Introdução à linguagem C

Diego Raphael Amancio

Baseado no material do Prof. Thiago A. S. Pardo e do Prof. André Backes

Sintaxe e Comandos básicos de C

- A sintaxe são regras detalhadas para cada construção válida na linguagem C.

- **Tipos:** definem as propriedades dos dados manipulados em um programa.
- **Declarações:** expressam as partes do programa, podendo dar significado a um identificador, alocar memória, definir conteúdo inicial, definir funções.
- **Funções**: especificam as ações que um programa executa quando roda.

Funções e comentários

- As funções são as entidades operacionais básicas dos programas em C.
  - Exemplo: *printf*() e *scanf*()
- O programador também pode definir novas funções
- Os programas incluem também bibliotecas que permitem o uso de diversas funções que não estão definidas na linguagem
- Todo programa C inicia sua execução chamando a função main(), sendo obrigatória a sua declaração no programa principal.
- Comentários são colocados entre /**/ ou // (uma linha)

Exemplo de programa simples

```
#include <stdio.h>

//Bloco principal do programa
int main() {
   printf("Oi, Mundo.\n");
   return 0;
}
```

A linha include indica a biblioteca que o programa vai usar. Neste caso “stdio.h” é uma biblioteca que permite utilizar a função printf

Principais funções de I/O

- printf("expressão de controle", argumentos) ;
  - função de I/O, que permite escrever no dispositivo padrão (tela). A expressão de controle pode conter caracteres que serão exibidos e os códigos de formatação que indicam o formato em que os argumentos devem ser impressos. Cada argumento deve ser separado por vírgula.
- scanf("expressão de controle", argumentos);
  - é uma função de I/O que nos permite ler dados formatados da entrada padrão (teclado). A lista de argumentos deve consistir nos endereços das variáveis.

Printf- Comandos básicos

\n nova linha
\t tabulação
\" aspas
%c catactere simples
%d inteiro
%f ponto flutuante
%s cadeia de caracteres (string)
%u inteiro sem sinal

Exemplo:

```
printf( “Este é o numero dois: %d“ , 2 );

printf("%s está a %d milhões de milhas\ndo sol", "Vênus", 67);
```

Scanf-Comandos básicos

- Usa um operador para tipos básicos chamado operador de endereço e referenciado pelo símbolo "&“, que retorna o endereço do operando.

  - int x = 10; //Armazena o valor 10 em x
  - &x //retorna o endereço da memória onde x está

Exemplo-Scanf

```
#include<stdio.h>
int main() {
    int num;
    printf("Digite um número: ");
    scanf( "%d", &num);
    return 0;
}
```

Sintaxe e comandos básicos de C

- Cada instrução encerra com ; (ponto e vírgula)
- Letras minúsculas são diferentes de maiúsculas.
  - Palavra != palavra != PaLaVRA (case sensitive)
- As palavras reservadas da linguagem estão sempre em minúsculas.

Palavras reservadas (ANSI)

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Sintaxe e comandos básicos de C

- As inclusões de bibliotecas devem estar acima de todas as outras declarações do programa.
- No início dos blocos declaramos todas as variáveis que vamos utilizar
- O comando de declaração de variáveis tem a sintaxe:

  Tipo nome1[=valor][,nome2[=valor]]...[,nomeN[=valor]

Exemplo

```
#include <stdio.h>

   void main () {
      int a = 10;
      printf (“Este é o valor de a : %d\n", a );
   }
```

Atribuição e Inicialização de variáveis

- Operador de atribuição (=)
  - int a; a = 10;
  - int a = 10; //inicializado na declaração
- Inicializar significa atribuir à mesma um valor inicial válido.
  - Ao se declarar a variável, a posição de memória da mesma contém um valor aleatório.

Onde declarar variáveis?

- Em três lugares, basicamente:
  - Dentro de funções: variáveis locais.
  - Na definição de parâmetros de funções: parâmetros formais.
  - Fora de todas as funções: variáveis globais.

Escopo das variáveis

- Escopo define onde e quando uma variável pode ser usada em um programa.
- **Variável global** - tem escopo em todo o programa:

```
#include <stdio.h>
int i = 0;                  /* variável global */
                            /* visível em todo arquivo */

void incr_i() { i++;}
...
void main() { incr_i(); printf("%d", i);}
```

Exercício 1

- Escreva um programa em C que mostre quantas horas e minutos correspondem 34706 segundos, este último valor deve ser visível globalmente .

Exercicio 1

```
#include <stdio.h>
int segundos = 34706;

void main ()
    {
       int minutos = segundos / 60;
       int horas = minutos / 60;
       printf ("34706 segundos correspondem a %d\n horas",
    horas );
       printf ( "e %d \n minutos", minutos);
    }
```

Identificadores

- São nomes usados para se fazer referência a variáveis, funções, rótulos e vários outros objetos definidos pelo usuário.
  - O primeiro caracter deve ser uma letra ou um sublinhado.
- Os 32 primeiros caracteres de um identificador são significativos.
- É case sensitive, ou seja, as letras maiúsculas diferem das minúsculas.

```
int x; /* é diferente de int X;*/
```

