**AS MANHAS DE C**

Sumário

[Sintaxe mais simples 49](#_Toc122373376)

**PRIMEIRO CONTATO:**

• Cada projeto C, suporta um programa C

• Interessante criar um sistema de pastas para guarda os códigos fontes dos exercícios

#include <stdio.h>

Int main(){

Printf(“Hello World!”);

Return 0;

}

**ENTENDENDO UM POUCO DA PRIMEIRA SITNAXE:**

#include <stdio.h> // é uma biblioteca arquivos cabeçalhos, que trabalha com funções de entrada de dados e saídas, como o printf

Int main(){} // chamada de função na qual qualquer código entre chaves será executado.

Printf(“”) // Basicamente é uma função que escreve na tela uam saída padrão qualquer, colocada entre parênteses

Por de trás, é como se fosse char nome[numero de caracteres]

Um array de letras.

Return 0; Finaliza a função main(), porem é um conceito mais avançado.

Na função *int main*, tudo que está dentro de { } sera executado.

Toda instrução ira terminar com ;

int main(){printf("Hello World!");return 0;}, é tudo uma linguiça de bits no final é assim que o compilador enxerga o código, mas espaçamos para ser mais legível.

**QUEBRANDO LINHAS:**

Printf(“Algo!”)

Printf(“Legal”)

Sairia como: Algo!Legal, logo o programa não adiciona espaços sozinho, você precisa usar o comando \n dentro da string.

Printf(“Algo!\n”)

Printf(“Legal”)

Ou

Printf(“Algo!\n\n”) // Nesse caso para dar mais espaço.

Printf(“Legal!”)

Saira:

Algo!

Legal

Podes encontrar mais quebras de linhas, nos códigos fontes.

**COMENTARIOS:**

Podem ser usados para deixar o código mais legível, explicando as partes deles,

Como também pode ser usado para comentar códigos alternativos, logo se um não der certo, descometa o outro possível código

**Sintaxe:** // Algo

Qualquer texto em uma linha depois de // é ignorado pelo compilador.

**Comentando varias linhas:**

/\* Pega tudo que está

Aqui dentro disso \*/

**VARIAVEIS:**

Em C existem diferentes tipos de variáveis, definidas por palavras chaves diferentes.

**Exemplo:**

Int – Armazena números inteiro, sem decimais, como 123 ou -123

Float – Armazena números de ponto flutuante, com decimais como 19,99 ou -19,99

Char – armazena caracteres únicos, como ‘a’ ou ‘B’, todo caractere estará entre aspas simples.

**Declarando Essas Variáveis:**

**Exemplo de sintaxe:**

Tipo nomeVariavel = valor;

Ou

Tipo nomeVariavel;

**Exemplo real:**

Int meuNumero = 15;

Ou

Int meuNumero;

meuNumero = 15; //exemplo de atribuição do valor.

Printf(“%n”, meuNumero)

// já já falaremos sobre as formatações de saídas de dados. Ou melhor “Especificadores de formato”

OBS: Se você atribuir um valor a uma variável existente, ela substituirá o valer anterior.

**ESPECIFICADORES DE FORMATO:**

São usados juntos ao Printf()

Que será basicamente um espaço reservado para informar ao compilador o tipo de valor da variável que sera gerado dentro da função Printf.

**Especificadores em si:**

Int – “%d” ou “%i” //para valores inteiros.

Char – “%c”

Float – “%f”

**Exemplo:**

#include <stdio.h>

int main(){

// declarando variaveis

int inteiro = 5;

float real = 31.55;

char nome = 'D';

// Printando valores

printf("%d\n", inteiro);

printf("%0.1f\n", real); // %0.2f" o 0.2 define quantas casa decimais o numero exibira.

printf("%c\n", nome); // só recebe um caractere, pois para declarar nomes inteiros, é necessário fazer um array de char.

return 0;

}

**DECLARANDO UMA VARIÁVEL COM UM TEXTO:**

Int meuNumero = 5;

Printf(“Meu numero favorito é: %d”, meuNumero)

Ou imprimindo diferentes tipos de números:

Int numb = 5;

Char letra = ‘n’;

Printf(“Meu numero e: %d e minha letra é %c”, numb, letra);

**ADICIONANDO VARIÁVEIS JUNTAS:**

Int x = 5;

Int y = 6;

Int sum = x + y;

Printf(“%d”, sum);

**DECLARANDO VÁRIAS VARIÁVEIS:**

Int x = 5, y = 6, z = 50;

Printf(“%d”, x + y + z);

**NOMES:**

Obviamente é indicado que, usem-se nomes descritivos no padrão camel case, já conhecido.

* Nome podem conter letras, dígitos e sublinhados
* Nomes devem começar com uma letra ou sublinado(\_)
* Os nomes diferenciam maiúsculas e minúsculas! Myvar e myvar são variáveis diferentes para o compilador.
* Os nomes não podem contem espaços em brancos ou caracteres especiais, (#,!,% etc)
* Palavras reservadas não podem ser usadas! Int, char, float etc.

**TIPOS DE DADOS E SEUS ESPECIFICADORES:**

**Int** – 2 ou 4 bits , armazena números inteiros **(“%d”** ou **“%i” )**

**Float** – 4 bits , armazena números fracionados, contendo um ou mais decimais e é suficiente para armazenar 7 dígitos. **(“%f”)**

**Double** – 8 bits, armazena números fracionários, contendo uma ou mais casas decimais. É suficiente para armazenar um número de 15 dígitos decimais.

**(“%lf”)**

**Char** – Armazena um único caractere/letra/número ou valores da tabela ASCII.

**(“%s”)** usado para strings de textos, nas quais veremos em breve

Se for so para exibir uma letra pode usar o “%c”

**(“%lu”)** Usado para exibir o tamanho em bytes de uma variável.

**(“%P”)** Exibição de ponteiros.

**Constantes:**

São usadas para a criação de variáveis constantes que não são mutáveis de forma alguma

Pra isso, usamos a palavra const

Assim a variável será imutável, somente servindo para leitura.

Exemplos:

Const int myNum = 15;

myNum = 10; // Vai acontecer um erro, logo a variável myNum é umas constante imutável.

Sempre devemos declarar constantes, em casos de valores que obviamente não serão alterados

Exemplo:

# include <stdio.h>

int main()

{

const int MINUTESPERHOUR = 60;

const float PI = 3.14;

printf("%d\n", MINUTESPERHOUR);

printf("%0.2f", PI);

return 0;

}

**OBS:** Ao criar uma varaivel constante, nela já se deve atribuir um valor,

Logo, fazer isso mais tarde não será possivel isso não é possível:

Const int minutesPerHous;

MINUTESPERHOUR = 60;

Printf(“%d”,MINUTESPERHOUR); // isso daria erro, logo MINUTEPERHOUR esta com o valor constante vazio, e o mesmo não pode ser mudado.

