UniEVANGÉLICA
CENTRO UNIVERSITÁRIO

Algoritmos e Programação

Bacharelados em Computação

Centro Universitário de Anápolis - UniEVANGÉLICA

Título: **Algoritmos e Programação** / Ano – 2018

Preparo de Originais

Aline Dayany de Lemos	Larissa Bitencourt Cardodo
Charley Junior Jabbar	Nathalia Siqueira Cecilio
Isabela Ribeiro Canedo	Paulo Henrique Pereira Almeida
Isaias Batista Franca	Thaís Tavares Amorim
João Vitor Moreira de Sousa	Victor Eugenio Bento Souza

Equipe editorial:
Revisora:
Prof. Esp. Aline Dayany de Lemos
Prof. M.e. Luciana Nishi
Prof. M.e. Jeane Silveira Oliveira
Prof. M.e. Marcio Mariano da Silva
Coordenadora dos Projetos Interdisciplinares: Prof. M.e Adrielle Beze Peixoto
Diretora dos Cursos de Engenharia de Computação e Engenharia de Software: Prof. M.e Viviane Carla Batista Pocivi

Centro Universitário de Anápolis
Chanceler – Ernei de Oliveira Pina
Reitor – Carlos Hassel Mendes da Silva
Pró-Reitora Acadêmica – Cristiane Martins Rodrigues Bernardes
Pró-Reitor de Pós-Graduação, Pesquisa, Extensão e Ação Comunitária – Sandro Dutra e Silva
Coordenador da Pesquisa e Inovação – Bruno Junior Neves
Coordenador de Extensão e Ação Comunitária – Fábio Fernandes Rodrigues

Cursos de Engenharia de Computação e Engenharia de Software:
Diretora - Viviane Carla Batista Pocivi
Núcleo Docente Estruturante - Adrielle Beze Peixoto
Núcleo Docente Estruturante - Natasha Sophie Pereira
Núcleo Docente Estruturante - Renata Dutra Braga
Núcleo Docente Estruturante - Walquíria Fernandes Marins

Sumário

Indicação de Ícones

Saiba mais: oferece novas informações que enriquecem o assunto ou "curiosidades" e notícias recentes relacionadas ao tema estudado.

Glossário: indica a definição de um termo, palavra ou expressão utilizada no texto.

Atenção: indica pontos de maior relevância no texto.

Mídias integradas: sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais e outras.

Palavra da Coordenação

A formação integral é uma preocupação constante na construção didático-pedagógica dos Bacharelados em Computação do Centro Universitário de Anápolis – UniEvangélica. Sua efetividade é garantida por meio da inserção de projetos interdisciplinares, como disciplinas integradoras na matriz curricular do curso, e objetiva que as habilidades e competências exigidas no perfil do egresso sejam fortalecidas por meio de projetos que contribuam para a integração dos conhecimentos teóricos, técnicos, práticos e interpessoais.

As disciplinas de Projeto Interdisciplinar (PI) promovem a associação entre os diferentes conteúdos, habilidades e cenários em projetos que favoreçam a construção do conhecimento científico, aliado à autoaprendizagem, proatividade, resolução conjunta de problemas, trabalho em equipe, reflexividade, entre outros. Para tanto, estas disciplinas têm início no primeiro período do curso e evolui em uma constante de maturidade pessoal, interpessoal e científica.

Esta revista apresenta o resultado dos trabalhos desenvolvidos nestas disciplinas e apresentados durante o SITES – Seminário Interdisciplinar de Tecnologia e Sociedade. Entendemos que este é um passo importante para o processo que vem sendo construído e intentamos que assim como o desenvolvimento deste material foi enriquecedor aos discentes do curso, o acesso aos mesmos, traga resultados positivos a aqueles que tiverem acesso.

Apresentação

Este material foi produzido por alunos e com a supervisão de professores do curso de Engenharia de Computação da UniEvangélica de Anápolis – Goiás. Este documento tem como intenção atender às competências e habilidades elencadas no Plano de Pedagógico do Curso e reiterado no Plano de Ensino da disciplina de Projeto Interdisciplinar IV.

A construção de um perfil profissional, interpessoal e técnico é um desafio que construímos juntos e ao longo do semestre. Despertar no aluno a capacidade de organizar e estruturar conhecimentos gerais e desafia-lo na tomada de ações além da capacidade de absolvição e reprodução de conteúdo. Proporcionar a construção de conteúdos técnicos para auxiliar colegas e a comunidade em geral, possibilitando a identificação de potencialidades e fraquezas sobre a sua comunicação verbal e escritas.

Em contraponto delíamos ao aluno do minicurso a responsabilidade quanto à sua responsabilidade de aprendizado. Em ambientes EAD, ou a responsabilidade do aluno é significativamente maior em relação à educação presencial. Esta modalidade pode ser um pouco nova para você. Estamos “conectados” em um mesmo local, porém não o estamos fisicamente. O aluno, a partir desta perspectiva, tem total responsabilidade sobre o seu aprendizado, então leia, releia e leia novamente se necessário, mas não deixe que o tempo passe e você não aprenda o que se propôs.

Sobre o Conteúdo

Neste documento vamos aprender um pouco sobre lógica de programação, e que a utilizamos em vários momentos do dia a dia. Veremos ainda a parte de construção de um algoritmo em portugol, suas restrições. Despois aplicaremos a lógica e conhecimentos de algoritmos na construção de programas de computador, inicialmente simples e gradativamente aumentando o grau de complexidade.

É comum ouvir de nossos professores, que se você consegue entender bem a estrutura de dados será possível utilizar esse conhecimento em várias outras linguagens de programação. Então mãos à obra...

Lógica

A lógica pode ser atribuída a mente de Aristóteles, que sistematizou e codificou o assunto de modo que sua teoria não foi ultrapassada por mais de dois milênios. A lógica moderna deve-se, em grande parte, ao alemão Gottlob Frege no século XIX. Sabe-se que a lógica teve sua maior desenvoltura na Filosofia, caminhado pela Linguística, Matemática e Computação.

Segundo o dicionário Aurélio, lógica significa “coerência de raciocínio, de ideias. Modo de raciocinar peculiar alguém, ou um grupo. Sequência coerente, regular e necessária de acontecimentos, de coisas”. Um outro conceito poderia ser: A ciência das leis, ideias do pensamento e a arte de aplica-los corretamente no processo de investigação e demonstração da verdade.

Embora não percebamos, em nosso dia-a-dia utilizamos a lógica de programação e o algoritmo. Para executarmos tarefas simples ou até mesmo resolvermos um problema, muitas vezes de maneira inconsciente executamos algoritmos. Da mesma forma que para fazer um bolo é necessária uma receita, para executarmos tarefas mesmo simples precisamos de uma “receita” ou planejamento. Nos deparamos com vários problemas nos quais usamos “lógica” de forma “consciente ou inconsciente” para resolvê-los, ou seja, um raciocínio detalhista, minucioso e com bastante clareza.

Podemos ainda raciocinar de forma lógica sem tomarmos conhecimento, intuitivamente. Vamos ver uma situação:

Você está viajando e fura um pneu de seu carro. Encosta-o e Pará. Será que você é capaz de descrever todos os passos desde a parada do carro até o pneu trocado?

Se você tentou, agora responda:

Você desligou o carro?

Ligou pisca alerta?

Puxou freio de mão?

Tirou sinto de segurança?

Abriu a Porta?

Levou a chave para abrir a porta malas?

Colocou o Triangulo de aviso?

Verificou se o estepe está cheio?

Existem diversos detalhes que muitas vezes fazemos intuitivamente e não nos preocupamos com isso, portanto, ao descrever cada um desses aspectos chegamos a esquecer muitos deles. A lógica seria então a sequência detalhada e clara do fato. Outro exemplo claro da lógica é:

Quando alguém pergunta qual é a soma de 20 + 30, o resultado multiplicado por 4 e este resultado dividido por 2, se você fizer o cálculo de cabeça, geralmente você irá seguir um raciocínio logico, como:

Obter o resultado da soma (20+30=50) que vamos chamar de Resultado

Vamos pegar o resultado 1 que é 50 e multiplicar por 4 (50*4=200) sendo assim, chamaremos este de resultado

Pegamos este resultado e dividimos por 2 (200/2=100) que chamaremos de resultado

Responder a pessoa que perguntou com o resultado que neste caso será 100.

