemplo a seguir o programa por meio de uma função preenche um vetor de três posições e em outra função percorre o vetor imprimindo o dobro de cada valor do vetor:

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

void inserir(int a[]);
void imprimir(int b[]);

int main() {
    setlocale(LC_ALL, "portuguese");

    int numeros[3];

    printf("Preenchendo o vetor...\n\n");

    inserir(numeros);
```

```
    printf("\nDobro dos valores informados:\n\n");

    imprimir(numeros);

    return 0;
}

void inserir(int a[]) {
    int i = 0;

    for(i = 0; i < 3; i++) {
        printf("Digite o valor %d: ", i);
        scanf("%d", &a[i]);
    }
}

void imprimir(int b[]) {
    int i = 0;

    for(i = 0; i < 3; i++) {
        printf("Número [%d]: %d\n", i, 2 * b[i]);
    }
}
```

Nessa aula vimos sobre escopo de variáveis, bem como a passagem de parâmetros em uma função, como passagem de parâmetros por valor ou por referência. As técnicas apresentadas, fazem parte do cotidiano de um desenvolvedor em qualquer linguagem de programação.

Recursividade

Vimos como criar uma função, qual sua importância dentro de uma implementação, estudamos a saída de dados de uma função, bem como a entrada, feita por meio dos parâmetros. E nesta aula, vamos ver uma categoria especial de função, chamada de funções recursivas.

Recursividade

A palavra recursividade está associada a ideia de recorrência de uma determinada situação. Em programação, as funções recursivas, quando uma função chama a si própria.

A sintaxe para implementação de uma função recursiva, nada difere das funções gerais, a diferença está no corpo da função, pois a função será invocada dentro dela mesma.

Veja a seguir alguns pontos de atenção da função recursiva:

- **Ponto de Parada:** A função recursiva chama a si própria até que um ponto de parada seja estabelecido. O ponto de parada poderá ser alcançado através de uma estrutura condicional ou através de um valor informado pelo usuário.
- **Instâncias:** Uma função possui em seu corpo variáveis e comandos, os quais são alocados na memória de trabalho. No uso de uma função recursiva, os recursos (variáveis e comandos) são alocados (instâncias) em outro local da memória, ou seja, para cada chamada

da função novos espaços são destinados a execução do programa. E é justamente por esse ponto que o ponto de parada é crucial.

- **Variáveis:** As variáveis criadas em cada instância da função na memória são independentes, ou seja, mesmo as variáveis tendo nomes iguais, cada uma tem seu próprio endereço de memória e a alteração do valor em uma não afetará na outra.
- **Caso Base:** Toda função recursiva, obrigatoriamente, tem que ter uma instância com um caso que interromperá a chamada a novas instâncias. Essa instância é chamada de caso base, pois representa o caso mais simples, que resultará na interrupção.

Mecanismo da Função Recursiva

A instância 1 representa a primeira chamada à função, que por sua vez, possui em seu corpo um comando que chama a si mesma, nesse momento é criada a segunda instância dessa função na memória de trabalho. Um novo espaço é alocado, com variáveis e comandos, como a função é recursiva, novamente ela chama a si mesma, criando então a terceira instância da função.

Dentro da terceira instância, uma determinada condição de parada é satisfeita, nesse caso, a função deixa de ser instanciada e passa a retornar valores. A partir do momento que a função recursiva alcança o ponto de parada, cada instância da função passa a retornar seus resultados para a instância anterior (a que fez a chamada). No caso, a instância três retornará para a dois, e a dois, retornará para a um.

Veja no exemplo uma função recursiva que faz a somatória dos antecessores de um número, inteiro positivo, informado pelo usuário, ou seja, se o usuário digitar 5, o programa deverá retornar o resultado da soma 5 + 4 + 3 + 2 + 1 + 0. A função deverá somar até o valor zero, portanto esse será o critério de parada.

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

int somar(int valor);

int main() {
   setlocale(LC_ALL, "portuguese");

   int num, resultado;

   printf("Digite um número inteiro positivo: ");
   scanf("%d", &num);

   resultado = somar(num); // Primeira chamada da função

   printf("\nResultado: %d.\n", resultado);

   return 0;
}

int somar(int valor) {
   if(valor == 0) {
      return valor;
   }
```

```
    else {
        return valor + somar(valor - 1);
    }
}
```

Técnica de Recursividade ou Estruturas de Repetição?