É uma boa pratica, declarar variáveis constantes com letras maiúsculas.

**OPERADORES**

Usamos operadores para fazer operações em variáveis. Já veremos também a ordem de resolução.

Exemplo:

Int myNum = 100 + 50;

Podemos somar também, variável com valor, duas variáveis, etc..

Int sum1 = 100 + 50;

Int sum2 = sum1 + 250;

Int sum3 = sum1 + sum2;

**C divide os operadores nos seguintes grupos:**

• Operadores aritméticos

• Operadores de atribuição

• Operadores de comparação

• Operadores lógicos

• Operadores bit a bit

**OPERADORES ARITIMETICOS**

*ADICAO:* ( + ) exemplo padrão!!

Int x = 5;

Int y = 3;

printf(“%d”, x + y)

//Ou

Int z = x + y

Printf(“%d”, z)

//Ou

Int z;

z = x + y

printf(“%d”, z)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*SUBTRACAO:* ( - )

Int x = 5;

Int y = 2;

Printf(“%d”, x - y)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*MULTIPLICACAO:* ( \* )

Int x = 5;

Int y = y;

Printf(“%d”, x \* y)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*DIVISÃO:* ( / )

Int x = 5;

Int y = y;

Printf(“%d”, x / y)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*MODULO:* ( % ) resto da divisão

Int x = 5;

Int y = 2;

Printf(“%d”, x % y); // que é igual a: 1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*INCREMENTACAO:* ( + + ) Adiciona mais 1

Int x = 5;

Printf(“%d”, ++x) // vai da 6

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*DECREMENTACAO:* ( -- ) Diminui 1

Int x = 5;

Printf(“%d”, --x) // vai dar 4

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Podem ser executados tanto dentro de uma variável, como dentro do print

**OBS:** A seguir, vamos ver que é necessário dividir as linhagens de operadores, e uma explicação mais detalhada sobre os operados bit a bit, logo são para coisa de muito baixo nível, que vamos intender já já.

Importante saber que a própria Microsoft, tem algumas dicas peculiares sobre C e sua biblioteca CRT.

**OPERADORES DE ATRIBUICAO**

Para entender alguns operadores de atribuição é necessário entender programação bit a bit, porem vamos lá.

=

Atribuímos alguma coisa ou valor a variável.

Int = 5;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

+=

Atribui a soma de um valor a uma variável que já tem um valor.

Int x = 5;

X += 3; // resultado 8 5 + 3

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

-=

Atribui uma diminuição ao valor de uma variável que já tem um valor

Int = 5;

x -= 3; // atribui menos 3 ao 5 resultando em 2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*=

Atribui uma multiplicação a variável

Int x = 5;

X \*= 3; // resultado 15 pois atruibui que 5 deve ser multiplicado por 3

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/=

Atribui uma divisão a variável

Float x = 5;

X /= 3; // 1.6667 atriu o resultado da divisão (nesse caso real) de 5 por 3

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

%=

Atribui ao resto da divisão de uma variável

int x = 5;

X %= 3; // Atribui o resto da divisão de 5 / 3 que no caso é 2.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

&=

Attribution AND bit a bit

Int x = 5;

X &= 3; // 1 não faço ideia

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|=

Atribuição OR inclusivo bit a bit

Int x = 5;

X |= 3; // 7 não faço idiea, mas tem aver com a fato de flipar bits

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

^=

Atribuição OR exclusivo bit a bit

Int x = 5;

X ^= 3; // 6

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>>=

Atribuição shift direita

Int x = 5;

X >>= 3; // 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<<=

Atribuição shift esquerda, não faço muita ideia de como isso vem a funcionar.

**OPERADORES DE COMPARACAO**

Esses operadores obviamente, são usados para comparar dois valores.

O valor de retorno de uma comparação pode ser:

Tru (1)

False(0)

Exemplo rápido:

Int x = 5 ;

Int y = 3;

Printf(%d, x > y); // retorna 1, pois 5 é maior que 3

Seria como dizer, “é verdadeiro que x é maior que y?”

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

==

Igual

X == y // X é igual a Y??

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

!=

Diferente

X != y // X é diferente de Y?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>

Maior que

X > y // X é mairo que Y?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<

Menor que

X < y // X é menor que Y?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>=

Maior ou igual a

X >= y // X é maior ou igual a Y?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

<=

Menor ou igual a

X <= y // X é menor ou igual a Y?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**OPERADORES LOGICOS**

Esses operadores são usados para determinar a lógica das variáveis

Exemplo:

Int x = 5;

Int y = 3;

Printf (“%d”, x > 3 && x < 10) // retorna 1, pois 5 é maior que 3 e menor que 10, já vimos tabelas verdades antes.

E é: So é verdadeiro se os dois forem verdade

OU: So é verdadeiro se um dos dois forem verdade

NÃO: Nega qualquer expressão, se é verdade vira mentira e se é mentira vira verdade.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

&&

AND

Retorna verdadeiro somente se as duas comparações forem verdadeiras.

X < 5 && x < 10

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

||

OR

Retorna verdadeiro se uma das duas comparações forem verdadeiras

X < 5 || x < 4

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

!

NOT  
Nega uma comparação total, ou seja, se algo sai de um valor verdadeiro de uma expressão comparativa, ele vai virar negativo, e verse versa

!(x < 5 && x < 10)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Logo, assim por diante, temos as expressões logicas e etc.

**TAMANHO DO OPERADOR**

Podemos ver o tamanho do espaço que uma variavel esta pegando dentro da memoria (em bits hexadecimal.)