Variáveis e Constantes

- Constante: valor fixo que não pode ser modificado pelo programa. Exemplo:
  - Valores inteiros: 123, 1, 1000, -23
  - Strings: “abcd”, “isto é uma string!”, “Av. São Carlos, 2350”
  - Reais: 123.45F, 3.1415e-10F
- const é utilizada para determinar constantes:
  - const char LETRA_B =‘B’;
- Variável: Podem ser modificadas durante a execução do programa

Variáveis

- Em um programa C estão associadas a **posições de memória** que armazenam informações.
- Toda variável deve estar associada a um identificador.
- Palavras-chave de C não podem ser utilizadas como identificador (evita ambiguidade)
  - Ex.: int, for, while, etc...
- C é case-sensitive:
  - contador ≠ Contador ≠ CONTADOR ≠ cOntaDor

Tipos de dados básicos

- Define a **quantidade de memória** que deve ser **reservada para uma variável** e como os bits devem ser interpretados.
- O *tipo* de uma variável define os valores que ela pode assumir e as operações que podem ser realizadas com ela.
- Ex:
  - variáveis tipo *int* recebem apenas valores inteiros.
  - variáveis tipo *float* armazenam apenas valores reais.

Tipos de dados básicos

- Os tipos de dados básicos, em C, são 5:
  - Caracter: **char** (‘a’, ‘1’, ‘+’, ‘$’, ...)
  - Inteiro: **int** (-1, 1, 0, ...)
  - Real: **float** (25.9, -2.8, ...)
  - Real de precisão dupla: **double** (25.9, -2.8, ...)
  - Sem valor: **void**
- Todos os outros tipos são derivados desses tipos.

Modificadores de Tipos

- Modificadores alteram algumas características dos tipos básicos para adequá-los a necessidades específicas.
- Modificadores:
  - **signed**: indica número com sinal (inteiros e caracteres).
  - **unsigned**: número apenas positivo (inteiros e caracteres).
  - **long**: aumenta abrangência (inteiros e reais).
  - **short**: reduz a abrangência (inteiros).

Tipos fundamentais agrupados por funcionalidade

- **Tipos inteiros**: char, signed char, unsigned char, short, int, long, unsigned short, unsigned long.

- **Tipos de ponto flutuante**: float, double, long double.

Exercicio 2

- Calcule à area e perímetro de um retângulo com dados submetidos pelo usuário e imprima separadamente os resultados.

Exercicio 2

```
#include <stdio.h>

void main ()
{
  int base, altura, perimetro;

  printf ("Digite o valor da  base \n");
  scanf("%d", &base);
  printf ("Digite o valor da altura\n");
  scanf("%d", &altura);
  perimetro = (2 * base) + (2 * altura);
  printf ("Área: %d\n", base * altura);
  printf ("Perímetro: %d\n", perimetro);

 }
```

Arrays ou Vetores

- Um vetor é uma coleção de **variáveis do mesmo tipo** referenciadas por um nome comum.
- Uma determinada variável do vetor é chamada de **elemento** do vetor.
- Os elementos de um vetor podem ser acessados, individualmente, por meio de índices

Arrays ou Vetores em C

- Os elementos de um vetor ocupam **posições contíguas** na memória.
- Um vetor é considerado uma matriz - ou array - unidimensional.
- Em C, vetores (matrizes também) e ponteiros são assuntos relacionados.

Arrays ou Vetores

- Forma geral da declaração:
  - **tipo A[*expressão*]**
    - **tipo** é um tipo válido em C.
    - A é um identificador.
    - *expressão* é qualquer expressão válida em C que retorne um **valor inteiro positivo**.
  - Exemplos

    float salario[100];

    int numeros[15];

    double distancia[a+b]; //com a+b >= 0

Inicialização Vetores

- float F[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
- int A[100] = {1};
  - Vetor A = [1 0 0 0 ... 0].
- int A[100] = {1, 2};
  - A[0] recebe 1, A[1] recebe 2 e o restante recebe 0.
- int A[] = {2, 3, 4};
  - Equivale a: int A[3] = {2, 3, 4};
- **A primeira posição de um vetor é a posição 0**;

Exemplo

int c[10], i;

Primeira posição = ?

Exemplo

int c[10], i;

Primeira posição = 0 → c[0]

Ultima posição = ?

Exemplo

int c[10], i;

Primeira posição = 0 → c[0]

Ultima posição = 9 → c[9]

Exercício

- Implemente em C um programa que leia 100 números reais e imprima o desvio padrão.

Exercício

- Implemente em C um programa que leia 100 números reais e imprima o desvio padrão.

- Leitura da i-ésima posição do vetor

  scanf(“%f”, &v[i]);

  scanf(“%f”, v + i);

Strings

- Não existe um tipo String em C.
- Strings em C são uma array do tipo char que termina com ‘\0’.
- Para literais string, o próprio compilador coloca ‘\0’.