Agora pensando um pouco, provavelmente você nunca tenha imaginado realizar uma tarefa tão simples utilizando tanta lógica, seguindo tantos passos. Nosso propósito é levar a você desenvolver a lógica que já existe em você, levando assim a utilizar um raciocínio lógico.

Vamos trabalhar?

Existe um rio a ser atravessado por três pessoas que pesam 50, 50 e 100 Kg. Para atravessar este rio, as três pessoas têm uma canoa que leva no máximo 100 Kg por viagem. Esta canoa tem que ser conduzida, isto é, deve ser levada manualmente por meio do remo. Eis a questão, como estas pessoas chegam no outro lado da margem? É um problema com resolução simples.

Pense e responda...

Para trabalhar um pouco sua lógica acesse o endereço www.rachacuca.com.br e escolha algum teste do seu interesse, indicamos o Teste de Einstein https://rachacuca.com.br/teste-de-einstein/

Regras básicas para resolver o teste

- Há 5 casas de diferentes cores;
  - Amarela
  - Azul
  - Branca
  - Verde
  - Vermelha
- Em cada casa mora uma pessoa de uma diferente nacionalidade;
  - Alemão
  - Dinamarquês

    - Inglês
    - Norueguês
    - Sueco
- Esses 5 proprietários bebem diferentes bebidas;
    - Água
    - Café
    - Cerveja
    - Chá
    - Leite
- Esses 5 proprietários fumam diferentes tipos de cigarros;
    - Blends
    - Bluemaster
    - Dunhill
    - Pall Mall
    - Prince
- Esses 5 proprietários têm um certo animal de estimação;
    - Cachorro
    - Cavalo
    - Gato
    - Pássaro
    - Peixe
- Nenhum deles têm o mesmo animal, fumam o mesmo cigarro ou bebem a mesma bebida

	1ª Casa	2ª Casa	3ª Casa	4ª Casa	5ª Casa
Cor					
Nacionalidade					
Bebida					
Cigarro					
Animal					

Dicas:
O Norueguês vive na primeira casa.
O Inglês vive na casa Vermelha.
O Sueco tem Cachorros como animais de estimação.
O Dinamarquês bebe Chá.
A casa Verde fica do lado esquerdo da casa Branca.
O homem que vive na casa Verde bebe Café.
O homem que fuma Pall Mall cria Pássaros.
O homem que vive na casa Amarela fuma Dunhill.
O homem que vive na casa do meio bebe Leite.
O homem que fuma Blends vive ao lado do que tem Gatos.
O homem que cria Cavalos vive ao lado do que fuma Dunhill.
O homem que fuma BlueMaster bebe Cerveja.
O Alemão fuma Prince.

O Norueguês vive ao lado da casa Azul.
O homem que fuma Blends é vizinho do que bebe Água.

Algoritmo

Algoritmo é um conjunto de passos aperfeiçoados definidos que visam a solução de um determinado problema. A descrição das diversas operações a serem realizadas para resolver um problema e obter resultados preestabelecidos, onde eventos imprevistos não são aceitáveis. Assim como para fazer um bolo é necessária uma receita, definir ingredientes e os métodos para fazê-lo, em algoritmo primeiro é apresentado os "ingredientes" ou o que o problema que o programa precisa solucionar, depois ele informa os métodos para que você execute a tarefa utilizando-se sempre de raciocínio lógico.

Programa é nada mais do que o algoritmo escrito de uma maneira que aceitável ao computador, ou em linguagem de programação. Para que um computador processe os dados e informações, é necessário que um programa informe ou descreva a tarefa que ele deverá realizar. O algoritmo é o esboço, ou seja, a preparação para a escrita do programa, aonde poderemos avaliar todas as possibilidades de erros operacionais.

Vamos ver um exemplo simples de algoritmo para que possamos entender melhor. "Faça um algoritmo que receba do usuário o nome de uma pessoa e o mostre na tela". Vamos analisar um pouco de forma lógica e interpreta-lo, sabendo que vamos montar nossa receita qual será o primeiro passo?

Precisamos fazer com que o programa solicite o nome do usuário;

Depois precisamos fazer com que o programa leia o nome informado;

Por último precisamos fazer com que o programa imprima o nome.

Então como fica?

```
algoritmo
|     declare NOME literal;
|     leia NOME;
|     escreva NOME;
Fim Algoritmo.
```

Informando início do programa.

Nome da variável (Tipo de Dado)

É o comando que informa ao algoritmo que ele deverá receber a informação

É o comando utilizado para que o algoritmo possa mostrar os cálculos, a resposta da resolução do problema. Todo programa deve ter pelo menos uma saída.

A função dos identificadores é armazenar informações. Essas informações são chamadas de conteúdo, eles dependem do que é declarado, o tipo da variável. Tipo de Identificadores:

Numérico: Toda informação numérica que pertença ao conjunto dos números reais (negativa, nula ou positiva)

Literal: Toda informação composta por um conjunto de caracteres alfanuméricos: numéricos (0 a 9), alfabéticos (A até Z, a até z) e especiais (por exemplo: #, @, $, % , !)

Lógico: Toda informação que pode assumir duas situações (verdadeiro ou falso).

Para criar identificadores é necessário seguir algumas regras, como?

Todo identificar deve iniciar com Letra.

Não deve utilizar caracteres especiais (por exemplo: #, @, $, % , !) e espaços brancos.

Números são aceitáveis após o primeiro caractere do identificador.

Use nomes significativos.

Não utilize palavras reservadas do algoritmo.

A declaração dos identificadores consiste em definir os identificadores que serão utilizados pelos algoritmos e também os tipos de dados que estes identificadores podem guardar. Cada identificador só pode ser usado para um tipo de dado.

```
Exemplos:

declare NOME literal;
declare IDADE numérico;
declare CONFORME lógico;
```

Comando de Atribuição: Representado pelo símbolo: ←

Um comando de atribuição permite-nos fornecer um valor a uma variável, em que o tipo de dado deve ser compatível com o tipo da variável.

EXEMPLOS:

```
declare A, B lógico;
declare X, Y, Z numérico;
declare NOME, ENDERECO literal;
B ← verdadeiro;
A ← B;
```

```
Y ← 4;
NOME ← ” Maria Augusta”;
B ← 5 = 3;
ENDEREÇO ← “ Avenida Brasil, N° 256 – Centro”;
Z ← 6,5;
X ← 8 + 13/5;
```

Após vermos essas definições vamos realizar outro exercício para entendermos melhor o que é cada método e como devemos colocá-lo em nosso algoritmo.

Faça um algoritmo que receba do usuário o nome e a idade de duas pessoas e mostre na tela a soma das idades.

```
algoritmo {Começo do Algoritmo}
        {declaração de variáveis}
        declare NOME1, NOME2 literal;
        declare IDADE1, IDADE2, SOMA numérico;
        {entrada de dados}
        leia NOME1, NOME2, IDADE1,IDADE2;
        {Comando de Atribuição}
        SOMA ← IDADE1 + IDADE2;
        {saída de dados}
        escreva SOMA;
fim algoritmo {Término do algoritmo}
```

Estrutura Sequencial

Em um algoritmo aparece em primeiro lugar as declarações seguidas por comandos que, se não houver indicação em contrário, deverão ser executados em sequência linear de cima para baixo e da esquerda para direita, ou seja, na mesma ordem que foram escritas. As ações deverão ser separadas com ponto e vírgula (;), que tem como objetivo de separar uma ação da outra e auxiliar a organização sequencial das ações, após encontrar um ; o programa deverá executar o próximo comando da sequência.

```
algoritmo {Identificação do início do algoritmo}
        {declaração de variáveis vem primeiro}
        {corpo do algoritmo vem depois}
        ação1; {as ações são comandos de}
        ação2; {entrada, saída e atribuições}
        ...
        ação n;
fim algoritmo. {Identificação do fim do algoritmo}
```

Operadores Aritméticos

Operador	Ação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
pot(x,y)	Potenciação
rad(x)	Radiciação
x mod y	Resto da divisão de x por y
x div y	Quociente da divisão de x por y

EXEMPLOS:

```
2 + 3; X + Y; 4 – 3; N – m; 3 * 4; a * b; 10/2;
x1 / x2 ; 9 mod 4 (resulta em 1);
27 mod 5 (resulta em 2);
9 div 4 (resulta em 2);
27 div 5 (resulta em 5);
pot(2,3) significa 2³
rad(9) (significa raiz quadrada de 9)
```

Operações Relacionais

Operador	Ação
=	igual
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
<>	Diferente

Usamos os operadores relacionais para realizar comparações entre 2 valores do mesmo tipo. Os operadores relacionais são comuns para construção de equações. O resultado obtido de uma relação é sempre um valor lógico. Um exemplo, analisando a relação numérica A + B = C, o resultado poderá ser verdadeiro ou falso à medida que o valor da expressão aritmética A + B seja igual ou diferente do conteúdo da variável C.