Sizeof com a especificação %lu

Exemplo:

Int inteiro;

Float flutuante;

Double real;

Char carac;

Printf(“%lu\n”, sizeof(Inteiro));

Printf(“%lu\n”,sizeof(flutuante));

Printf(“%lu\n”,sizeof(real));

Printf(“%lu\n”,sizeof(carac));

// 4,4,8,1 bits

**Operador De Incremento**

++ incrementa +1

-- decrementa -1

X++ **Pos** incremento

++X **Pre** incremento

X-- **Pos** decremento

--X **Pre** decremento

int main()

{

    /\*

1. soma -> +=

        2) subtração -> -=

        3) multiplicação -> \*=

        4) divisão -> /=

        5) resto (módulo) -> %=

    \*/

    int i = 1;

    printf( “%i\n”, i );

    printf( “%i\n”, ++i );

    i++;

    printf( “%i\n”, i );

    system(“cls”);

    int i2 = 5;

    printf(“%i”, i2);

    printf(“%i”, --i2);

    i2--;

    printf(“%i”, i2 );

    /\*

1. Incremento\

        >Pré ou Pos

        2) Decremento/

        Pre -> o valor será incrementado/decrementado na instrução

                que a variável estiver contida

        Pos -> o valor será incrementado/decrementado na próxima

                instrução

    \*/

    int x = 0;

    x = x + 10;//incrementar qntas unidades desejarmos

    x += 10;//incrementar qntas unidades desejarmos

    ++x;// op. Incremento, só podemos incrementar

        // uma única unidade

    system( “cls “);

    printf(“%i”, x);

    return 0;

}

**IF , ELSE**

Para fazer estruturas condicionais usaremos:

If, else, else if

E depois veremos o Switch

**Exemplo de sintaxe:**

if (Condição)

{

(execução caso seja verdade);

}

// exemplo real

if (20 > 18)

{

Printf(“Ual em man.”);

}

**Obviamente também podemos testar variáveis.**

Int x = 4;

Int y = 7;

if x > y

{

Printf(“Muito baum”);

}

**Agora a sintaxe com else**

if (úmeros)

{

(bloco de comando);

} else

{

(bloco de comando);

}

**Exemplo:**

Const int time = 20;

If (time < 18)

{

Printf(“Bom dia”)

}

Else

{

Printf(“Boa noite”)

}

**Syntaxes else if**

If (condicao)

{

(bloco de úmeros);

}

Else if (condicao2)

{

(Bloco de úmeros);

}

Else

{

(Bloco de úmeros);

}

**Exemplo:**

Const int time = 22;

if (time < 10)

{

Printf(“Boa manhã”);

}

else if (time < 20)

{

Printf(“Bom dia”);

}

Else

{

Printf(“Boa noite”);

}

Por isso é bom você já ter feito o curso de logica em visualg que vai te dar as primeiras ideais para resolver certas coisas simples.

**Forma curta de se fazer um se**

Int time = 20;

(time < 18)? Prinf(“Bom dia”) : printf(“Boa tarde”);

É uma abreviação de

Int time = 20;

if (time < 18)

{

Printf(“Bom dia”);

}

Else

{

Printf(“Boa tarde”);

}

**SWITCH**

Interruptores, são uma forma mais fácil, para não fazer muitos ifs e elses, usados para quando você sabe com certeza os cenários possíveis.

**Exemplo:**

Int day = 4; // vai falar qual o nome do dia da semana de acordo com o numero.

Switch (day)

{

Case 1:

Printf(“Monday”);

Break; // é necessaries sempre o break após uma opcao.

Case 2:

Printf(“Tuesday”);

Break;

Case 3:

Printf(“Wednesday”);

Break;

}

AQUI AQUI AQUI <<<<<<<

Ou no último case, pode usar default para sinalizar um bloco a ser executado caso a variável não atenda nenhum dos casos.

Int day = 4;

Switch (day)

{

Case 1:

Printf(“Monday”);

Break;

Case 2:

Printf(“Tuesday”);

Break;

Case 3:

Printf(“Wednesday”);

Break;

Default:

Printf(“Não intende!”);

}

**LOOPINGS**

**WHILE:**   
Percorre um bloco de código desde que a condição especifica seja true.

Ex:

Int i = 0

While (i < 5)

{

Printf(“ %d\n”,int );

i++;

}

**LOOPING DO WHILE**

Executa o looping ou um código completo,uma vez, antes de entrar no while em ci e testar uma condição para iniciar o looping.

Ex:

Int = 0;

Do

{

Printf(“%d\n”, i);

I++;

}

While(1 < 5);

“Faz declaração, enquanto a resposta para a pergunta for verdadeira.”

Fazer

Saída “Qual é a sua idade?”

Insira a idade\_do usuário

Saída “Qual é a idade do seu amigo?”

Insira amigo\_idade

Saída “Juntas suas idades somam: “

Saída age\_user + friend\_age

Saída “Deseja tentar novamente? S ou n “

Loop\_response de entrada

Enquanto loop\_response == ‘y’

**LOOPING FOR**

Usado para quando sabemos exatamente quantas vezes queremos percorrer o looping.

Dentro do For:

Statement 1: Executa o código uma vez antes da execução do bloco, logo determina o valor inicial que o i valera

Statement 2: Define a condição para executar o bloco de código de quantas vezes o algoritmo percorre o looping

Statement 3: Executa o código todas as vezes após a execução do bloco de códigos ou incrementa um numero amais no i

Ex: Imprime 0 a 4:

Int i;

For (i = 0; i < 5, i++)

{

Printf(“%d\n”, i);

}

// Podemos brincar com as expressões do looping de acordo com a necessidade

For (I = 2; i+1 < 6; i++) ou

for (i = 1; i-1 < 6; i++)

que imprimiria de 1 até 6

ou

// imprime úmeros pares.

Int i;

for(i = 0; i <= 10; i = i + 2)

{

    printf(“%d\n”,i);

}

 return 0;

}

Não tem como colocar == em expressões de for.

**BREAK AND CONTINUE**

**Break:**

Vimos que no switch o break salta um bloco de comando caso a condição verdadeira

Em loopings, break pode ser usado para sair de um looping, quando por exemplo alguma condição for verdadeira.

int main()

{

    int i;

    for (i = 0; i < 10; i++)

    {

        if (i == 4)

        {

            break;

        }

        printf("%d\n", i);

    }

    printf("stop");

       // 0

       // 1

       // 2

       // 3

       // para

    return 0;

}

Lembrado que 0 conta como uma volta no looping. Ou seja 0 ate i < 10

Seria exatamente 10 passadas.

**CONTINUE:**

Interrompe uma ação do looping, caso ocorra uma condição especifica

E continua com a próxima interação do looping

int main()

{

    int i;

    for (i = 0; i < 10; i++)

    {

        if (i == 4)

        {

            continue;;

        }

        printf("%d\n", i);

    }

    printf("stop");

       // 0

       // 1

       // 2

       // 3

       // 4

       // 5

       // 6

       // 7

       // 8

       // 9

       // 10

       // para

    return 0;

}

**PODEMOS USAR AMBOS LOOPS WHILE:**

int main()

{

    int i = 0;

    while( i < 10) // imprimi de 0 a 9 = 10 passadas

    {

         if (i == 4)

         {

            break;         // ele para todo o processo no 3

         }

         printf("%d\n", i);

         i++;

    }

    // 0

    // 1

    // 2

    // 3

    return 0;

}

**UMA OPÇÃO PULANDO O 4.**

int main()

{

int i = 0;

while (i < 10)

{

   if(i == 4)

   {

    i++;          // quando chega ao quatro ele soma mais 1 e continua o processo.

    continue;

   }

   printf("%d\n", i);

   i++;

}

    return 0;

}

// 0,1,2,3,5,6,7,8,9.