Exemplo de String

```
#include <stdio.h>

void main() {
    char re[4] = “aim";
    //char re[4] = {‘a’,‘i’,‘m’,‘\0’}
    printf ( "%s", re );
}
```

Declaração: duas formas

```c
#include <stdio.h>

int main(int, char **)
{
    char ola[] = "ola";

    printf(ola);

    return 0;
}
```

Em forma de array

```c
#include <stdio.h>

int main(int, char **)
{
    const char *ola = "ola";

    printf(ola);

    return 0;
}
```

Em forma de ponteiro

Leitura de uma String

- scanf: não lê espaços em branco
  - scanf(“%s”, stringName): não é necessário o operador &
- gets: lê espaços em branco

```
#include <stdio.h>
void main(){
    char re[80];
    printf ("Digite o seu nome: ");
    gets(re);
    printf ("Oi %s\n", re);
}
```

Contando caracteres

```c
#include <stdio.h>

int contaChar(const char *str)
{
    int i = 0;

    for(;str[i] != 0; ++i);

    return i;
}

int main(int, char **)
{
    char ola[] = "ola";

    printf("A string %s possui %d caracteres\n", ola, contaChar(ola));

    return 0;
}
```

Biblioteca <string.h>

- Biblioteca que contém as funções para mexer com strings
  - Ex: strlen → a função retorna um valor inteiro com o número de caracteres da String

Referência: www.cplusplus.com/reference/cstring

Exemplo

```
#include <stdio.h>
#include <string.h>
main() {
    char re[80];
    printf ("Digite a palavra: ");
    scanf ("%s", re);
    int size = strlen(re);
    printf ( "Esta palavra tem %d caracteres.\n", size );
}
```

Comparando strings (igualdade)

```c
#include <stdio.h>

int main(int, char **)
{
    char ola[] = "ola";
    char ola2[] = "ola";

    if(ola == ola2)
        printf("Iguais");
    else
       printf("Nao sao iguais");

    return 0;
}
```

Saída ?

Comparando strings (igualdade)

```c
#include <stdio.h>

int main(int, char **)
{
    char ola[] = "ola";
    char ola2[] = "ola";

    if(ola == ola2)
        printf("Iguais");
    else
       printf("Nao sao iguais");

    return 0;
}
```

Saída ?

Não são iguais

Por quê?

Comparando strings (igualdade)

```
int saoIguais(const char *s1, const char *s2)
{
    int i;
    for ( i = 0; s1[i] == s2[i]; ++i )
    {
        if( s1[i] == '\0' )
            return 1;
    }
    return 0;
}
```

Comparando strings (igualdade)

int strcmp (const char * str1, const char * str2);

- Compara as strings str1 e str2
- Retorna
  - zero, se são iguais
  - <0, se str1 < str2
  - >0, se str2 > str1

Comparando duas strings

strcmp é equivalente a código abaixo?

```
char *a, *b;
a = "abacate";
b = "uva";
if (a < b)
   printf( "%s vem antes de %s no dicionário", a, b);
else
   printf( "%s vem depois de %s no dicionário", a, b);
```

Comparando duas strings

strcmp é equivalente a código abaixo?

```
char *a, *b;
a = "abacate";
b = "uva";
if (a < b)
   printf( "%s vem antes de %s no dicionário", a, b);
else
   printf( "%s vem depois de %s no dicionário", a, b);
```

Comparação de ponteiros

Copiando strings

```
1  int main(int, char **)
2  {
3      char str1[] = "abc";
4      char str2[10];
5
6      str2 = str1;
7
8      return 0;
9  }
```

Copiando strings

```
1  int main(int, char **)
2  {
3      char str1[] = "abc";
4      char str2[10];
5
6      str2 = str1;
7
8      return 0;
9  }
```

Erro de compilação

Copiando strings

```
void copia(char destino[], char origem[])
{
    int i;
    int size = strlen(origem);

    for ( i=0 ; i < size; i++)
        destino[i] = origem[i];

    destino[i] = '\0';
}
```

Para copiar o conteúdo de uma string em outra

- Usa se a função: “para” é a string onde vai se copiar a informação nova e “de” é a string antiga

  strcpy(para, de);

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

main() {
    char str[80];
    strcpy (str, "Alo");
    printf ("%s", str);
}
```

Funções de conversão

- De string para double
  - Exemplo: “51.2341” para 51.2341
- strtod
  - double strtod(const char *toConvert, char **endPtr)
    - toConvert: string a ser convertida
    - endPtr: ponteiro para a string restante após a conversão

strtod

Saída: 51.2
% are omitted

```c
int main()
{

    //String a ser convertida
    const char *string = "51.2% are omitted";

    double d;
    char *stringPtr;

    d = strtod ( string, &stringPtr);

    printf( "%f\n%s\n", d, stringPtr);

}
```

Exercício

- Escreva uma função que transforme uma string (que contenha apenas dígitos) em um número inteiro
  - int strtoi(char *string)

- Escreva um programa que leia duas strings, que as junte numa string só e imprima o tamanho e o conteúdo da string final.

Outras funções de conversão

- strtoi(const char *str, char **endPtr, int base)
  - Converte str para um long int
  - str: string a ser convertida
  - endPtr: string restante não convertida
  - base: base da conversão

Outras funções de conversão

- strtoul(const char *str, char **endPtr, int base)
  - Converte str para um unsigned long int
  - str: string a ser convertida
  - endPtr: string restante não convertida
  - base: base da conversão

sprintf

- Funciona de forma análoga ao printf
  - Escreve em string e não na tela

- Exemplo:

```
//Cria a string str como “o número é 10”
int num = 10
sprintf ( str, “o numéro é%d”, num );
```

sscanf

- Análoga a função scanf
- Exemplo