Estrutura Condicional

Em algumas situações haverá necessidade de escolha. Para que possamos tomar estas decisões foram criadas as estruturas condicionais. Existem duas estruturas condicionais: SE e CASO. Existem ainda estruturas simples e compostas. Vamos ver?

Estrutura Condicional - Se:

Simples

```
se{Condição}
    então{sequência de comandos}
              {caso a condição seja verdadeira}
fim se;
```

Composta:

```
se{Condição}
    então{sequência de comandos}
              {caso a condição seja verdadeira}
```

```
        senão{sequência de comandos}
                    {caso a condição seja falsa}
    fim se;
```

A condição é formada por uma variável + operador relacional + outra variável (ou um valor constante). Operadores relacionais são os relacionados anteriormente.

Exemplos de Condições:

```
se Y < 6
se MEDIA <> 8
se A=B
```

Trabalhando em algoritmos vamos analisar como ficaria um exemplo com estrutura condicional. É nos dado a seguinte questão " Faça um algoritmo que leia 3 notas e fale se o aluno foi aprovado ou não, baseado na média igual ou superior a 7".

```
algoritmo
        declare NOME literal;
        declare  N1, N2, N3, MEDIA numérico;
        leia NOME, N1,N2,N3;
        MEDIA ←(N1+N2+N3) / 3;
         se (MEDIA >= 7)
                então escreva(NOME, "Foi aprovado com média:",
MEDIA);
                senão escreva(NOME, "Foi reprovado com média:",
MEDIA);
         fim se;
fim algoritmo.
```

Observação: É sempre bom observar a pontuação de finalização de cada linha de comando em seu algoritmo, se esquecermos de colocarmos no (;) ponto e vírgula em nosso programa ele poderá não compilar e gerar

erros que muitas vezes não notamos. Parece simples, mas é muito importante observarmos todo o código antes de executa-lo para evitar erros em nosso sistema.

Estrutura Condicional - Caso:

Utilizado para casos de igualdade:

Sintaxe:

```
caso {variável}
     valor 1: {Sequência de Comandos}
     valor 2: {Sequência de Comandos}
     valor n: {Sequência de Comandos}
     senão {Sequência de Comandos}
fim caso;
```

Em algoritmos simples, as estruturas mais comuns e mais usadas são Se e Senão para realizar condições e resolver problemas simples como por exemplo mostrar se um aluno está reprovado ou não.

Estrutura de Repetição

Estrutura de repetição nada mais é que laços de repetição. Laços que permitem que um conjunto de instruções seja repetido até que se faça a condição desejada. Temos:

Para: Quando você tem uma quantidade fixa de repetições a fazer.

Enquanto: Pode ser usada com a mesma finalidade do "para", mas não é tão eficiente quanto o "para" neste caso. Seu outro caso, é quando não temos quantidade fixa de repetições, quando queremos deixar com que o usuário escolher quantas vezes ele vai repetir a condição.

Repita: Usado da mesma forma que o "enquanto", porém ao invés de olhar a condição antes de entrar na repetição, o "repita" executa qualquer forma a primeira vez e só depois vai observar se a condição é verdadeira ou falsa.

Exemplo:

```
para {variável} de {valor inicial} até {valor final} passo {valor} faça
          {sequência de comandos}
fim para;
```

```
enquanto {condição} faça
                  {sequência de comandos}
fim enquanto;


repita
        {sequência de comandos}
até {condição};
```

Recebemos a questão “Faça um algoritmo que leia 3 notas de 15 pessoas e escreva para cada uma se foi aprovado ou reprovado, sabendo que a média é 6”.

Se fosse apenas uma pessoa, como ficaria?

```
algoritmo
        declare  N1, N2, N3, MEDIA numérico;
        leia N1,N2,N3;
        MEDIA ß(N1+N2+N3) / 3;
        se (MEDIA >= 6)
             então escreva “APROVADO”;
             senão escreva “REPROVADO”;
        fim se;
fim algoritmo.
```

Porém, como diz o exercício são 15 pessoas:

```
algoritmo
        declare  N1, N2, N3, MEDIA numérico;
        declare INDICE numérico;
        para INDICE de 1 até 15 passo 1 faça
                 leia N1,N2,N3;
                 MEDIA ß(N1+N2+N3) / 3;
```

```
                se (MEDIA >= 6)
                      então escreva "APROVADO";
                      senão escreva "REPROVADO";
                fim se;
          fim para;
fim algoritmo.
```

E se ao invés de apenas 15 pessoas não tivéssemos um valor delimitado? Como seria? Neste caso, usaremos a estrutura de repetição "enquanto"

```
algoritmo
        declare  N1, N2, N3, MEDIA numérico;
         leia N1
         enquanto N1 <> -1 faça
                     leia N2,N3;
                     MEDIA ß(N1+N2+N3) / 3;
                     se (MEDIA >= 6)
                              então escreva "APROVADO";
                              senão escreva "REPROVADO";
                    fim se;
                    leia N1;
           fim enquanto;
fim algoritmo.
```

Programação

Podemos imaginar um computador como uma grande calculadora, que faz cálculos bem mais rápido que nós humanos, mas para que isso aconteça é necessário que informemos ao computador o que ele deve fazer. A função da linguagem de programação é exatamente essa, ou seja, servir como meio de comunicação entre a máquina e o ser humano.

Linguagem de Programação: Uma linguagem de programação pode ser definida como sendo um conjunto limitado de instruções (vocabulário), associado a um conjunto de regras (sintaxe) que define como as instruções podem ser associadas, ou seja, como se pode compor os programas para resolução de um determinado problema.

Código-Fonte: O texto de um programa que um usuário pode ler, normalmente interpretado como o programa. O código-fonte é a entrada para o compilador.

Código-Objeto: Tradução do código-fonte de um programa em código máquina que o computador pode ler e executar diretamente. O código-objeto é a entrada para o linkeditor.

Linkeditor: Um programa que une funções compiladas separadamente em um programa. Ele compila as funções da biblioteca padrão com o código que você escreveu. A saída do linkeditor é um programa executável.

Biblioteca: O arquivo contento as funções padrão que seu programa pode usar. Essas funções incluem todas as operações de E/S (Entrada e Saída) como também outras rotinas.

Palavras Reservadas: Um aspecto importante em C é que ele tem apenas 32 palavras-chaves, que são os comandos que compõem a linguagem C. A linguagem de alto nível tipicamente tem várias vezes esse número de palavras reservadas.

Linguagem Estruturada: Uma linguagem estruturada permite muitas possibilidades na programação. Ela suporta, diretamente, diversas construções de laços (loops), como while, do-while e for.

O C é "Case Sensitive", ou seja, maiúsculas e minúsculas fazem diferença. Se a variável for declarada com o nome soma ela será diferente de Soma, SOMA, SoMA, SoMa. Da mesma maneira os comandos em C só podem ser escritos em minúsculas pois senão o compilador não irá interpretá-lo como sendo comandos, mas sim como variáveis.

Linguagem C

Estrutura Básica de um programa C:

```
main( )
{
    }
```

Este programa compõe-se de uma única função chamada main. O parêntese após o nome indica que está é uma função. O nome de uma função C pode ser qualquer um com exceção de "main", reservado para a função que inicia a execução do programa. Toda função em C deve ser iniciada por uma chave de abertura, { e encerrada por uma chave de fechamento}. O nome da função, os parênteses e as chaves são os únicos elementos obrigatórios de uma função.

Vamos fazer nosso primeiro programa em C – O algoritmo

```
algoritmo {O primeiro programa}
                escreva "Olá ! Estamos vivos!";
```

```
fim algoritmo
```

Este algoritmo simplesmente escreve um texto constante na tela. Vamos colocá-lo na linguagem C?

```
/*O Primeiro Programa */
main(){
    printf("Olá! Estamos vivos! \n");
}
```

Compilando e executando este programa você verá que ele coloca a mensagem: Olá! Estamos vivos! na tela.