**MATRIZES / ARRAYS**

Usado para armazenar vários valores em uma única variável, invés de declarar varias separas.

Um array só pode ter um tipo de dado.

//sintaxe array

    int Algo[100] = {}; // O numero de elementos sempre precisa ser

    // pre calculado, nesse caso as pocisoes de 0 a 100 não tem nada

    int algo2[100] = {12, 31, 52, 51, 522}; // com elementos ja postos

**Acessando os elementos:**

Basta consultar o índice da matriz, como já vi no curso de logica.

// lembrando que 0 é a primeira posicao.

    int algo[] = {25, 50, 75, 100};

    printf("%d", algo[0]);

    return 0;

**Alterando um elemento da matriz:**

// lembrando que 0 é a primeira posicao.

    int algo[] = {25, 50, 75, 100};

    // Alterando um elemento

    algo[0] = 80;

    printf("%d", algo[0]);

    return 0;

    // saida sera 80

// NÃO colocar um numero para que a memoria pre calcule o espaço é uma pratica ruim, sempre é bom ter ao menos uma constante (define) que se aplica a matrizes dando a elas um numero finito e evitadno deixar o trabalho de pre-calculo para memoria.

**Loop através de uma matriz:**

   //looping por matriz

   int teste[] = {25, 50, 75, 100};

   int i ;

   for (i = 0; i < 4; i++)

   {

        printf("Pocisão %i - valor - %i\n",i,teste[i]);

   }

   // printa todos os numeros da matriz

**Veremos durante os exercicos a gambiarra para fazer uma matriz vertical e horizontal.**

**Lembrando que o looping em sua exencia é vertical**

**0 = 25**

**1 = 50**

**2 = 75**

**3 = 100**

**STRINGS**

São majoritariamente usadas para armazenas textos de caracteres.

“Hello World” é uma string de caracteres.

C não possui um tipo “String” para criar facilemente uma variável de string. Invés disso, aqui usamos um array de char, para fazer um texto.

É exatamente isso que a função prinft faz.

Exemplo:

 char texto[30] = "Hello World!";

    printf("%s\n",texto); // imprime o texto

    printf("%c", texto[2]); //chama a letra "l"

**Modificando uma string:**

    char name[30] = {"Hello World"};

    name[2] = 'Z';

    name[3] = 'z';

    printf("%s", name);

// HeZzo World

**Outra maneira de criar strings:**

    char name[] = {'H', 'e', 'l', 'l', 'o', ' ', 'J', 'e', 'i', 'e', 'l', '!', '\0' };

    printf("%s", name);

    return 0;

Na verdade, na memoria em ci, é exatamente assim que uma string vai estar armazenada!

\0 é um caractere de terminação nulo, que indica para a memoria o momento de para de ler a string, lembrando que cada caractere faz parte da tabela UTF, até o espaço.

0 = ‘H’

1 =’e’

2 =’l’

3 = ‘l’

..... etc

11 = ‘\0’ // que o progama intende que é o fim da string. Porque? Porque o compilador ler apenas um linguição de bits.

Tudo funciona como uma maquina de turring, que vai para frente e para tras.

Diz ao compilador que esse é o fim da string.

Ambos têm os mesmos 13 caracteres. Logo darão 13 bytes

   char name[] = {'H', 'e', 'l', 'l', 'o', ' ', 'J', 'e', 'i', 'e', 'l', '!', '\0' };

    char name2[] = {"Hello Jeiel!"};

    printf("%lu - Bytes\n", sizeof(name));

    printf("%lu - Bytes", sizeof(name2));

     // 13 bytes os dois

    return 0;

// Isso é um baita exemplo de aplicação do sizeoff, para sabermos qual o tamanho real das variaveis que estamos criando e poder fazer um gerenciamento de memoria mais apropiado.

**CARACTERES ESPECIAIS:**

As vezes escrever os textos entre aspas das strings, pode gerar um erro

A solução seria o caractere de escape de barra invertida.

**Exemplo:**

Char txt[ ] = {“Eles eram chamados de “vinkings” no norte europeu”};

Jeito certo

Char txt[ ] = {eles eram chamados de \” Vikings\” no norte europeu}

Outro exemplo

Char txt[ ] = {“It\’ alright.”}; // It’ alright.

**Caracteres de escape em ci:**

\’ , \” , \\

Barra primeiro, caractere depois!!

Sempre é a barra primeiro e o caractere depois.

\n new line

\t tab

\0 null

Alguns exemplos basicos de formatação!.

**COMANDOS DE ENTRADA DO USUARIO**

Usamos: Scanf() pra ler valores digitados

    int numb;

    printf("Digite seu numero: ");

    scanf("%d", &numb,"\n");

    printf("Seu numer é: %d", numb);

    return 0;

No geral, cuidado ao colocar esse \n dentro de um scanf.

Se for colocar, coloca depois do scanf em ci.

O Scanf sempre recebe o tipo de valor que será digitado nele e o & antes da variável, que é indicando para armazenar em algum canto da memória.

**ENDERECO DE MEMORIA**

Ao se criar uma variável em C, um endereço de memória é atribuído a ela.

**O endereço** é o local aonde a variável esta armazenada no computador.

Ao atribuirmos um valor a variável, o mesmo é armazenado neste endereço de memória.

Podemos descobrir aonde um resultado está armazenado, ao pedir para o compilador imprimar a variável, porem com “&” antes.

 int numb;

    printf("Digite seu numero: ");

    scanf("%d", &numb,"\n");

    printf("Seu numer e: %d", numb,"\n");

    printf("%p", &numb);

%p é uma formatação já vista, para nos devolver a posição em hexadecimal que aquele valor esta armazenado.

Digite seu numero: 23

Seu número e: 23

0x0061FF1C

Importante!

**0x = valor hexadecimal**

**0b= valor binário**

    int numb;

    printf("Digite seu numero: ");

    scanf("%d", &numb,"\n");

    printf("Seu numer e: %d\n", numb);

    printf("POSICAO: 0x%p", &numb);

É apenas uma gambiarra minha para exibir de forma mais nítida!