```
char s[] = “312 3.14159”;
int x; double y;

sscanf( s, “%d%f”, &x, &y);
```

Exercício

- Escreve um trecho de código que substitua uma substring por outra
  - **strsubst**(string_in,searchStr,replaceStr)
  - Exemplos

strsubst(“communication”, “tion”, “cao”) retorna “communicacao”
strsubst(“communication”, “unication”, “”) retorna “comm”

Expressões

- Em C, expressões são compostas por:
  - Operadores: +, -, %, ...
  - Constantes e variáveis.
- Precedência: ( )
- Exemplos

  x;

  14;

  x + y;

  (x + y)*z + w – v;

Expressões

- Expressões podem aparecer em diversos pontos de um programa:
  - comandos /* x = y; */
  - parâmetros de funções /* sqrt (x + y); */
  - condições de teste /* if (x == y) */

Expressões

- Expressões retornam um valor:

```
x = 5 + 4 /* retorna 9 */
```

  - esta expressão retorna 9 como resultado da expressão e atribui 9 a x

```
((x = 5 + 4) == 9) /* retorna true */
```

  - na expressão acima, além de atribuir 9 a *x*, o valor retornado é utilizado em uma comparação

- Expressões em C seguem, geralmente, as regras da álgebra.

Operadores Aritméticos

- Operadores unários
- Operadores binários

Operadores Unários

+ : mais unário (positivo)	/* + x; */
- : menos unário (negativo)	/* - x; */
! : NOT ou negação lógica	/* ! x; */
&: endereço	/* &x; */
*: conteúdo (ponteiros)	/* (*x); */
++: pré ou pós incremento	/* ++x ou x++ */
-- : pré ou pós decremento	/* -- x ou x -- */

Operador ++

- Incremento em variáveis
  - ++ pode ser usado de modo pré ou pós-fixado
  - Exemplo:

```
int x =1, y =2;
x++; /* equivale a x = x + 1*/
++y; /* equivale a y = y + 1*/
```

- Não pode ser aplicado a constantes nem a expressões.

Operador ++

- A instrução ++x
  1. Executa o incremento
  2. Depois retorna x
- A instrução x++
  1. Usa o valor de x
  2. Depois incrementa.

Exercício 4

Diga quais são os valores das variáveis y e x em cada momento de execução do seguinte programa

```
int main (void)  {
    int x = 10;  int y = 0;
    y = ++x;
    printf (“%d %d”, x, y);
    y = 0; x = 10;
    y = x++;
    printf (“%d %d”, x, y);
    return(0);
}
```

Exercício 4

Diga quais são os valores das variáveis y e x em cada momento de execução do seguinte programa

```
int main (void)  {
   int x = 10;  int y = 0;
   y = ++x;
   printf (“%d %d”, x, y);
   y = 0; x = 10;
   y = x++;
   printf (“%d %d”, x, y);
   return(0);
}
```

X = 10
Y = 0

Exercício 4

Diga quais são os valores das variáveis y e x em cada momento de execução do seguinte programa

```
int main (void)  {
   int x = 10;  int y = 0;
   y = ++x;
   printf (“%d %d”, x, y);
   y = 0; x = 10;
   y = x++;
   printf (“%d %d”, x, y);
   return(0);
}
```

X = 11
Y = 0

Exercício 4

Diga quais são os valores das variáveis y e x em cada momento de execução do seguinte programa

```
int main (void)  {
   int x = 10;  int y = 0;
   y = ++x;
   printf (“%d %d”, x, y);
   y = 0; x = 10;
   y = x++;
   printf (“%d %d”, x, y);
   return(0);
}
```

X = 11
Y = 11

Exercício 4

Diga quais são os valores das variáveis y e x em cada momento de execução do seguinte programa

```
int main (void)  {
    int x = 10;  int y = 0;
    y = ++x;
    printf (“%d %d”, x, y);
    y = 0; x = 10;
    y = x++;
    printf (“%d %d”, x, y);
    return(0);
}
```

X = 10
Y = 0

Exercício 4

Diga quais são os valores das variáveis y e x em cada momento de execução do seguinte programa

```
int main (void)  {
    int x = 10;  int y = 0;
    y = ++x;
    printf (“%d %d”, x, y);
    y = 0; x = 10;
    y = x++;
    printf (“%d %d”, x, y);
    return(0);
}
```

X = 10
Y = 10

Exercício 4

Diga quais são os valores das variáveis y e x em cada momento de execução do seguinte programa

```
int main (void)  {
    int x = 10;  int y = 0;
    y = ++x;
    printf (“%d %d”, x, y);
    y = 0; x = 10;
    y = x++;
    printf (“%d %d”, x, y);
    return(0);
}
```

X = 11
Y = 10

Operador --

- Decremento
  - O operador -- decrementa seu operando de uma unidade.
  - Funciona de modo similar ao operador ++.

Exercício

- O quê será impresso?