A técnica de recursividade pode substituir o uso de estruturas de repetição, tornando o código mais elegante, do ponto de vista das boas práticas de programação. Entretanto, as funções recursivas podem consumir mais memória que as estruturas iterativas.

Portanto, a recursividade é indicar quando um problema maior, pode ser dividido em instâncias menores do mesmo problema, porém considerando a utilização dos recursos computacionais que cada método empregará.

Veja um exemplo clássico de cálculo do fatorial. O fatorial de um número qualquer, consistem em multiplicações sucessivas até que seja igual ao valor unitário, ou seja, 5! = 5 x 4 x 3 x 2 x 1, que resulta em 120.

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

int fatorial(int valor);

int main() {
    setlocale(LC_ALL, "portuguese");

    int num, resultado;

    printf("Digite um número inteiro positivo: ");
    scanf("%d", &num);

    resultado = fatorial(num);

    printf("\nResultado: %d.\n", resultado);

    return 0;
}

int fatorial(int valor) {
    if(valor != 1) {
        return valor * fatorial(valor - 1);
    }
    else {
        return valor;
    }
}
```

Recursividade em Cauda

O mecanismo da função recursiva é custoso para o computador, pois tem que alocar recursos para as variáveis e comandos da função, procedimento chamado de empilhamento, e tem também que armazenar o local onde foi feita a chamada da função (Oliveira, 2018). Para usar de forma mais otimizada a memória, existe uma alternativa chamada recursividade em cauda.

Nesse tipo de técnica a recursividade funcionará como uma função iterativa (Oliveira, 2018). Uma função é caracterizada como recursiva em cauda quando a chamada a si mesmo é a última operação a ser feita no corpo da função. Nesse tipo de função, o caso base costuma ser passado como parâmetro, o que resultará em um comportamento diferente.

Veja no exemplo como implementar o cálculo do fatorial usando essa técnica. Na linha 3, a função recursiva em cauda retorna o fatorial, sem nenhuma outra operação matemática e passa o número a ser calculado e o critério de parada junto. Foi preciso criar uma função auxiliar para efetuar o cálculo.

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

int fatorialCauda(int valor);
int fatorialAux(int valor, int parcial);

int main() {
   setlocale(LC_ALL, "portuguese");

   int num, resultado;

   printf("Digite um número inteiro positivo: ");
   scanf("%d", &num);

   resultado = fatorialCauda(num);

   printf("\nResultado: %d.\n", resultado);

   return 0;
}

int fatorialCauda(int valor) {
   return fatorialAux(valor, 1);
}

int fatorialAux(int valor, int parcial) {
   if(valor != 1) {
      return fatorialAux(valor - 1, parcial * valor);
   }
   else {
      return parcial;
   }
}
```

Nesta aula vimos sobre o assunto de funções recursivas. É importante que o desenvolvedor tenha um bom repertório de técnicas para solucionar os mais diversos problemas, sendo a recursividade uma delas.

Listas

Estruturas de Dados

As estruturas de dados são formas de organização e distribuição de dados para tornar mais eficientes a busca e manipulação dos dados por algoritmos. As listas são estruturas de dados dinâmicas, uma vez que o número de elementos de uma lista é variável conforme eles são inseridos ou removidos.

Para a implementação de listas, os vetores são muito utilizados para representar um conjunto de dados, permitindo definir um tamanho máximo de elementos a ser utilizado neste vetor.

O uso do vetor, ao ser declarado, reserva um espaço contíguo na memória para armazenar seus elementos. Assim, é possível acessar qualquer um dos seus elementos a partir do primeiro elemento, por meio de um ponteiro (Celes, Cerqueira e Rangel, 2004).

Listas Ligadas

Uma lista ligada, também conhecida como lista encadeada, é um conjunto de dados dispostos por uma sequência de nós, onde a relação de sucessão desses elementos é determinada por um ponteiro que indica a posição do próximo elemento, podendo estar ordenado ou não (Silva, 2007).