Digite seu número: 32

Seu número e: 32

POSICAO: 0x0061FF1C

**PONTEIROS**

&variável, são conhecidos como ponteiros

Ponteiros são variáveis que armazenam o endereço de memória de outras variáveis como seu valor.

Uma variável de ponteiros aponta para um tipo de dado (como int) do mesmo tipo e é criado com o operador: \*.

Assim o endereço da variável na qual você está trabalhando é atribuído ao ponteiro.

Comumente usado para uma progamação mais dinamica, quando você esta tentando economizar memoria e priorizar desempenho.

Vamos ver mais do raciocínio por imagens, e tem mais nas minhas anotações.

Exemplo:

Ao declarar “\*” depois do tipo de dado, voce informa ao progama que aquela variavel é um ponteiro, e a mesma vai esta armazenando o espaço (O ESPAÇO E NÃO O VALOR) da varaivel principal para qual voce vai estar apontando!

Porque isso pode economizar memoria? Simples

Suponha que voce crie uma varaivel float, que armazena um numero enorme com varias casas decimais! Para que voce possa usar mais vezes esse numero sem se procupar com gastar muito espaco, voce pode criar um ponteiro que vai armazenar somento o numero do espaco que se encontra a variavel float gigante.

Logo, ao ser invocado esse ponteiro, ele pode “Replicar” o valor da variavel float, apenas mostrando o valor para qual ela esta apontando, (valor presente dentro daquele espaço na memoria)

Logo, é um 2 em 1.

  int idade = 32;

    int\* ponteiro = &idade; //ponteiro recebe a posicao de idade

    printf("%d\n", idade);

    printf("%p\n", &idade); // exibi a posicao original da variavel idade

    printf("%p\n", ponteiro); // exibi a posicao do ponteiro.

    printf("%d", \*ponteiro); // exibe o valor apontado

32

0061FF18

0061FF18

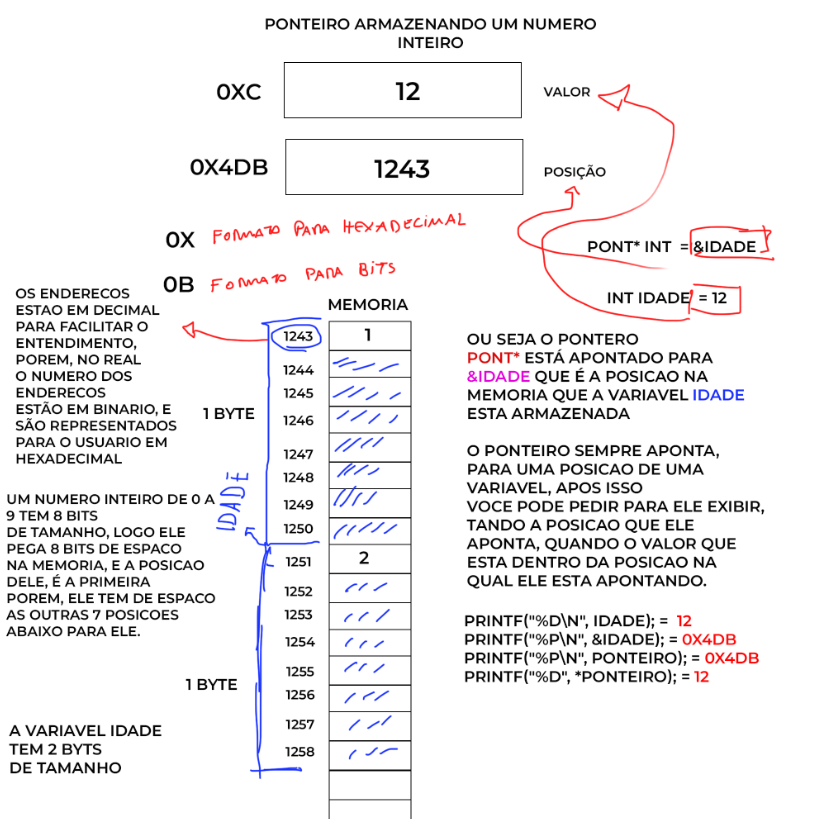
32

Enfim, ponteiros é um longo ponto, é um assunto cabeludo, pois ajuda na manipulação de dados da memória, aumentando o desempenho de grandes programas

Logo, se forem mal manuseados, podem alterar o valor de dados armazenados em outros endereços.

Pois, vc já deve ter em mente, que é possível fazer mil coisas com a posição ou valor armazenando em um ponteiro.

Se aprofundando um pouco mais sobre isso:



**FUNCOES**

É um bloco de codigo que so é executado quando chamado.

Voce pode passar dados, conhecidos comumente como pararmetros, para uma funcao, que são as tais variaveis de escopo.

É importante para a reutilizacao de codigo, e diminuacao do codigo principal, pois eu imagino que grandes projetos são compostos por bibliotecas propias que serão integradas no final ao progama main.

Esse tempo todos já estavamos usando funcoes, por exemplo

Int main(){} é uma funcao que executa os codifos dentro das chaves

E printf() é uma funcao usada para imprimir textos na tela.

**Criando uma funcao:**

void myFunc();

int main()

{

  myFunc();            // essa é uma boa pratica.

  return 0;

}

  void myFunc()

  {

    printf("Hello Worlds!");

  }

A funcao sera executada, em ordem, quantas vezes for declarada.

Da para fazer, umzilhão de coisas com isso, vai depender da sua imaginação

void Message();

int main()

{

int i;

  for (i = 0; i < 3 + 1; i++)

  {

     Message();

  }

  return 0;

}

void Message()

{

  printf("Isso e apenas um teste.\n");

}

**PASSANDO PARAMETROS E ARGUMENTOS PARA UMA FUNCAO:**

Informacoes podem ser passadas para uma funcao como parametros. Os parametros

Atuam como variaveis dentro da funcao(variaveis de escopo)

Os parametros são especificados dentro da funcao, entre parenteses.

Voce pode adicionar quantos parametros quiser, basta separalos com

Uma virgula.

Lembrando que tabem a como a funcao mudar globalmente

O valor de uma variavel passada como parametro.