```
int a, b = 0, c = 0;
a = ++b + ++c;
printf(“%d %d %d\n”, a, b, c );
a = b++ + ++c;
printf(“%d %d %d\n”, a, b, c );
```

Operadores binários

- São eles:
  - +: adição de dois números /* x + y */
  - - : subtração de dois números /* x - y */
  - * : multiplicação de dois números /* x * y */
  - / : quociente de dois números /* x / y */
  - %: resto da divisão inteira /* x % y */
    - Só aplicável a operandos inteiros.

Exercício

- Faça um programa que leia dois números, calcule e imprima: a parte inteira do resultado e a parte fracionária do resultado.

Exercício

```
#include<stdio.h>
void main(){
int a, b, p_int, p_frac;
 printf(“Escreva dois numeros:\n”);
 scanf(“%d\n%d”,&a,&b);
 p_frac=(a%b)/b;
 p_int= a/b – p_frac;
Printf(“Parte inteira: %d\n Parte Fraccionaria:
%d\n”,p_int, p_frac);
}
```

Operadores de Atribuição

= : atribui	x = y;
+= : soma e atribui	x += y; <=> x = x + y;
-= : subtrai e atribui	x -= y; <=> x = x - y;
*= : multiplica e atribui	x *= y; <=> x = x * y;
/= : divide e atribui	x /= y; <=> x = x / y;
%= : divide e atribui resto	x %= y; <=> x =x%y;

Exercício

1- Diga o resultado das variáveis x, y e z depois da seguinte seqüência de operações:

```
int x, y, z;
x=y=10;
z=++x;
x=-x;
y++;
x=x+y-(z--);
```

a) x = 11, y = 11, z = 11
b) x = -11, y = 11, z = 10
c) x = -10, y = 11, z = 10
d) x = -10, y = 10, z = 10
e) Nenhuma das opções anteriores

Operadores Relacionais

- Aplicados a variáveis que obedeçam a uma relação de ordem, retornam 1 (true) ou 0 (false)

Operador	Relação
>	Maior do que
>=	Maior ou igual a
<	Menor do que
<=	Menor ou igual a
==	Igual a
!=	Diferente de

Operadores Lógicos

- Operam com valores lógicos e retornam um valor lógico verdadeiro (1) ou falso (0)

Operador	Função	Exemplo
&&	AND (E)	(c >='0' && c<='9')
||	OR (OU)	(a=='F' || b!=32)
!	NOT (NÃO)	(!var)

Exercício 8

Considerando as variáveis fornecidas, calcule o resultado das expressões.

int a = 5, b=4; float f = 2.0; char c =‘A’;

a) a++ + c * b

b) (((3*2.0)-b*10) && a) || (f / a) >= 3

c) c || 0 && 3+2 >= 2*3-1 && f != b || 3 > a

Comandos de Seleção

- São também chamados de comandos condicionais.
  - if
  - switch

Comando if

- Forma geral:

if (*expressão***)** *sentença1***;**

else *sentença2***;**

- *sentença1* e *sentença2* podem ser uma única sentença, um bloco de sentenças, ou nada.
- O **else** é opcional.

Comando if

- Se *expressão* é verdadeira (!= 0), a sentença seguinte é executada. Caso contrário, a sentença do **else** é executada.

- O uso de if-else garante que apenas uma das sentenças será executada.

Comando if

- comando if pode ser aninhado.
  - Possui em sentença um outro if.
  - ANSI C especifica máximo de 15 níveis.

- **Cuidado**: um else se refere, sempre, ao if mais próximo, que está dentro do mesmo bloco do else e não está associado a outro if.

Comando if -- ambiguidades

```
if ( 1 /*true*/ )
    if ( 0  /*false*/ )
        comando1;
else
        comando2;
```

Não executa nenhum comando ?

Comando if

```
if ( 1 /*true*/ )
{
    if ( 0 /*false*/ )
        comando1;
   else
        comando2;
}
```

Executa comando2

Comando Switch

- **switch** (expressão) {
  case *constante1*: *sequência1*; **break**;
  case *constante2*: *seqüência2*; **break**;
  ...
  default: *seqüência_n*;
  }

Comando Switch - cuidados

- Testa a igualdade do valor da expressão com constantes somente.
- Duas constantes case no mesmo switch não podem ter valores idênticos.
- Se constantes caractere são usadas em um switch, elas são automaticamente convertidas em inteiros
- **break** é opcional.
- **default** é opcional.
- **switch** pode ser aninhado.

Exemplo

```
int x;
scanf(“%d”, &x);
switch (x) {
    case 1: printf(“Um”); break;
    case 2: printf(“Dois”); break;
    case 3: printf(“Tres”); break;
    case 4: printf(“Quatro”); break;
    default: printf(“ default “);
```

Exemplo

```
int x;
scanf("%d", &x);
switch (x) {
    case 1: printf("Um"); break;
    case 2: printf("Dois"); break;
    case 3: printf("Tres"); break;
    case 4: printf("Quatro"); break;
    default: printf("  ");
```

O que acontece se o primeiro *break* for removido?

Exercício

Escreva um comando if que simule a funcionalidade do switch:

```
switch (x) {
    case 1: printf(“Um”); break;
    case 2: printf(“Dois”); break;
    case 3: printf(“Tres”); break;
    case 4: printf(“Quatro”); break;
    default: printf(“  “);
```

Comandos de Iteração

- Comando for
- Comando while
- Comando do-while

Comando for

- **for (***inicialização* **;** *condição* **;** *incremento***)** comando**;**
- As três seções – inicialização, condição e incremento - devem ser separadas por ponto-e-vírgula (;)
- Quando condição se torna falsa, programa continua execução na sentença seguinte ao for.

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

i = 10

Exemplo-for

```
#include <stdio.h>
int main (void) {
   int i;
   for (i=0; i<10; i++)
        printf("%d \n", i);

   return(0);
}
```

1. Inicialização i = 0
2. Teste i < 10
3. Escreve 0
4. i++
5. Teste i <10
6. Escreve 1
7. i++
8. ...
9. Escreve 9
10. i++
11. Teste i < 10
12. return 0

Comando for

- As expressões (inicialização, condição, incremento e comando) são opcionais!