Uma lista é uma coleção.

Sua propriedade estrutural baseia-se apenas na posição relativa dos elementos, que são dispostos linearmente.

O nó de lista é composto por duas informações, uma informação armazenada e um ponteiro que indica o próximo elemento da lista.

Diferentes dos vetores, onde o armazenamento é realizado de forma contígua, a lista ligada estabelece a sequência de forma lógica.

Na lista encadeada é definido um ponto inicial ou um ponteiro para o começo da lista, e a partir daí pode se inserir elementos, remover ou realizar buscas na lista.

Quando uma lista está sem nós, definimos como lista vazia ou lista nula, assim, o valor do ponteiro para o próximo nó da lista, é considerado como ponteiro nulo:

- Criação ou definição da estrutura de uma lista.
- Inicialização da lista.
- Inserção com base em um endereço como referência.
- Alocação de um endereço de nó para inserção na lista.
- Remoção do nó com base em um endereço como referência.
- Deslocamento do nó removido da lista.

Elementos de Dados em Listas Ligadas

Os elementos de uma lista são armazenados em posições sequenciais de memória, sempre que possível e de forma dinâmica, e permitem que a lista seja percorrida em qualquer direção.

Os elementos de informação de uma lista podem ser do tipo int, char e/ou float.

Ao criar uma estrutura de uma lista, definimos também o tipo de dados que será utilizado em sua implementação.

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

typedef struct lista lst;
```

Exemplo de declaração para criar uma lista em C:

```
struct alunos {
   char nome[25];
   struct alunos *prox;
};

typedef struct alunos classe;
```

- Criando uma *struct* (registro) alunos.
- Na *struct*, há uma variável nome do tipo char, que será nossa informação.
- Outra *struct prox* com ponteiro para a própria *struct* alunos para receber o endereço de apontamento da próxima informação.

Inicialização da Lista

Precisamos inicializar a lista para utilizarmos após sua criação.

Para isso, basta criarmos uma função onde inicializamos a lista como nula, como uma das possíveis formas de inicialização:

```
lista *inicializa(void) {
   return NULL;
}
```

Ponteiros – Elementos de Ligação

A utilização de ponteiros é indicada nos casos onde é preciso conhecer o endereço que está armazenada a variável:

```
// Ponteiros do tipo int, float e char, respectivamente:
int *iptr;
float *fptr;
char *cptr;
```

A seguir, veja um exemplo de uma estrutura de lista declarada para armazenar dados de uma agenda:

```
typedef struct lista {
   char *nome;
   int telefone;
   struct lista *proximo;
} dados;
```

Para sabermos o endereço da memória reservada para a variável, utiliza-se o operador & juntamente ao nome de uma variável.

```
int x = 10;
int *p;
p = &x;
```

Função malloc()

Em listas, além do uso de ponteiros, utilizamos também as alocações dinâmicas de memória, que são porções de memórias para utilização das listas. Para isso são utilizadas a função *malloc()*, memory allocation ou alocação de memória. Ela é a responsável pela reserva de espaços na memória principal. Sua finalidade é alocar uma faixa de bytes consecutivos na memória do computador e retornar seu endereço ao sistema.

Exemplo de utilização da função *malloc()* e de ponteiro:

```
char *pnt;
pnt = malloc(2);

scanf("%c", pnt);
```

O endereço retornado é o da posição inicial onde se localiza os bytes alocados.

Em uma lista, precisamos alocar o tipo de dado no qual foram declarados os elementos da lista, e por este tipo de dados ocupar vários bytes na memória, precisaremos utilizar a função *sizeof()*, que permite nos informar quantos bytes o tipo de elemento criado terá.

Exemplo de programa em C com *malloc()* e *sizeof()*:

```
int main() {
   int *p;
   p = (int*)malloc(sizeof(int));

   if(!p) {
      printf("Erro de memória insuficiente!");
   }
   else {
      printf("Memória alocada com sucesso!");
   }

   return 0;
}
```

Podemos classificar as listas de duas formas:

- **Lista com Cabeçalho:** Onde o primeiro elemento permanece sempre como ponto inicial na memória, independente se a lista está com valores ou não. Assim, o start é o endereço inicial da nossa lista e para determinar que a lista está vazia.
- **Lista sem Cabeçalho:** Onde o conteúdo do primeiro elemento tem a mesma importância que os demais elementos. Assim, a lista é considerada vazia se o primeiro elemento é *NULL*. A criação deste tipo de lista vazia pode ser definida por *start = NULL*.

Exemplo de um trecho de uma função do tipo inteiro, para inserir pacientes com propriedades na lista:

```
int insereComPrioridade(lista *f, paciente *p) {
   lista *ptnodo, *aux, *ant;
   ptnodo = (lista*)malloc(sizeof(lista));
}
```

Aplicação de Listas Ligadas no Nosso Dia a Dia

Lista de Compra de Supermercado

Ao anotar tudo que você precisa comprar, você automaticamente está criando uma lista, e esta lista pode ser alocada com novos produtos ou remover produtos dela conforme a compra está sendo realizada:

```
dados *iniciaListaMerc(char *prod, int numprod) {
   dados *novo;

   novo = (dados*)malloc(sizeof(dados));

   novo->prod = (char*)malloc(strlen(prod) + 1);
   strncpy(novo->prod, prod, strlen(prod) + 1);
   novo->numprod = numprod;
   novo->proximo = NULL;

   return novo;
}
```

Desenvolvimento de Um Sistema Para o Site de Uma Empresa de Casamentos

Os usuários criam a lista de convidados e a lista de presentes, os noivos podem adicionar convidados ou presentes e remover ou alterar da lista quando necessário:

```
void inserirConvid(tipoitem elemento, int &cont) {
   festa *novo, *aux, *aux1;
   aux = inicio;
   aux1 = inicio->prox;

   while(aux1 != NULL) {
      if(strcmp(aux1->item.nome, elemento.nome) > 0)
         break;
```

```
    aux = aux->prox;
    aux1 = aux1->prox;
  }

  if((novo = new(festa)) == NULL)
    printf("Memória insuficiente!");
  else {
    novo->prox = aux->prox;
    aux->prox = novo;
    novo->item = elemento;
    cont++;
    printf("Convidado inserido com sucesso!");
  }

  if(aux1 == NULL)
    fim = novo;

  return;
}
```

Pilhas

Uma pilha tem como definição básica ser um conjunto de elementos ordenados que permite a inserção e a remoção de mais elementos em apenas uma das extremidades da estrutura denominada topo da pilha (Tenenbaum, Langsam e Augenstein, 2007). Assim, um novo elemento que é inserido passa a ser o topo da pilha, e o único elemento que pode ser removida da pilha é o elemento que está no topo.

Os elementos inseridos em uma pilha apresenta, uma sequência de inserção. O primeiro elemento que entra na pilha só pode ser removido por último, após todos os outros elementos serem removidos.

Segundo Celes, Cerqueira e Rangel (2004), os elementos da pilha só pode ser retirados na ordem inversa da ordem em que nela foram inseridos. Isso é conhecido como LIFO (Last In, First Out, ou seja, o último que entra é o primeiro ao sair) ou FILO (First In, Last Out, ou seja, primeiro que entra é o último a sair).

Operações de Pilhas

Uma pilha também pode estar no estado de pilha vazia quando não houver elementos. Segundo Celes, Cerqueira e Rangel (2004), os passos para a criação de uma pilha são:

- Criar uma pilha vazia.
- Inserir um elemento no topo.
- Remover o elemento do topo.
- Verificar se a pilha está vazia.
- Liberar a estrutura de pilha.

Empilhar e Desempilhar

A operação de empilhar um novo elemento, segundo Lorenzi, Mattos, Carvalho (2015), tem a função de inserir um novo elemento na pilha, e definida na programação em C++ como *push()*. Já a operação de desempilhar, tem a função de remover um elemento do topo da pilha e utilizada na programação em C++ como *pop()*.

Conforme Drozdek (2016), outras operações que podem ser implementadas na pilha são as funções *clear()*, para limpar a pilha e *isEmpty()*, para verificar se a pilha está vazia.

Conforme Tenembaum (2007), uma pilha possui uma estrutura que pode ser declarada contendo dois objetos:

- Um ponteiro, que irá armazenar o endereçamento inicial da pilha.
- Um valor inteiro, que irá indicar a posição no topo da pilha.

A seguir, veja as implementações da pilha:

```
// Declaração da estrutura inicial:
struct pilha {
   int topo;
   int capacidade;
   float *proxElem;
};

struct pilha minhaPilha;

// Criar a pilha:
void criaPilha(struct pilha *p, int c) {
   p->proxElem = (float*)malloc(c * sizeof(float));
   p->topo = -1;
   p->capacidade = c;
}

// Inserir elemento na pilha:
void pushPilha(struct pilha *p, float v) {
   p->topo++;
   p->proxElem[p->topo] = v;
}

// Remover elemento da pilha:
float popPilha(struct pilha *p) {
   float aux = p->proxElem[p->topo];
   p->topo--;

   return aux;
}

// Informar se a pilha está vazia:
int pilhaVazia(struct pilha *p) {
   if(p->topo == -1)
      return 1;
   else
      return 0;
```

```
}

// Verificar se a pilha está cheia:
int pilhaCheia(struct pilha *p) {
   if(p->topo == p->capacidade - 1)
      return 1;
   else
      return 0;
}
```

Problema do Labirinto

Os labirintos são desafios criados como problematização de estrutura de dados.

Em um labirinto, por exemplo, para encontrar um caminho correto, podemos andar pelo labirinto até que se encontre uma divisão neste caminho. Caso o caminho escolhido não possua uma saída, é removido o ponto anterior da pilha, voltando ao último ponto em que o labirinto se dividiu e recomeçamos por um outro caminho ainda não escolhido, adicionando na pilha o novo caminho.

A seguir, um trecho de implementação de criação de uma solução para labirinto:

```
void inicLabirinto(labirinto *l, pilha *pl, int linha, int coluna) {
   int i, j, flag = 0;
   char aux;
   elemTPilha origem;

   for(i = 0; i < linha; i++) {
      for(j = 0; j < coluna; j++) {
         if(l->p[i][j].tipo == '0') {
            l->p[i][j].visitado = 1;
            origem.x = i;
            origem.y = j;

            pushPilha(pl, origem);
         }
      }
   }
}
```

Backtracking

As pilhas podem ser aplicadas também no uso de algoritmos de backtracking, que consiste em criar marcações para onde o algoritmo pode retornar. Podem ser aplicados também como operação de desfazer, existem em diversas aplicações de usuários, como a utilização deste algorimo em sistema de GPS. Quando o motorista utiliza uma rota não indicada pelo programa o algoritmo de backtracking é aplicado para definir a rota.

O algoritmo de backtracking tem como meta resolver o problema no menor intervalo de tempo possível, sem levar em consideração o esforço de operações para alcançar a solução do problema.

Fila

Uma fila é a representação de um conjunto de elementos. Podemos remover esses lementos deste conjunto por uma extremidade chamada de início da fila, e pela outra extremidade, na qual são inseridos os elementos, chamada de final da fila (Tenenbaum, Langsam e Augenstein, 2007).

Assim como uma pilha, as filas também são estruturas dinâmicas com tamanho variável, podendo aumentar ou diminuir conforme são inseridos ou removidos elementos da fila.

Em uma fila, os elementos entrar por uma extremidade e são removidos pela outra extremidade. Isso é conhecido como FIFO (First In, First Out, ou seja, o primeiro que entra é o primeiro a sair). No caso desta fila, sabemos quais os elementos com base em seu número de índice. Assim, a fila apresenta sua ordem de entrada (fim da fila) e sua ordem de saída dos elementos (início da fila).

Operações de Filas

Segundo Drozdek (2016), a estrutura de dados de fila possui operações similares às da estrutura de pilha para gerenciamento de uma fila, como:

- Criar uma fila vazia.
- Inserir um elemento no fim da fila.
- Remover o elemento do início da fila.
- Verificar se a fila está vazia.
- Liberar a fila.

Conforme Celes, Cerqueira e Rangel (2004), podemos simplesmente utilizar um vetor para armazenar os elementos e implementamos uma fila nessa estrutura de dados, ou podemos utilizar uma alocação dinâmica de memória para armazenar esses elementos.

A seguir, veja as implementações de fila:

```
// Definição de constante:
#define N 10

// Declaração da estrutura inicial:
struct fila {
    int n;
    int ini;
    char vet[N];
};

typedef struct fila fl;

// Inicializar a fila:
minhaFila *iniciaFila(void) {
    fl *f = (fl*)malloc(sizeof(fl));

    f->n = 0;
    f->ini = 0;

    return f;
}

// Inserir elemento na fila:
```

```
void insereFila(fl *f, char elem) {
   int fim;

   if(f->n == N) {
      printf("A fila está cheia!");
      exit(1);
   }

   fim = (f->ini + f->n) % N;

   f->vet[fim] = elem;
   f->n++;
}

// Remover elemento na fila:
float removeFila(fl *f) {
   char elem;

   if(filaVazia(f)) {
      printf("A fila está vazia!");
      exit(1);
   }

   elem = f->vet[f->ini];
   f->ini = (f->ini + 1) % N;
   f->n--;

   return elem;
}

// Informar se a fila está vazia:
int filaVazia(fl *f) {
   return (f->n == 0);
}

// Liberar aloação de memória:
void liberaFila(fl *f) {
   free(f);
}
```

Filas Circulares

Segundo Silva (2007), as filas não apresentam uma solução completa, sendo que, mesmo chegando ao final do vetor poderemos ter a fila cheia mesmo não estando cheia, uma vez que elementos podem ter sido removidos e para isso, podemos utilizar as filas circulares como solução para esta situação. A fila circular possui a seguinte definição, em sua implementação:

- Um vetor para os elementos.
- Um valor inteiro para o tamanho da fila.
- Um valor inteiro para o início da fila.
- Um valor inteiro para o fim da fila.

Conforme Drozdek (2016), em uma fila circular, o conceito de circularidade se baseia quando o último elemento da fila está na última posição do vetor, e é adjacente à primeira. Assim, são os ponteiros, e não os elementos da fila que se movem em direção ao início do vetor.

Para implementar uma estrutura de fila circular, podemos utilizar como elemento o código a seguir:

```
// Definição de constante:
#define N 10

// Declaração da estrutura inicial:
struct filaCirc {
   int tam, ini, fim;
   char vet[N];
};

typedef struct filaCirc flC;

// Função para inicializar a fila:
void iniciaFila(flC *f) {
   f->tam = 0;
   f->ini = 1;
   f->fim = 0;
}

// Função para inserir a fila:
void insereFila(flC *f, char elem) {
   if(f->tam == N - 1) {
      printf("A fila está cheia!");
   }
   else {
      f->fim = (f->fim % (N - 1)) + 1;
      f->vet[f->fim] = elem;
      f->tam++;
   }
}

// Verificar se a fila está vazia:
int filaVazia(flC *f) {
   return (f->tam == 0);
}

// Remover da fila:
char removeFila(flC *f) {
   if(filaVazia(f)) {
      printf("Fila vazia!");
   }
   else {
      f->ini = (f->ini % (N - 1)) + 1;
      f->tam--;
   }
}
```

Problemas com a Utilização de Fila

Considerando a utilização de uma fila, alguns problemas podem surgir:

- **Utilização de Vetor:** Se utilizarmos um vetor, teremos o problema de possuir um armazenamento de tamanho fixo e limitado, enquanto a fila pode crescer com a necessidade de uso do sistema. Para resolver essa problemática, teríamos que limitar o tamanho máximo da fila ao tamanho do vetor.
- **Fila Cheia ou Fila Vazia:** Em ambos os casos seria impossível realizar as operações. Como solução, é importante sempre implementar as funções para verificar se a fila está cheia e para verificar se ela está vazia.
- **Identificação das Posições da Fila:** Podem surgir problemas relacionados aos controles de início e fim de fila, em que não é possível identificar as posições em que se encontram. Como solução, é preciso incluir duas variáveis (início e fim) para armazenar a posição do início e do fim da fila, e sempre atualizar esses valores conforme a fila aumenta ou diminui.