Analisando o código que escrevemos. Em C qualquer declaração iniciada com /* e terminada com */ ao compilar o programa o sistema entenderá esta linha como comentário. A linha main significa que estamos definindo uma função de nome main. Todos os programas em C devem ser iniciados com main pois é esta função que será chamada quando o programa for executado. O conteúdo da função é delimitado por chaves {}. O código que estiver dentro das chaves será executado sequencialmente quando a função for chamada. A única coisa que o programa realmente faz é chamar a função printf (), passando a string (uma string é uma sequência de caracteres) "Olá! Estamos vivos! \n" como argumento. A função printf () neste caso irá apenas colocar a string na tela do computador, como veremos nos vídeos aulas mais à frente. O \n é uma constante chamada de constante barra invertida. No caso, \n é a constante barra invertida de "new line" e ele é interpretado como um comando de mudança de linha, ou seja, ao imprimir o que o programa pediu, o cursor do texto irá para próxima linha. É importe observar que todos os comandos em C terminam em ; (ponto e vírgula) .

Variáveis

Variável é um local reservado na memória para armazenar um tipo de dado. Toda variável deve ter um identificador, ou seja um nome. Além de ter um nome, a variável também precisa ter um tipo. O tipo de dado de uma variável determina o que ela é capaz de armazenar.

Devemos observar algumas regras para criar os identificadores das variáveis. O nome de uma variável deve sempre iniciar com uma letra (maiúscula ou minúscula) ou com _ , jamais deve iniciar com um número. Pode conter número após o primeiro caractere, e não pode conter em nenhum momento, nenhum caractere especial (!@#$%¨&*'^ç), e caracteres com acentuação.

Tipos de dados

Os dados manipulados em linguagem C são tipados, ou seja, para cada dado usado (nas variáveis, por exemplo) é preciso especificar o tipo de dado, o que permite conhecer a ocupação da memória (o número de bytes) dos dados.

Tipo de dado	**Significado**	**Tamanho (em bytes)**	**Intervalo**

char	caractere	1	de -128 a 127
unsigned char	Caractere não assinado	1	0 à 255
short int	Inteiro curto	2	de -32 768 a 32 767
unsigned short int	Inteiro curto não assinado	2	de 0 a 65 535
int	Inteiro	2 (no processador de 16 bits) 4 (no processador de 32 bits)	de -32 768 a 32 767 de -2 147 483 648 a 2147 483 647
unsigned int	Inteiro não assinado	2 (no processador de 16 bits) 4 (no processador de 32 bits)	de 0 a 65 535 de 0 a 4 294 967 295
long int	Inteiro longo	4	de -2 147 483 648 a 2 147 483 647
unsigned long int	Inteiro longo não assinado	4	de 0 a 4 294 967 295
float	Flutuante (real)	4	$3.4*10^{-38}$ à $3.4*10^{38}$
double			de $1.7*10^{-308}$ a $1.7*10^{308}$

Variáveis podem ser inicializadas a qualquer momento;

```
char a='a';
int first = 0;
int num1, num2;

num1 = num2;
```

Operadores Aritméticos

São símbolos utilizados para realizar as operações aritméticas elementares.

Operador	Ação
+	Soma

-	Subtração
*	Multiplicação
/	divisão
%	Resto da divisão (resto)
++	incremento
--	decremento

Operadores Relacionais

São usados para fazer comparações entre variáveis ou números;

Operador	Ação
=	Igual
<	Menor
>	Maior
<>	Diferente
<=	Menor ou Igual
>=	Maior ou Igual

Operadores Lógicos

São operadores utilizados na junção de operadores relacionais.

Operador	Ação
&&	E
||	Ou
!	Não

Estrutura Sequência

Entrada de dados : Tem por função efetuar a leitura de dados de uma fonte externa.

A função scanf() : É utilizada para fazer a leitura de dados formatados via teclado.

Sintaxe:

```
scanf("expressão de controle", lista de argumentos);
```

Exemplo:

```
scanf("%f", &salario);
```

Explicação: este comando efetua uma leitura do teclado onde é esperada uma variável float (indicada por "%f"). O valor lido será armazenado no endereço do variável salario.

Na lista de argumentos devemos indicar os endereços das variáveis. Para fazer isso adicionamos o símbolo "&" como prefixo na frente do nome da variável.

Tipo de dado	formato
char	%c ou %s
int	%d ou %i
float	%f
double	%lf

Saída de dados: Chamamos de saída de dados a exibição de textos ou valores de variáveis no vídeo.

A função printf(): O comando usado para exibir valores na tela é a função printf().

Sintaxe Básica

```
printf("Mensagem a ser escrita na tela");
```

Também é possível mostrar texto e valores de variáveis usando argumentos.

Sintaxe:

```
printf("Mensagem a ser escrita na tela", lista de argumentos);
```

Exemplo de mensagem que inclui o valor de uma variável:

```
printf("Total a pagar: R$ %f", total);
```

onde:

%f representa o local onde será escrita uma variável float

total é a variável float que será mostrada na posição marcada por %f

Estrutura Condicional

Uma estrutura de decisão examina condições e decide quais instruções serão executadas dependendo da condição.

Estrutura Se Simples – Possui somente a resposta se o teste for verdadeiro;

```
if(<condição>)

        <comandos se Verdadeiro>;
```

Estrutura Se Composta – possui opções para verdadeiro ou falso;

```
If(<condição>)

        <comandos se Verdadeiro>;

        else <comandos se Falso>;
```

É possível construir condicionais duplas, e utilizar os comandos lógicos

```
If((<condição1>)&&(condição2){

        <comandos se V>;

        } else{

                <comandos se F>;

                }

If((<condição1>)||(condição2){

        <comandos se V>;

        } else{

                <comandos se F>;

                }
```

É possível também aninhar os testes, conforme segue:

```
If(<condição>)
```

```
        <comandos se V>;
        else {
                if (<condição>)
                        <comandos se V>;
                        else {
                                if (<condição>)
                                        <comandos se V>;
                                else <comandos se F em todas as
                                anteriores>;
                                }
                }
}
```

A Instrução switch efetua vários testes de valores sobre o conteúdo de uma mesma variável. Esta conexão condicional simplifica muito o teste de vários valores de uma variável, pois esta operação teria sido complicada (mas possível) com ifs imbricados. Sua sintaxe é a seguinte:

```
switch(<variável>){
        case valor1: <comandos se a variável = valor1>;
        break;
        case valor2: <comandos se a variável = valor2>;
        break;
}
```

Estrutura de Repetição

Os loops são estruturas que executam várias vezes a mesma série de instruções até que uma condição não seja mais realizada. Às vezes, chamamos estas estruturas de instruções repetitivas ou iterações. A forma mais comum de fazer um loop é criar um contador (uma variável que se incrementa, ou seja, que aumenta de 1 a cada volta do loop) e parar o loop quando o contador excede um determinado valor.

A instrução for executa várias vezes a mesma série de instruções: é um loop.

Em sua sintaxe, basta especificar o nome da variável que serve de contador (e, eventualmente, o seu valor inicial, a condição sobre a variável para a qual o loop para (basicamente uma condição que testa se o valor do contador excede um limite) e, finalmente, uma instrução que incrementa (ou decrementa) o contador.

A sintaxe desta expressão é a seguinte:

Sintaxe:

```
FOR (Inicialização da variável; condição; incremento)
```

Exemplo:

```
for(x=1;x<=10;x++)
```

Programa exemplo:

```
main(){
     int x;
    for(x=0;x<10;x++)
         printf("x=%d \n", x)
}
```

A instrução while representa outro meio de executar várias vezes a mesma série de instruções. A sintaxe desta expressão é a seguinte:

Sintaxe:

```
Inicialização da variável
While(condição)
```

Exemplo:

```
int x=1;
while(x<=10)
```

Programa exemplo:

```
main() {
        int x=0, total=0;
        while (x<10) {
                total += x; /* igual a total = total + x; */
                printf("x=%d, total=%d\n",x++,total);
        }
}
```

A instrução do .. while garante que o bloco de instruções seja executado no mínimo uma vez, já que a condição que controla o laço é testada apenas no final do comando. A diferença entre o comando while e o do…while é justamente o local onde a condição que controla o laço é testada.

No comando while a condição é testada antes do bloco de instruções, e caso a condição seja falsa a

repetição não será executada. No do…while o bloco de comandos é executado pelo menos uma vez de forma obrigatória, independente do resultado da expressão lógica.