```
int x;
for (x = 0; x != 34; )
   scanf("%d", &x);
```

O que acontece?

Comando while

- Forma geral

  while (*condição***)**

  *comando***;**

- *condição*: é qualquer expressão. Determina o fim do laço: quando a condição é falsa.
- *comando*: pode ser vazio, simples ou um bloco.

Exemplo- while

```
#include<stdio.h>
void main(){
        int x;
        scanf(“%d”,&x);
        while (x != -1)
                scanf(“%d”,&x);
}
```

Comando do-while

- Forma geral

do{

comando ;

} while (*condição***);**

- *comando*: pode ser vazio, simples ou um bloco.
- *condição*: pode ser qualquer expressão. Se falsa, o comando é terminado e a execução continua na sentença seguinte ao do-while

Exercício

- Faça um comando **for** para somar os n primeiros números ímpares, com N sendo lido pelo usuário.
  - Mostre que o programa equivale a calcular N^2.

Exercício

- Escreva um trecho de código, que calcule a quantidade de combinações possíveis de X objetos tomados N a N, com N <= X.

- Escreva um trecho de código para computar a soma de uma matriz diagonal inferior de uma matriz bidimensional. O tamanho da matriz é especificado pelo usuário.

Fatorial

```
#include<stdio.h>

void main(){
    int i, j;
    scanf(“%d”, &i);
    j = i -1;
    do {
      i = i * j;
      j--;
    } while (j > 0);
}
```

Comandos de desvio

- Comando return
- Comando break

Comando return

- Forma geral:

  return *expressão***;**

Comando break

- Dois usos:
  - Terminar um case em um comando switch.

```
int x;
switch(x){
    case 1: printf (“1”);
    case 2: printf (“2”);
    case 3: printf (“3”); break;
    case 4: printf (“4”);
    case 5: printf (“5”); break;
}
```

x = 1 → “123”
x = 2 → “23”
x = 3 → “3”
x = 4 → “45”
x = 5 → “5”

Comando break

- Forçar a terminação imediata de um laço

```
int x;
for ( x = 1; x < 100; x++ )
{
    printf( “%d ”, x );
    if ( x == 13 )
      break;
}
```

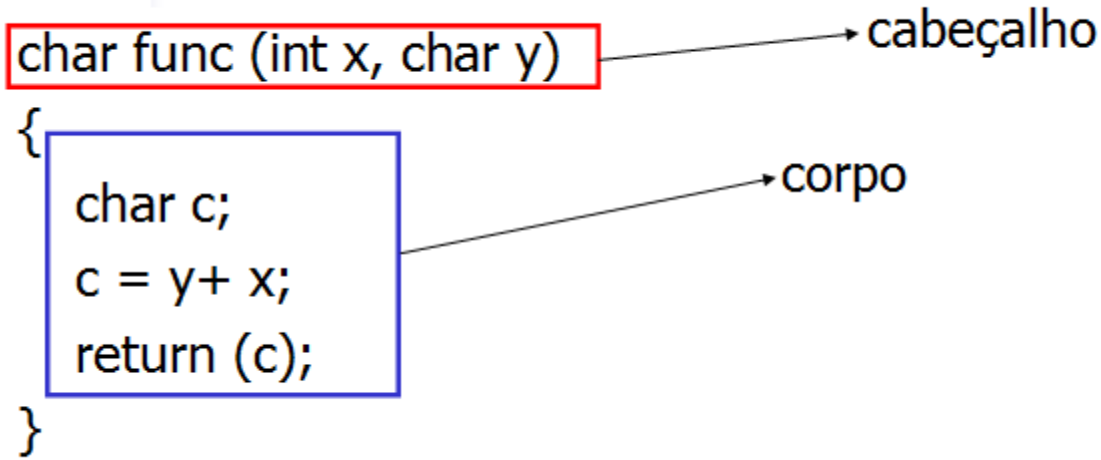
Funções -- declaração

- Forma Geral:

  tipo nome_da_função (lista de parâmetros)
  {declarações sentenças}

  - Tudo antes do “abre-chaves” compreende o **cabeçalho** da **definição** da função.
  - Tudo entre as chaves compreende o **corpo** da **definição** da função.

Exemplo- função



Funções

- *tipo nome_da_função (lista de parâmetros) {declarações sentenças}*
- **tipo:** é o tipo da função, i.e., especifica o tipo do valor que a função deve retornar (*return*).
  - Pode ser qualquer tipo válido.
  - Se a função não retorna valores, seu tipo é **void**.
  - Se o tipo não é especificado, tipo *default* é **int**.
  - Se necessário, o valor retornado será convertido para o tipo da função.

Exemplo

```
int suma(int x,int y);

void main() {
    int x,y, result;
    result= suma(x,y);
}

 int suma (int x, int y) {
      int c;
      c = y + x;
      return (c);
}
```

Passagem por valor

- Modo default de passagem em C
- Na chamada da função, os parâmetros são copiados localmente
- Uma mudança interna não acarreta uma mudança externa