Aqui vemos um exemplo passando um único parametro:

void nome(char name[31]);

int main()

{

  char nome\_um[20];

  char nome\_dois[20];

  char nome\_tres[20];

  printf("Qual o nome da sua equipe? ");

  scanf("%s", &nome\_um);

  scanf("%s", &nome\_dois);

  scanf("%s", &nome\_tres);

  nome(nome\_um);         // eu estaria passando de qualquer forma "nome"

  nome(nome\_dois);

  nome(nome\_tres);

  return 0;

}

void nome(char name[31])

{

  printf("Bem vindo: %s\n", name);

}

No exemplo acima, por ser uma void, (vazio) que não vai retornar nenhum numero, apenas prints (texto), não precisamos atribuir o resultado da funcao a uma varaivel e depois declara-la no printf especificando seu valor.

Aqui um exemplo passando varios parametros de uma vez:

void nome(char name[], int idade);

int main()

{

  char nome\_um[20];

  char nome\_dois[20];

  char nome\_tres[20];

  int idade\_um;

  int idade\_dois;

  int idade\_tres;

  printf("Qual o nome da sua equipe? ");

  scanf("%s", &nome\_um);

  scanf("%s", &nome\_dois);

  scanf("%s", &nome\_tres);

  printf("\n");

  printf("Qual a idade de ambos? ");

  scanf("%d", &idade\_um);

  scanf("%d", &idade\_dois);

  scanf("%d", &idade\_tres);

  nome(nome\_um, idade\_um);         // eu estaria passando de qualquer forma "nome" e um numero inteiro

  nome(nome\_dois, idade\_dois);

  nome(nome\_tres, idade\_tres);

  return 0;

}

void nome(char name[], int idade)

{

  printf("Bem vindo: %s  Sua idade e: %d\n", name, idade);

}

Lembrando que grandes projetos, voce faz suas propias bibliotecas, e puxa suas funcoes de um arquivo de extensao .h

Essa biblioteca deve esta na pasta librarys do compilador, e deve esta executada na pasta de Binarios, do mesmo compilador, para ela já ser precarregada.

**VALORES DE RETORNO:**

A palavra “VOID” indica que a funcao em ci não deve retornar nenhum valor

Agora, se voce quiser que a funcao retorne um tipo de valor, voce pode usar, tipos de dados como: int, float, etc.. e usar a palavra return no fim da funcao.

Exemplos

int valor(int x);

int func2(int x, int y);

int somas(int t, int z, int j);

int main()

{

  int numb = 32;

  int numb1 = 3;

  int numb2 = 5;

  int result = somas(numb1, numb2, numb);

  printf("O RESULTADO DISSO E: %d\n", valor(3));

  printf("OU O RESULTADO E: %d\n", valor(numb));

  printf("RETORNO DA SOMA DE DOIS VALORES: %d\n", func2(numb1, numb2));

  printf("MOSTRANDO A VARIAVEL QUE CONTEM O VALOR DA FUNCAO: %d", result);

  return 0;

}

int valor(int x)

{

  return 5 + x;

}

int func2(int x, int y)

{

     return x + y;

}

int somas(int t, int z, int j)

{

  return t + z + j;

}

Lembrando que o valor de uma funcao, pode ser atribuida a uma variavel ou a funcao pode ser invocada para executar direto no printf, bastando apenas usar o especificador do valor que a mesma vai retornar.

Enfim, muitas formas de usar.

**DECLARANDO DIVERSAS DE FUNCOES:**

No geral, funcoes cosistem em duas partes:

Declaracao: O nome da funcao, tipo de rretorno e parametro (se houver)

Definicao: O corpo da funcao (codigo a ser executado)

Para melhor otimizar o codigo e recomendado separar, a declareaca e a definicao da funcao.

Por mais estranho que pareça, frequentimente pode ser visto progamas em C que declarao a funcao antes de main,

E colocao as definicoes após ela. Pois é uma boa pratica

Porem isso é apenas quando voce já tem os valores da varaivel pre definidos, se não, não invente, pois ele vai pegar o valor aleatorios das variaveis que já existe, logo pela fato de C é uma linguagem que executa em ordem de cima para baixo.

int somas(int, int); // ou int x , int y

int main()

{

  int num1;

  int num2;

  int soma = somas(num1, num2); // nesse caso o parametro já esta passado

automaticamente.

  printf("Escreva um numero: \n");

  scanf("%d", &num1);

  printf("Escreva outro numero: \n");

  scanf("%d", &num2);

  printf("Resultado e: %d", soma);

  return 0;

}

int somas(int x, int y)

{

return x + y;

}

Desse jeito da certo.

**RECURSÃO:**

Mais conhecida como função recursiva.

É uma tecnica de fazer chamadas de funcoes entre ci. É usada para resolver problemas em partes menores e é uma especie de looping da deep web, eu particulamente nunca vi sendo muito usada.

Um looping executa um bloco de codigo, vericando acada vez se ele está no final da sequencia, e não existe esse fim sequencial para codigo recursivo .

Uma função rescursiva pode durar para sempre caso não tenha uma condição para acabar.

É simplesmente uma função chamando a ci mesma, adicionando a pilha de chamadas recursivas.

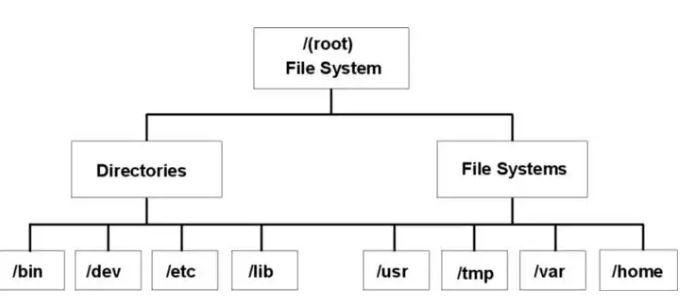
**Quando usar?**

Ela é feita para resolver problemas que podem ser divididos em problemas menores e repetitivos.

Boa para trabalhar com coisas que tenham muitas ramificações possiveis, e complexas para uma abordagem interativa.

Um exemplo disso seria pesquisar em um sistema de arquivos.

Comecnado a pesquisar pela pasta raiz em seguida por todos os arquivos das pastas dentro dela.

Entrando em todas as pastas dentro da pastas, etc. 

É pesquisar varios caminhos para varias ramificacoes, sem precisar incluir verificacoes e condicoes diferentes para cada possbilidade.

É bom para quem esta familiarizado com estruturas de dados, pois a recursão é uma boa maneira de percorrer arvores e grafos.

Ela aumenta o uso de memoria.

Essa tecnica acaba por oferecer uma maneira de quebrar problemas

Complicados em problemas simpels que são mais facies de resolver.

Exemplo: Adicionar dois numero é facil, mas adicionar um intervalor de numeros é mais complicado.