Sintaxe:

```
do {
} while (condição);
```

Exemplo:

```
do {
} while (x <= 10);
```

Programa exemplo:

```
main() {
        int x=0, total=0;
        do {
                total += x; /* igual a total = total + x; */
                printf("x=%d, total=%d\n",x++,total);
        } while (x<=10);
}
```

Estrutura de Repetição Unidimensional

Vetores, também conhecidos como arrays, são variáveis que servem para guardar vários valores do mesmo tipo de forma uniforme na memória. Por exemplo, se tivemos que criar 20 variáveis do mesmo tipo que querem dizer a mesma coisa, nós não criaríamos int x1, x2, x3, x4, x5, ... ao invés disso, criaríamos apenas uma variável de vetor para guardar todos os 20 números de uma vez.

Exato, simples desse jeito.

Como um vetor pode guardar vários valores temos que definir quantos valores ele deve guardar para que seja reservado o espaço necessário em memória. Então, definimos a declaração de um vetor da seguinte maneira:

Primeiro o tipo de dado: int, float, double, ...

Segundo o nome da variável: usando as mesmas convenções de uma variável comum. (array, vetor, variavelDeNumeros, ...)

E por fim, o tamanho necessário do vetor escrito entre colchetes: [5], [10], [3]...

Veja:

Agora que sabemos como criar uma variável de vetor, veremos como ela funciona exatamente.

Todo vetor é um espaço linear na memória dividido de acordo com o tamanho que declaramos. Portanto, se declaramos int array [4], na memória é representado da seguinte forma:

Uma única variável com 4 espaços nos quais podem ser guardados números inteiros.

Agora, para acessarmos um local específico dessa memória devemos indicar entre colchetes a posição desejada no vetor que chamamos de index. O importante mesmo é saber que não importa o tamanho do vetor, o index (número que indica a posição) sempre começa em 0. Portanto, um vetor de tamanho 20 vai da posição 0 a 19, um vetor de tamanho 180 vai da posição 0 a 179, um vetor de tamanho 4 vai da posição 0 a 3.

Agora, se quisermos atribuir os valores 540 na posição 1 e 8456 na posição 3 faríamos:

```
array [1]=540;
array [3]=8456.
```

Não podemos nunca nos esquecer que o limite do vetor é sempre seu tamanho menos 1. Usando o exemplo: vetor de tamanho 4, posição máxima é 3 (pois 4-1=3). Então, se atribuirmos um valor a posição 4 ocorrerá um erro. Resumidamente, jamais poderíamos fazer array [4]=200.

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

int main(){
    int i, vetor[5];
    for (i=0; i<=4; i++){
        vetor[i] = i+5;
    }
    for (i=0; i<=4; i++){
        printf("O valor do vetor na posição %i e %i \n", i+1,
vetor[i]) ;
    }
}
```

No código acima criamos um vetor de tamanho 5 e usamos uma iteração (repetição / loop) para preencher todos os espaços do vetor e então outra iteração para mostrar todos os valores guardados. Para mudarmos as posições do vetor usamos uma variável chamada i.

Erros comuns:

Ao desenvolver seus programas com vetores, preste atenção com relação aos seguintes detalhes:

índices inválidos: tome muito cuidado, especialmente dentro de um while ou for, de não utilizar índices negativos ou maiores que o tamanho máximo do vetor.

Definição do tamanho do vetor: se faz na declaração do vetor. O tamanho dele é constante, só mudando a sua declaração é que podemos alterar o seu tamanho. Isso significa que podemos estar "desperdiçando" algum espaço que fica no final do vetor.

Não cometa o erro de ler **n**, onde n seria o tamanho do vetor, e tentar "declarar" o vetor em seguida.

Estrutura de Repetição Multidimensional

Uma matriz multidimensional, como o próprio nome diz, nada mais é do que uma matriz com múltiplos seguimentos de dados em memória, que podem ser acessador por um conjunto de índices que identifica o valor em um item do array.

A declaração da matriz ou array multidimensional é efetuada com múltiplos colchetes, que informam a quantidade de elementos que a matriz poderá conter, podendo ser atribuído valores aos itens em sua declaração inicial ou, atribuir valores durante a demanda do processamento da aplicação de acordo com as necessidades do algoritmo.

Matriz em linguagem C

Exemplo de declaração de matriz com 2 dimensões usando linguagem C

```
float Media[5][2];
```

Onde:

O valor 5 representa a quantidade de linhas.

O valor 2 representa a quantidade de colunas.

Dizemos que esta matriz é do tipo 5 X 2.

Como temos 5 linhas com 2 posições de armazenamento em cada linha, temos capacidade para armazenar até 10 elementos (itens) do tipo float.

Será necessário utilizar um índice para cada dimensão da matriz, logo uma matriz bidimensional terá 2 índices, um para posicionar a linha, outro para a coluna.

Assim, como no vetor, o índice da primeira posição é zero.

Como atribuir valores a uma matriz?

Suponha a matriz como declarada:

```
float Media[5] [2];
```

para atribuir um valor precisamos identificar a posição usando os índices:

```
Media [0][0] = 5; //Atribui o valor 5 na primeira linha e primeira coluna.

Media [1][0] = 7; // Atribui o valor 7 na segunda linha, primeira coluna.
```

Para fazer o preenchimento de uma matriz, devemos percorrer todos os seus elementos e atribuir-lhes um valor. Isto pode ser feito tanto gerando valores para cada elemento da matriz, como recebendo os valores pelo teclado. Um método interessante para percorrer uma matriz é usar duas estruturas de repetição for e duas variáveis inteiras, uma para a linha e a outra para a coluna.

Exemplo:

Suponha uma matriz de 3 linhas por 3 colunas do tipo inteiro.

Para percorrer a matriz recebendo seus valores, podemos fazer:

```
for ( i=0; i<3; i++ ){
    for ( j=0; j<3; j++ ){
      scanf ("%d", &matriz[ i ][ j ]);
    }
}
```

Neste trecho de código, a variável i representa a linha i e j a coluna da matriz. Usando as repetições, percorremos cada posição da matriz e recebemos um valor usando scanf().

Para mostrar os elementos de uma matriz, podemos usar o mesmo método, porém usamos o comando de saída printf().

```
for ( i=0; i<3; i++ ){
  for ( j=0; j<3; j++ ){
     printf ("%d", matriz[ i ][ j ]);
  }
}
```

Exemplo:

Programa em C que preenche uma matriz com 3 linhas X 3 colunas e depois mostra os dados recebidos.

```
#include<stdio.h>
#include<conio.h>
int main (void ){
        int matriz[2][2],i, j;

 printf ("\nDigite valor para os elementos da matriz\n\n");

 for ( i=0; i<3; i++ ){
        for ( j=0; j<3; j++ ){

                 printf ("\nElemento[%d][%d] = ", i, j);
                scanf ("%d", &matriz[ i ][ j ]);
        }
}

 printf("\n\n************Saida de Dados *************** \n\n");

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

 getch();
 return(0);
}
```

Procedimento

Os procedimentos são estruturas que agrupam um conjunto de comando, que são executados quando o procedimento é chamado.

A utilização de procedimentos e funções tende a manter uma melhor organização do programa, ao reaproveitamento de códigos construídos anteriormente, a evitar a repetição de trechos de códigos, minimizando erros e facilitando alterações.

Por exemplo, se quisermos calcular a média aritmética das provas de um aluno, temos que somar a nota da primeira prova com a nota da segunda prova e dividir por 2. Se tivermos mais de um aluno, temos que repetir esse cálculo para cada aluno. Com um simples “Copia e cola” poderia resolver o problema da repetição, mas será que foi a melhor escolha? E se tivermos 100 alunos? E se após termos escrito o programa todo, alguém nos disser que eram 3 provas? Em ambos os casos teremos um bom trabalho para ajustar o código do programa.

Diante desse problema, podemos afirmar que a repetição de trechos de código dentro de um programa é considerada uma prática muito ruim.

Quando o procedimento é chamado, a execução do programa que o chamou é interrompida, passando o controle ao procedimento chamado. Após a execução do procedimento, o controle retorna ao programa que efetuou a chamada no comando imediatamente subsequente, e a execução do programa continua a partir desse ponto.

Calma vamos explicar melhor, não existe retorno de um único valor. Qualquer valor a ser retornado por um procedimento retorna ao programa que efetuou a chamada por meio de seus parâmetros.

Geralmente utilizamos procedimentos quando precisamos utilizar alguns códigos que vamos ter que ficar repetindo, sendo assim um procedimento concentra nossos códigos e podemos a qualquer momento o chamar.

Todo procedimento, é do tipo VOID.

MAIS COMO USAR UM PROCEDIMENTO? E O QUE É VOID?

Void em inglês significa vazio, e é exatamente isto que um procedimento é, ele nos permite fazer uma função que não retorna nada.

Void é um tipo que não representa "nada", ou seja, uma função desse tipo retorna um conteúdo indeterminado.

Vamos dar uma olhada no exemplo:

```
void nome_do_procedimento (<tipo> parâmetro1, <tipo> parâmetro2, ..., <tipo> parâmetroN) {

<corpo do procedimento>

}
```

Para declarar um procedimento, basta:

```
void imprime_dobro (int x) {

        printf(" Dobro de X: %d", 2*x);

}
```

Para chamar um procedimento, devemos utilizá-lo como qualquer outro comando:

```
procedimento(parâmetros);
```

Exemplo:

```
int x, y, resultado;

void soma(){ // "()" procedimento sem parâmetro
        resultado = x + y;
}

int main(){
```

```
        x = 3;
        y = 5;
        soma (); // chamando o procedimento
        printf("%d\n", resultado); // mostrando o procedimento
}
```

Os procedimentos são utilizados com muita frequência em desenvolvimento de softwares. São vários os benefícios, ele evita duplicação de código quando precisamos executar a mesma operação várias vezes, deixa o entendimento do algoritmo mais intuitivo, pois tiramos a parte complexa do código do fluxo principal do algoritmo, etc.

Pratique identificando no seu código, onde você pode utilizar funções e procedimentos. Um exemplo, é unir em um só procedimento aquele código de leitura de valores do usuário que sempre utilizamos:

```
printf ("Informe o valor de A:");

scanf("%d",&a)3B
```

Funções

As funções são unidades de código desenhadas para cumprir uma determinada tarefa. O código só é executado quando a função é chamada no programa que está ligado. Funções são procedimentos que retornam um valor ao seu termino.

Sempre que alguma delas é chamada, o programa que a chamou fica suspenso temporariamente, em seguida são executadas todas as instruções presentes no corpo da função. Uma vez terminada, o controle de execução do programa volta no ponto em que foi chamado.

O programa que chamou uma função pode enviar dados e esses dados são recebidos pela função e armazenados em variáveis locais, essas variáveis chamam-se parâmetros das funções.

Depois de a função terminar o seu funcionamento, a mesma pode devolver um valor para o programa que a chamou.

Vamos conferir a estrutura de uma função:

```
tipo nome_da_função (declaração dos parâmetros de entrada) {
        declaração das variáveis locais; //Variáveis que só são visíveis pela função
            instruções;
              return valor; //Volta para o programa que chamou a função
    }
```

O tipo de retorno pode ser int, float, char, double ou void.

Exemplo de um programa com uma função para o calculo da área de um círculo:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.14159 //constante pi

float calcula_area_circulo (int raio){
    float area;
    area = PI * raio * raio;
    return area;
}

int main() {
        float Area;
        int Raio;

        printf("Insira o raio: ");
        scanf("%d", &Raio);

        Area = calcula_area_circulo (Raio);

        printf("%.3f", Area);
        exit(0);

}
```

A função é declarada antes do programa principal main, cria-se um cabeçalho, no caso desta função será, "float calcula_area_circulo(int raio) {...}"

Float é o tipo de dado da função.

calcula_area_circulo Nome da função.

(int raio) parâmetros

Uma função simples para calcular a área de um círculo, sabemos que para se obter esse resultado deve-se calcular:

Área = PI * raio² que é o mesmo que Área = PI * raio * raio

Fazemos esse cálculo dentro da função calcula_area_circulo, com isso poderemos realizar vários cálculos de diferentes círculos utilizando a mesma estrutura.

Após isso passaremos ao programa principal.

O programa principal (main) é montado normalmente, pode-se declarar variáveis com o mesmo nome que foi declarado na função calcula_area_circulo, como por exemplo, float área, declarado dentro da função, e float Area, declarado dentro do programa principal, isso é permitido, pelo fato das variáveis declaradas serem visíveis apenas dentro das funções.

Após chamar uma função, deve-se modificar a variável de parâmetro, onde a mesma modificara as variáveis dentro da função, por exemplo, ao modificar "... (int raio)" da função, por "... (Raio)" do programa principal, irá alterar todo a palavra raio de dentro da função pelos valores inseridos dentro do programa principal.

As razões que nos levam a utilizar ou escrever as funções são reduzir a complexidade de um programa e evitar a repetição de código.

As funções normalmente devolvem um valor através da instrução return, que não é obrigatória.

As funções que não devolvem qualquer valor são chamas procedimentos.

A função do código

```
float calcula_area_circulo (int raio)
{
   float area;
   area = PI * raio * raio;
   return area;
}
```

Porque usar funções?

- Para permitir o reaproveitamento de código já construído (por você ou por outros programadores).
- Para evitar que um trecho de código que seja repetido várias vezes dentro de um mesmo programa.
- Para permitir a alteração de um trecho de código de uma forma mais rápida. Com o uso de uma função é preciso alterar apenas dentro da função que se deseja.
- Para que os blocos do programa não fiquem grandes demais e, por conseqüência, mais difíceis de entender.
- Para separar o programa em partes(blocos) que possam ser logicamente compreendidos de forma isolada.

Função vai facilitar a leitura do seu código e deixar sua vida mais fácil.

Parâmetros

Parâmetros ou argumentos são os valores recebidos ou retornados por uma função, eles podem ser divididos em duas categorias que são:

Formais – Correspondem aos parâmetros utilizados na definição da função.

Reais – Correspondem aos parâmetros da função chamadora utilizados para chamar a função.

```
int soma (int a, int b){  // Parâmetros formais
        return (a+b);
```

```
}

int main () {
       int x = 3;
       int y = 5;
       printf("%d\n", soma(x + y)); // Parâmetros reais
}
```

Passagem de parâmetros: É o mecanismo de informar sobre quais valores o processamento definido na função deve ser realizado. Os parâmetros são passados para uma função de acordo com a sua posição. Os parâmetros formais de uma função comportam-se como variáveis locais (criados na entrada e destruídos na saída). Existem duas categorias:

Valor: É a forma mais comum utilizada para passagem de parâmetros. Por exemplo, considere a família de funções trigonométricas, como seno, cosseno, etc. A função seno, por exemplo, recebe o valor de um ângulo (um número real) e devolve o seno desse ângulo. Se tivermos as funções seno e cosseno, podemos facilmente definir uma função tangente. Em projetos grandes de desenvolvimento de software, grupos de programadores podem trabalhar no desenvolvimento de funções distintas e juntar os seus trabalhos uma vez que tenham suas funções prontas. Para isso, basta que cada grupo conheça o protótipo das funções que precisa utilizar, e ao final, um programa chamado "linker" é responsável por juntar os pedaços e construir um programa executável. Por exemplo, considere que nós temos disponíveis os seguintes protótipos para as funções seno e cosseno:

```
float seno (float angulo);
float cosseno (float angulo);
```

Podemos escrever a função tangente da seguinte forma:

```
float tangente (float angulo) {
   float s, c;
   s = seno(angulo);
   c = cosseno(angulo);
   return s/c;
}
```

Essa função claramente calcula o seno antes de calcular o cosseno. Imagine se a função seno pudesse modificar o valor do parâmetro angulo, nesse caso, o valor passado para a função cosseno seria diferente do valor original usado para o seno, e o resultado da função tangente estaria incorreto. A passagem por valor consegue evitar esse tipo de "efeito colateral" através da criação de variáveis locais para os parâmetros. Quando uma função é chamada, essas variáveis são carregadas, como em atribuições, antes do início da execução do corpo da função.

Referência: Para passarmos uma variável para uma função e fazer com que ela seja alterada, precisamos passar a referência dessa variável, em vez de seu valor. Até o momento, vínhamos passando somente o valor das variáveis.

Conforme explicamos, quando passamos um valor, a função copia esse valor e trabalha somente em cima da cópia dessa variável, e não na variável em si. Por isso nossas variáveis nunca eram alteradas quando passadas

para funções. Porém, é muito importante e útil que algumas funções alterem valores de variáveis. Para fazer isso, usaremos um artifício chamado Passagem por Referência.

Por referência entenda endereço, um local. No caso, em vez de passar o valor da variável, na passagem por referência vamos passar o endereço da variável para a função. Para fazer isso, basta colocar o operador & antes do argumento que vamos enviar, e colocar um asterisco * no parâmetro da função, no seu cabeçalho de declaração, para dizer a função que ela deve esperar um endereço de memória, e não um valor.