```
void potencia2_valor (int n) { n = n * n; }
```

Passagem por valor

```
void potencia2_valor (int n) { n = n * n; }
```

Chamada:

```
int x = 3;
potencia2_valor ( x );
printf( “%d\n”, x );     //Imprime 3
```

Passagem por referência

- O endereço é passado na chamada da função
- Uma mudança interna acarreta uma mudança externa

```
void potencia2_ref (int *n) { *n = *n * *n ; }
```

Passagem por referência

```
void potencia2_ref (int *n) { *n = *n * *n ; }
```

Chamada:

```
int x = 3;
potencia2_ref ( &x );  //Chamada com endereço
printf( “%d\n”, x );     //Imprime 9
```

Passagem de matrizes

- É sempre feita por referência! Por quê?

```
void sort ( int num [ 10 ] );
void sort ( int num [ ] );
void sort ( int *num );
```

argc e argv

- Passando informações quando o programa é executado via linha de comando
- argc – número de argumentos passados
  - argc > 0 porque o nome do programa é o primeiro argumento
- argv: lista de argumentos passados
  - argv[0] – nome do programa
  - argv[1] – primeiro argumento ...

argc e argv

- argv – vetor de strings

```
void main(int argc, char *argv[])
{
    if ( argc != 2 )
        printf("Você esqueceu o segundo argumento")

    print( "Ola %s\n", argv[1]);
}
```

Chamada: ./program.exe myName

Tipos de funções

- As funções são geralmente de dois tipos
  1. Executam um cálculo: sqrt(), sin()
  2. Manipula informações: devolve sucesso/falha
  3. Procedimentos: exit(), retornam void

Não é necessário utilizar os valores retornados

Funções de tipo não inteiro

- Antes de ser usada, seu tipo deve ser declarado
- Duas formas
  - Método tradicional
  - Protótipos

Funções de tipo não inteiro

- Método tradicional
  - Declaração do tipo e nome antes do uso
  - Os argumentos não são indicados, mesmo que existam

```
float sum();
void main() { .... }
float sum( float a, float b ) {...}
```

Funções de tipo não inteiro

- Protótipo
  - Inclui também a quantidade e tipos dos parâmetros

```
float sum(int  , int);
void main() { .... }
float sum( float a, float b ) {...}
```

  - E no caso de funções sem argumento?
    - Declarar lista como void

Lista de argumentos variável

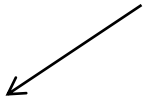
- Uma função pode admitir uma lista com tamanho e tipos variáveis
- Exemplo?

Lista de argumentos variável

- Uma função pode admitir uma lista com tamanho e tipos variáveis
- Exemplo?
  - printf(char *string, ...)
- Utilização de uma macro
  - <stdarg.h>

Exemplo

INDICADOR DE LISTA VARIÁVEL DE PARÂMETROS



```
#include <stdarg.h>

double average( int i, ... )
{
    double total = 0;
    int j;

    va_list ap;          //Cria objeto de manipulação
    va_start( ap, i ); //Inicializa o objeto ap

    for ( j = 1; j <= i; j++ )
        total += va_arg( ap, double )

    va_end(ap);           //Limpando a memória

    return total / i;
}
```

Lista de argumentos variáveis

- va_start(ap, i)
  - ap é o nome da estrutura a ser inicializada
  - o segundo argumento é o nome da última variável antes da elipse (...)
- va_arg(ap, double)
  - Cada chamada retorna o valor passado
  - O segundo argumento representa o tipo do dado esperado na chamada
- va_end(ap): limpeza da estrutura criada

Exercício

- Usando a biblioteca stdarg.h, implemente a função com o seguinte protótipo
  - void printf(char *str, ...);
- Considere como formatação possível apenas %u
  - Range do %u: 0 até 4294967295

Enumeração

- Conjunto de constantes

```
enum months { JAN, FEV, MAR, ABR, MAI, JUN, JUL,
    AGO, SET, OUT, NOV, DEZ };
```

- Primeiro valor é zero, se nenhum valor é especificado
- Outros valores são incrementados de 1

```
enum months { JAN = 1, FEV, MAR, ABR, MAI, JUN, JUL,
    AGO, SET, OUT, NOV, DEZ };
```

Enumeração

```
int main( void )
{
   enum months month; // can contain any of the 12 months

   // initialize array of pointers
   const char *monthName[] = { "", "January", "February",
   "March", "April", "May", "June", "July", "August",
   "September", "October", "November", "December" };