No exemplo asseguir, a recursão é utilizda para somar um intervalo de numeros, dividindo na simples tarefa de somar dois numeros:

Lembrando que é usado a bilbioteca Math.h

int sum(int k);

int main()

{

int result = sum(10);

printf("%d", result);

// ou printf(“%d”, sum(10));

return 0;

}

int sum(int k)

{

  if (k > 0)

  {

    return k + sum( k - 1);

  }

  else

  {

    return 0;

  }

}

É basicamente (nesse caso) um looping da xuxa, quando a funcao sum() é chamada

ela adiciona o parametro K a soma de todos os numeros menores que K

e retorna o resultado. Quando K se torna igual a 0, a funcao retorna apeanas 0

ao executar o progama faz os seguintes passos.

Uma funcao geralmente se divide em uma chamada recursiva e o caso base.

Aqui temos que ela é util para uma soma consecutiva de numeros, já vimos antes que fazer isso em um looping acaba por ser muito chato.

O desenvolvedor deve ter cuidado ao escrever um recursão, pois assim como loopings, é muito facil escrever uma funcao que nunca termina ou uma que usa quantidade excessivas de memória ou poder de processamento.

No geral, recursões tem uma boa abordagem, matematicamente muito elegante.

Pode-se basicamente fazer uma boa sequencia de fibbonaci com isso, muito mais pratica de como é feito com o looping.

Essa é a tal tecnica que colocar que muda o valor de uma variavel após entrar dentro de uma funcao, tornando o valor Global e alterado.

int fibonacci(int i) {

if(i == 0) {

return 0;

}

if(i == 1) {

return 1;

return fibonacci(i-1) + fibonacci(i-2);

}

int main() {

int i;

for (i = 0; i < 10; i++) {

printf("%d\t\n", fibonacci(i));

}

return 0;

}

// sla que poha de exemplo eh esse!

**TEMOS TAMBEM EXEMPLO DE BILBIOTECA DE FUNCOES QUE PUXAM:**

floor(x) - Arredonda para Baixo

ceil(x) - Arredonda para cima

sqtr(x) - Calcula a Raiz quadrada

pow(x, y) - potenciacao

sin(x) - seno

cos(x) - cosseno

tan(x) - tangente

log(x) - logaritmo natural

log10(x) - logaritmo base 10

**STRUCT**

É uma maneira de agrupar varias variaveis relacionadas em um só lugar, provavelmente foi o pensamento que originou a orientação a objetos.

Cada variavel na estrutura é conhecida como um **membro** da estrutura.

Diferente de um array, uma estrutura pode conter muitos tipos de dados diferentes (int, float, double, char, etc. )

**Criando uma estrutura:**

Usamos a plavra “sctruc” o nome e abre chaves para declarar seus menbros.

Exemplo:

Struct teste

{

Int meu\_numero;

Char Minha\_letra;

};

// deve-se sempre terminar uma estrutura de struct com ponto e virgula.

**Para acessar a estrutura, deve-se criar uma *varaivel* dela,**

Nesse caso, o nome dessa varia que vai armazenar o struct sera “s1”

Veja:

Struct teste

{

Int meu\_numero;

Char minha\_letra;

};

Int main()

{

Struct teste s1;

Return 0;

}

**Acessando os menbros de uma estrutura:**

Para isso, usaremos a sintaxe (.) ponto

Veja:

Struct teste

{

Int numero;

Char letra;

};

Int main()

{

Struct teste s1;

S1.numero = 12;

S1.letra = ‘c’;

Printf(“Minha letra é %c\n”, s1.letra);

Printf(“Meu numero é %d\n”, s1.numero);

Return 0;

}

Como notado, é possivel ir criando varias variaveis de estruturas com valores diferentes, usando apenas a mesma estrutura, so renomeando novas “variaveis de estrutura”

#include <stdio.h>

struct teste

{

int numero;

float flutu;

char letra;

};

int main()

{

struct teste s1;

struct teste s2;

s1.numero = 31;

s1.flutu = 31.441;

s1.letra = 'h';

s2.numero = 623;

s2.flutu = 3154.51;

s2.letra = 'd';

printf("Letra da estrutura 1: %c\n", s1.letra);

printf("Letra da estrutura 2: %c\n", s2.letra);

printf("Numero da estrutura 1: %d\n", s1.numero);

printf("Numero da estrutura 2: %d\n", s2.numero);

printf("Fluturante da estrutura 1: %0.2f\n", s1.flutu);

printf("Flutuante da estrutura 2: %0.2f\n", s2.flutu);

return 0;

}

**Resolvendo erros especificos.**

Um erro comum, é na utilização de strings, no caso mais especifico, jogar uma string para um menbro de uma struct.

Veja o exemplo:

#include <stdio.h>

struct teste

{

char nome[];

};

int main()

{

  struct teste s1;

  nome.s1 = 'Jeiel Souza'; // isso consequentimente ocasiona a poha de um erro. por alguma frescura do C

  return 0;

}

Logo, para solucionarmos isso, podemos usar uma funcao chamada. Strcpy() e atribuir o valor de s1.nome a ela.

Lembrando que essa funcao é única e exclusiva da biblioteca string.h

#include <stdio.h>

#include <string.h>

#define SPC 100

struct teste

{

  char nome[SPC];

};

int main()

{

  struct teste s1;

  strcpy(s1.nome, "Jeiel Souza");

  printf("Seu nome e: %s", s1.nome);

  return 0;

}

Agora sim o codigo vai funcionar.

Note que até o C precisa de bibliotecas para funcionar, quem dira o maldito assembly.

**Atribuindo varios valores em uma única linha**

#include <stdio.h>

#include<string.h>

/\*Declarando varios dados em uma unica linhas. \*/

struct vars

{

  char letra;

  char nome[30];

  int idade;

  float saldo;

};

int main()

{

  struct vars s1 = {'B', "Jeiel Souza", 32, 588.21}; // da um erro

  printf("Values: %c, %s, %d, %0.2f", s1.letra, s1.nome, s1.idade, s1.saldo);

  return 0;

}

**Saida:** **Values: B, Jeiel Souza, 32, 588.21**

Nesse caso em especifico, não é necessarios a cambiarra para declarar uma string.

**Atribuindo uma estrutura a outras. (Copiar estruturas)**

#include <stdio.h>

#include<string.h>

/\*Declarando varios dados em uma unica linhas. \*/

struct vars

{

  char letra;

  char nome[30];

  int idade;

  float saldo;

};

int main()

{

  struct vars s1 = {'B', "Jeiel Souza", 32, 588.21}; // da um erro

  struct vars s2;

  s2 = s1;

  printf("Values: %c, %s, %d, %0.2f\n", s1.letra, s1.nome, s1.idade, s1.saldo);

  printf("Values: %c, %s, %d, %0.2f", s2.letra, s2.nome, s2.idade, s2.saldo);

  return 0;

}

Simplesmente criamos uma nova estrutura e embaixo dela atribuimos a estrutura passada.