Sim, parece confuso e difícil entender assim de cara, mas vamos mostrar um exemplo em que usaremos a passagem por referência.

Exemplo de código: Passagem por referência em C

Crie um programa que recebe um inteiro e dobra seu valor. Para que uma função altere o valor de uma variável, é necessário que essa função atue no endereço de memória. Para isso, temos que passar o endereço da variável para a função(usando &), e a função tem que ser declarada de modo a esperar um ponteiro (usando *).

Dentro da função, devemos trabalhar com o valor na qual aquele ponteiro aponta. Ou seja, se o ponteiro tiver nome 'ptr', devemos trabalhar com o valor '*ptr'. Veja como ficou nosso código C para entender melhor:

```
#include <stdio.h>
void dobra(int *num){
    (*num) = (*num) * 2;
}
int main(void){
        int num;
        printf("Insira um numero: ");
        scanf("%d", &num);

        dobra(&num);

        printf("O dobro dele eh: %d\n", num);

        return 0;
}
```

Resumindo, quando desejamos criar uma função que devolva mais de um valor, precisamos definir esses parâmetros com um caractere '*' no protótipo da função, e ao chamar a função, os parâmetros utilizados na chamada correspondentes à saída precisam ser precedidos pelo caractere '&' para indicar que eles podem ser modificados pela função chamada.

Registros

Uma estrutura (struct) é uma coleção de uma ou mais variáveis, possivelmente de tipos diferentes, agrupadas sob um único nome. Estruturas constituem um recurso importante para organizar os dados utilizados por um programa pois trata de um grupo de valores como uma única variável. São chamadas de registros em outras linguagens de programação.

Estruturas ou registros são classificados como variáveis compostas heterogêneas, pois podem agrupar variáveis de tipos diferentes. Em contraposição, temos os vetores e matrizes, classificados como variáveis compostas homogêneas, pois somente agrupam variáveis do mesmo tipo.

Exemplo:

```
struct data{
 int dia;
 int mes;
 int ano;
};
```

A palavra-chave struct informa ao compilador que um modelo de estrutura está sendo definido.

"data" é uma etiqueta que dá nome à definição da estrutura.

Uma definição de estrutura é um comando, por isso deve terminar em ponto-e-vírgula.

Os nomes declarados entram as chaves são os campos (ou membros) da estrutura.

Os campos de uma mesma estrutura devem ter nomes diferentes. Porém, estruturas diferentes podem conter campos com o mesmo nome.

A definição de uma estrutura não reserva qualquer espaço na memória.

Note que, no exemplo dado, nenhuma variável foi declarada de fato, apenas a forma dos dados foi definida. Essa definição, porém, cria um novo tipo de dados, que pode ser usado para declarar variáveis.

Declarando um registro

Duas maneiras de declarar a variável x do tipo data:

```
struct data{
 int dia;
 int mes;
 int ano;
};
...
struct data x;
```

Dois comandos:

Define estrutura como novo tipo.

Declara variável do novo tipo definido.

ou

```
struct data {
 int dia;
 int mes;
 int ano;
} x;
```

Um comando:

Define estrutura e declara variável do novo tipo definido.

Os campos de uma estrutura podem ser de qualquer tipo, inclusive uma estrutura previamente definida. Porém, o tipo dos campos não pode ser do próprio tipo que está sendo definido.

A definição do formato de uma estrutura pode ser feita dentro da função principal (main) ou fora dela.

Usualmente, declara- se fora da função principal, de modo que outras funções também possam "enxergar" a estrutura definida.

Forma geral de definição de um registro:

```
struct <etiqueta> {
 <tipo> campo_1;
 <tipo> campo_2;
 ...
 <tipo> campo_n;
} <varáveis>;
```

Declarando um registro utilizando typedef

Novos tipos de dados podem ser definidos utilizando-se a palavra-chave typedef.

```
typedef struct nome_da_estrutura{
 <tipo> campo_1;
 <tipo> campo_2;
 ...
 <tipo> campo_n;
} nome_do_tipo;
```

Exemplo:

```
typedef struct data{
 int dia;
 int mes;
 int ano;
} tipoDta;
```

O uso mais comum de typedef é com estruturas de dados, pois evita que a palavra-chave struct tenha de ser colocada toda vez que uma estrutura é declarada.

```
struct data dia_de_hoje;

tipoData dia_de_hoje;
```

Podemos acessar individualmente os campos de uma determinada estrutura como se fossem variáveis comuns.

A sintaxe para acessar e manipular campos de estruturas é a seguinte:

```
<nome_da_vari-avel>.<campo>
```

A leitura dos campos de uma estrutura a partir do teclado deve ser feita campo a campo, como se fosse variáveis independentes.

```
printf ("Digite o nome do aluno: ");
scanf("%s", &aluno.nome);

printf ("Digite a idade do aluno: ");
scanf ("%d", &aluno.idade);
```

Vamos dar uma olhada nesse exercício para entender melhor.

```
#include <stdio.h>
#include <conio.h>
int main(void){
 /*Criando a struct */
       struct ficha_de_aluno {
              char nome[50];
              char disciplina[30];
              float nota_prova1;
              float nota_prova2;
       };

 /*Criando a variável aluno que será do tipo struct ficha_de_aluno */
 struct ficha_de_aluno aluno;

 printf("\n---------- Cadastro de aluno ----------\n\n\n");

 printf("Nome do aluno ......: ");
 fflush(stdin);

 /*usaremos o comando fgets() para ler strings, no caso o nome
 do aluno e a disciplina
 fgets(variavel, tamanho da string, entrada)
 como estamos lendo do teclado a entrada é stdin (entrada padrão),
 porém em outro caso, a entrada tambem poderia ser um arquivo */

 fgets(aluno.nome, 40, stdin);
```

```
 printf("Disciplina ......: ");
 fflush(stdin);
 fgets(aluno.disciplina, 40, stdin);

 printf("Informe a 1a. nota ..: ");
 scanf("%f", &aluno.nota_prova1);

 printf("Informe a 2a. nota ..: ");
 scanf("%f", &aluno.nota_prova2);

 printf("\n\n --------- Lendo os dados da struct ---------\n\n");
 printf("Nome ............: %s", aluno.nome);
 printf("Disciplina .....: %s", aluno.disciplina);
 printf("Nota da Prova 1 ...: %.2f\n" , aluno.nota_prova1);
 printf("Nota da Prova 2 ...: %.2f\n" , aluno.nota_prova2);

 getch();
 return(0);
}
```

Tela de execução utilizando struct

Ponteiros

A utilização de ponteiros em linguagem C é uma das características que tornam a linguagem tão flexível e poderosa. Ponteiros ou apontadores, são variáveis que armazenam o endereço de memória de outras variáveis. Dizemos que um ponteiro “aponta” para uma variável quando contém o endereço da mesma.

Os ponteiros podem apontar para qualquer tipo de variável. Portanto temos ponteiros para int, float, double, etc.

Por que usar ponteiros?

Ponteiros são muito úteis quando uma variável tem que ser acessada em diferentes partes de um programa. Neste caso, o código pode ter vários ponteiros espalhados por diversas partes do programa, “apontando” para a variável que contém o dado desejado. Caso este dado seja alterado, não há problema algum, pois todas as partes do programa tem um ponteiro que aponta para o endereço onde reside o dado atualizado.

Existem várias situações onde ponteiros são úteis, por exemplo:

Alocação dinâmica de memória

Manipulação de arrays.

Para retornar mais de um valor em uma função.

Referência para listas, pilhas, árvores e grafos.

Sintaxe de declaração de ponteiro

```
tipo *nome_ponteiro;
```

Onde temos:

tipo : é o tipo de dado da variável cujo endereço o ponteiro armazena.

*nome_ponteiro : O nome da variável ponteiro.

O asterisco * neste tipo de declaração determina que a variável será um ponteiro.

Exemplo de declaração de ponteiro:

int *ptr;

Exemplo:

Programa utilizando ponteiro

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

int main(void){
 //valor é a variável que
 //será apontada pelo ponteiro
 int valor = 27;

 //declaração de variável ponteiro
 int *ptr;

 //atribuindo o endereço da variável valor ao ponteiro
 ptr = &valor;

 printf("Utilizando ponteiros\n\n");
 printf ("Conteudo da variavel valor: %d\n", valor);
 printf ("Endereço da variavel valor: %x \n", &valor);
 printf ("Conteudo da variavel ponteiro ptr: %x", ptr);

 getch();
 return(0);
}
```

Note que foi usado o operador * para designar que a variável ptr é um ponteiro. Como a intenção é armazenar o endereço da variável denominada valor que é uma variável do tipo int, o ponteiro também tem que ser do tipo int. Isto significa que vai apontar para uma variável do tipo inteiro.

Para atribuir o endereço da variável valor ao ponteiro ptr utilizamos o operador de endereço &, pois estamos nos referindo ao endereço da variável valor e não ao conteúdo da mesma.

```
printf("Utilizando ponteiros\n\n");
```

```
printf ("Conteudo da variavel valor: %d\n", valor);

printf ("Endereço da variavel valor: %x \n", &valor);

printf ("Conteudo da variavel ponteiro ptr: %x", ptr);
```

Foi utilizado %x para exibir o endereço e o conteúdo do ponteiro ptr, pois trata-se de um valor hexadecimal por ser endereço de memória.

Arquivos

Os arquivos permitem gravar os dados de um programa de forma permanente em mídia digital.

Vantagens de utilizar arquivos

Armazenamento permanente de dados: as informações permanecem disponíveis mesmo que o programa que as gravou tenha sido encerrado, ou seja, podem ser consultadas a qualquer momento.

Grande quantidade dados pode ser armazenada: A quantidade de dados que pode ser armazenada depende apenas da capacidade disponível da mídia de armazenamento. Normalmente a capacidade da mídia é muito maior do que a capacidade disponível na memória RAM.

Acesso concorrente: Vários programas podem acessar um arquivo de forma concorrente.

A linguagem C trata os arquivos como uma sequência de bytes. Esta sequência pode ser manipulada de várias formas e para tanto, existem funções em C para criar, ler e escrever o conteúdo de arquivos independente de quais sejam os dados armazenados.

Em C trabalhamos com dois tipos de arquivos:

Arquivo texto armazena caracteres que podem ser mostrados diretamente na tela ou modificados por um editor de texto.

Arquivo binário é uma sequência de bits que obedece a regras do programa que o gerou.

O ponteiro para arquivo

Em C, o arquivo é manipulado através de um ponteiro especial para o arquivo. A função deste ponteiro é "apontar" a localização de um registro.

Sintaxe:

```
FILE < *ponteiro >
```

O tipo FILE está definido na biblioteca stdio.h.

Exemplo de declaração de um ponteiro para arquivo em C:

```
FILE *pont_arq;
```

Lembrando que FILE deve ser escrito em letras maiúsculas.

Operações com arquivos do tipo texto

Abertura de arquivos.

Para trabalhar com um arquivo, a primeira operação necessária é abrir este arquivo.

```
< ponteiro > = fopen("nome do arquivo","tipo de abertura");
```

A função `fopen` recebe como parâmetros o nome do arquivo a ser aberto e o tipo de abertura a ser realizado.

Depois de aberto, realizamos as operações necessárias e fechamos o arquivo.

Para fechar o arquivo usamos a função `fclose`.

```
fclose< ponteiro >;
```

Lembrando que o ponteiro deve ser a mesma variável ponteiro associada ao comando de abertura de arquivo.

Tipos de abertura de arquivos

Opção	Ação
r	Abre o arquivo somente para leitura, a partir do início. O arquivo deve existir.
w	Cria um arquivo vazio para escrita. Se já havia o arquivo, ele é perdido.
a	Adiciona no final do arquivo. Se o arquivo não existir, a função cria.
r+	Abre o arquivo para leitura e escrita, a partir do início. O arquivo deve existir
w+	Cria um arquivo vazio para leitura e escrita. Se já havia o arquivo, ele é perdido.
a+	Abre para adição ou leitura no final do arquivo. Se o arquivo não existir, a função cria.

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

int main(void) {
        // criando a variável ponteiro para o arquivo
        FILE *pont_arq;

        //abrindo o arquivo
```

```
        pont_arq = fopen("arquivo.txt", "a");

        // fechando arquivo
        fclose(pont_arq);

        //mensagem para o usuário
        printf("O arquivo foi criado com sucesso!");

        system("pause");
        return(0);
}
```

Na prática, nem sempre é possível abrir um arquivo. Podem ocorrer algumas situações que impedem essa abertura, por exemplo:

Você está tentando abrir um arquivo no modo de leitura, mas o arquivo não existe;

Você não tem permissão para ler ou gravar no arquivo;

O arquivo está bloqueado por estar sendo usado por outro programa.

Quando o arquivo não pode ser aberto a função `fopen` retorna o valor NULL.

É altamente recomendável criar um trecho de código a fim de verificar se a abertura ocorreu com sucesso ou não.

```
if (pont_arq == NULL){
    printf("ERRO! O arquivo não foi aberto!\n");
} else    {
     printf("O arquivo foi aberto com sucesso!");
   }
```

A função fprintf armazena dados em um arquivo. Seu funcionamento é muito semelhante ao printf, a diferença principal é a existência de um parâmetro para informar o arquivo onde os dados serão armazenados.

Sintaxe:

```
fprintf(nome_do_ponteiro_para_o_arquivo, "%s",variavel_string)
```

//Exemplo: Abrindo, gravando e fechando arquivo

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


int main(void){
 FILE *pont_arq; // cria variável ponteiro para o arquivo
```

```
 char palavra[20]; // variável do tipo string
 //abrindo o arquivo com tipo de abertura w
 pont_arq = fopen("arquivo_palavra.txt", "w");

 //testando se o arquivo foi realmente criado
 if(pont_arq == NULL){
       printf("Erro na abertura do arquivo!");
       return 1;
 }
 printf("Escreva uma palavra para testar gravacao de arquivo: ");
 scanf("%s", palavra);

 //usando fprintf para armazenar a string no arquivo
 fprintf(pont_arq, "%s", palavra);

 //usando fclose para fechar o arquivo
 fclose(pont_arq);

 printf("Dados gravados com sucesso!");

 getch();
 return(0);
}
```

Leitura caracter por caracter – Função getch() - faz a leitura de um caracter no arquivo.

Sintaxe:

```
getc(ponteiro_do_arquivo);
```

Para realizar a leitura de um arquivo inteiro caracter por caracter podemos usar getch dentro de um laço de repetição.

```
do{
    //faz a leitura do caracter no arquivo apontado por pont_arq
    c = getc(pont_arq);

    //exibe o caracter lido na tela
    printf("%c" , c);

}while (c != EOF);
```

A leitura será realizada até que o final do arquivo seja encontrado.

Função fgets() - É utilizada para leitura de strings em um arquivo. Realiza a leitura dos caracteres até o final da linha quando encontra o caracter \n. A leitura é efetuada de tal forma que a string lida é armazenada em um ponteiro do tipo char. A função pode ser finalizada quando encontrar o final do arquivo, neste caso retorna o endereço da string lida. Se ocorrer algum erro na leitura do arquivo, a função retorna NULL.

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

int main(void){
 FILE *pont_arq;
 char texto_str[20];

 //abrindo o arquivo_frase em modo "somente leitura"
 pont_arq = fopen("arquivo_palavra.txt", "r");

 //enquanto não for fim de arquivo o looping será executado
 //e será impresso o texto
 while(fgets(texto_str, 20, pont_arq) != NULL)
 printf("%s", texto_str);

 //fechando o arquivo
 fclose(pont_arq);

 getch();
 return(0);
}
```

Referências Bibliográficas

FARRER, Harry. Algoritmos Estruturados. 3 ed. São Paulo. LTC, 1999.

ASCENCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Verenuchi de. Fundamentos da Programação de Computadores: algoritmos, Pascal e C/C++. São Paulo: Pearson Education, 2009.

SCHILDT, Herbert. C Completo e Total. Tradução de Roberto Carlos MAYER. 3 ed. São Paulo: Makron Books, 2006.

DEITEL, Paul J. DEITEL, Harvey M. C: como programar. 6 ed. São Paulo: Pearson Education Hall, 2011.

FEOFILOFF, Paulo. Algoritmos em linguagem C. Rio de Janeiro: Elsevier, 2009.

FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. Lógica de Programação. 3 ed. São Paulo: Pearson Education Hall, 2005.

SEDGEWICK, Robert. Algorithms in C: graph algorithms. 3 ed. Addison Wesley Longman, 2002.

VELOSO, Paulo at al. Estrutura de Dados. Rio de Janeiro: Campus, 1983.