**Modificando valores já declarados:**

/\*Modificando valores já declarados. \*/

#include <stdio.h>

#include <string.h>

struct trocas

{

   int numero;

   char letra;

   char nome[31];

};

int main()

{

struct trocas s1 = {12, 'g', "Jeiel Souza"};

printf("Dados em sequencia: %d, %c, %s\n", s1.numero, s1.letra, s1.nome);

s1.numero = 14;

s1.letra = 'z';

strcpy(s1.nome, "Lia nagela");

printf("Novos dados: %d, %c, %s", s1.numero, s1.letra, s1.nome);

return 0;

}

Facilmente podemos alterar de algumas formas, os menbros de uma struct.

Pode ser feito tabém quando voce copia valores de uma estrutura para outra, e precisa modificalas depois.

**ENUMS << HERE**

Parte final do basico sobre C, olha que perfeito, finalmente voce deixou de procastinar e aprendeu o basico de uma nova linguagem, parabens e lembre-se, não aprendeu nem 1% de tudo.

Enuns é um tipo de dado especial que representa um grupo de constantes, que são valores imutaveis.

Enum é uma abreviação de “Enumerações”, que significa lista especifica.

Sintaxe:

Lembrando que nove de constante se coloca sempre com letra maiuscula.

enum Level

{

  LOW,

  MEDIUM,

  HIGHT

};

**Acessando o ENUM:**

É necessario criar a variavel dele

int main()

{

  enum Level myVar;

  return 0;

}

Agora sim podemos atribuir valores a ele .

Por padrão os valores de cima para baixo são (0, 1, 2, etc... )

Vamos atribuir um dos leveis para a variavel “Myvar”

int main()

{

  enum Level myVar = MEDIUM;

  return 0;

}

Por padrão, se pedimos para ela ser imprimida, ela retornara 1.

Nesse exemplo, por padrão seria assim:

LOW: 0

MEDIUM: 1

HIGHT: 2

#include <stdio.h>

enum Level

{

  LOW,

  MEDIUM,

  HIGHT

};

int main()

{

  enum Level myVar = MEDIUM;

  printf("%d", myVar);

  return 0;

}

Retorna: 1

**Mudando valores:**

enum level

{

  LOW = 10,

  MEDIUM = 34,

  RIGHT = 51,

};

int main()

{

  enum level myvar = RIGHT;

  printf("%d\n", myvar);

  myvar = MEDIUM;

  printf("%d\n", myvar);

  myvar = LOW;

  printf("%d", myvar);

  return 0;

}

Modificamos normamente, como modificariamos se fosse uma variavel.

**Ainda falta ver operadores bit a bit**

Progamas em C, no seu geral é composto por variaias bibliotecas, algo.h, que no fim são referenciadas no progama.c principal, aonde contem a int main.

Bibliotecas podem ser formadas por outras bibliotecas e etc, a imaginação é o limite.

Ponto de parada:

<https://www.w3schools.com/c/c_structs.php>

## Sintaxe mais simples

## While DSL upstream [ITU 1999], and 55 Mbps downstream and 15 Mbps upstream [ITU 2006]. Because the downstream and upstream rates are different, the access is said to be asymmetric. The actual downstream and upstream transmission rates achieved may be less than the rates noted above, as the DSL provider may purposefully limit a residential rate when tiered service (different rates, available at different prices) are offered. The maximum rate is also limited by the distance between the home and the CO, the gauge of the twisted-pair line and the degree of electrical interference. Engineers have expressly designed DSL for short distances between the home and the CO; generally, if the residence is not located within 5 to 10 miles of the CO, the residence must resort to an alternative form of Interne

modem is typically an external device and connects to the home PC through an Ethernet port. (We will discuss Ethernet in great detail in Chapter 6.) At the cable head end, the cable modem termination system (CMTS) serves a similar function as the DSL network’s DSLAM—turning the analog signal sent from the cable modems in many downstream homes back into digital format. Cable modems divide the HFC network into two channels, a downstream and an upstream channel. As with DSL, access is typically asymmetric, with the downstream channel typically allocated a higher transmission rate than the upstream channel. The DOCSIS 2.0 standard defines downstream rates up to 42.8 Mbps and upstream rates of up to 30.7 Mbps. As in the case of DSL networks, the maximum achievable rate may not be realized due to lower contracted data rates or media impairments. One imp

ortant characteristic of cable Internet access is that it is a shared broadcast medium. In particular, every packet sent by the head end travels downstream on every link to every home and every packet sent by a home travels on the upstream channel to the head end. For this reason, if several users are simultaneously downloading a video file on the downstream channel, the actual rate at which each user receives its video file will be significantly lower than the aggregate cable downstream rate. On the other hand, if there are only a few active users and they are all Web surfing, then each of the users may actually receive Web pages at the full cable downstream rate, because the users will rarely request a Web page at exactly the same time. Because the upstream channel is also shared, a distributed multiple access protocol is needed to coordinate transmissions and avoid

collisions. (We’ll discuss this collision issue in some detail in **Chapter 6.)** Although DSL and cable networks currently represent more than 85 percent of residential broadband access in the United States, an up-and-coming technology that provides even higher speeds is fiber to the home (FTTH) [FTTH Council 2016]. As the name suggests, the FTTH concept is simple—provide an optical fiber path from the CO directly to the home. Many countries today—including the UAE, South Korea, Hong Kong, Japan, Singapore, Taiwan, Lithuania, and Sweden—now have household penetration rates exceeding 30% [FTTH Council 2016]. There are several competing technologies for optical distribution from the CO to the homes. The simplest optical distribution network is called direct fiber, with one fiber leaving the CO for each home. More commonly, each fiber leave

ng the central office is actually shared by many homes; it is not until the fiber gets relatively close to the homes that it is split into individual customer-specific fibers. There are two competing optical-distribution network architectures that perform this splitting: active optical networks (AONs) and passive optical networks (PONs). AON is essentially switched Ethernet, which is discussed in Chapter 6. Here, we briefly discuss PON, which is used in Verizon’s FIOS service. Figure 1.7 shows FTTH using the PON distribution architecture. Each home has an optical network terminator (ONT), which is connected by dedicated optical fiber to a neighborhood splitter. The splitter combines a number of homes (typically les