   // loop through months
   for ( month = JAN; month <= DEC; ++month ) {
      printf( "%2d%11s\n", month, monthName[ month ] );
   } // end for
} // end main
```

Arquivos

- Sequência de bytes que reside no disco e não na memória principal
- Endereçamento
  - Sequencial
- Manipulação
  - FILE *fd; //Biblioteca stdio.h

Tipos de arquivos em C

- Arquivos de textos
  - Facilmente editável em programas de edição de texto
- Arquivos binários

Arquivos de texto

- São gravados como caracteres de 8 bits
- A escrita é feita da mesma forma que os dados seriam impressos na tela
- Conversão de dados não texto para dados tipo texto

Quando usar arquivos de texto?

- Exemplo
  - int x = 38472039;
  - char str[8] = “38472039”

- Na memória x ocupa 32 bits
- No arquivo texto, ocupará 8*8 = 64 bits
- A escolha de arquivo texto depende da aplicação

Arquivo binário

- Os dados são gravados exatamente como são armazenados na memória

- Como não há conversão
  - A leitura e escrita são mais rápidas
  - Os arquivos são menores

Manipulação de arquivos de texto

- FILE *fd = fopen(arqName, “r”);
  - Abre para leitura
- FILE *fd = fopen(arqNAme, “w”);
  - Abre para escrita

- Depois de usar o arquivo, é necessário fechá-lo
  - fclose(fd);

Manipulação de arquivos binários

- FILE *fd = fopen(arqName, “rb”);
  - Abre para leitura
- FILE *fd = fopen(arqNAme, “wb”);
  - Abre para escrita

- Depois de usar o arquivo, é necessário fechá-lo
  - fclose(fd);

Leitura - fscanf

- fscanf(FILE *fd, char *str, …)
  - fd = descritor do arquivo a ser lido
  - str = formato a ser lido
  - ... = lista de variáveis a serem lidas

- Se fd == stdin, fscanf equivale a scanf

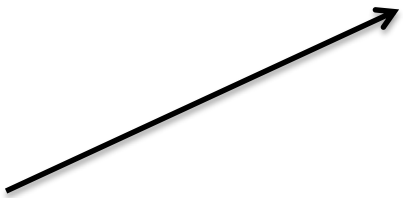
Escrita - fprintf

- fprintf(FILE *fd, char *str, …)
  - fd = descritor do arquivo a ser escrito
  - str = formato a ser escrito
  - ... = lista de variáveis a serem escritas

- Se fd == stdout, fprintf equivale a printf

```c
int main( void) {

    int x, n = 0, k;
    double soma = 0;

    FILE *entrada = fopen( "dados.txt", "r");
    if (entrada == NULL)
        exit( EXIT_FAILURE);

    while ( 1 ) {
        k = fscanf( entrada, "%d", &x);
        if (k != 1) break;
        soma += x;
        n += 1;
    }
    fclose( entrada);
    printf( "A média dos números é %f\n", soma / n);
    return EXIT_SUCCESS;
}
```

dados.txt
1 3 6 1 2 93 12



putc e getc

- int putc (int character, FILE * stream);
  - Escreve caracter no arquivo
  - Retorna o caracter escrito caso tenha sucesso, cc. retorna EOF
- int getc (FILE * stream);
  - Retorna caracter lido
  - Caso erro, retorna EOF

Escrita em arquivos binários

- Mais adequada para a escrita de dados mais complexos, como structs

```
int fwrite (void *buffer, int bytes,
            int count, FILE *fp)
```

Retorna número de unidades escritas com sucesso

fwrite

int fwrite (void *buffer, int bytes,
int count, FILE *fp)

- **buffer**: ponteiro genérico para os dados
- **bytes**: tamanho, em bytes, de cada unidade de dado a ser gravada
- **count**: total de unidades a gravar
- **fp**: ponteiro para o arquivo

fwrite

```c
int main() {

    //Abre para escrita
    FILE *f = fopen ( "file.txt", "wb");
    if ( f == NULL ) exit(1);

    //Vetor de inteiros
    int total_gravado, v[5] = {0,1,2,3,4};

    //Escreve o vetor
    total_gravado = fwrite(v, sizeof(int), 5, f);
    fclose(f);

    return 0;

}
```

fread

- Leitura de bloco de dados de um arquivo
- Usado em conjunto com fwrite

```
int fread ( void *buffer, int bytes,
            int count, FILE *fp );
```

Retorna número de unidades lidas com sucesso

```c
int main() {

    //Abre para escrita
    FILE *f = fopen ( "file.txt", "rb");
    if ( f == NULL ) exit(1);

    //Vetor de inteiros
    int total_lido, v[5];// = {0,1,2,3,4};

    //Escreve o vetor
    total_lido = fread(v, sizeof(int), 5, f);
    fclose(f);

    if (total_lido != 5) exit(1);

    printf( "%d %d %d %d %d\n", v[0], v[1], v[2], v[3], v[4]);

    return 0;
}
```

Acesso randômico em arquivos

- Em geral o acesso é feito de modo sequencial
- A linguagem C fornece ferramentas para realizar leitura e escrita randômica

```
int fseek(FILE *fp, long numbytes, int origem)
```

- numbytes pode ser negativo

Acesso randômico

- A origem pode assumir as seguintes constantes (definida em *stdio.h*)

SEEK_SET (0): início do arquivo

SEEK_CUR (1): posição atual do arquivo

SEEK_END (2): final do arquivo

Acesso randômico em arquivos

```c
struct cadastro { char name[30], int age };

int main () {

    //Abre para escrita
    FILE *f = fopen ( "file.txt", "wb");
    if ( f == NULL ) exit(1);

    //Inicializando vetor de cadastro
    struct cadastro cad[4] = { "a", 1, "b", 2, "c", 3, "d", "4" };

    //Escreve no arquivo
    fwrite( cad, sizeof( struct cadastro ), 4, f );

    return 0;
}
```

Acesso randômico em arquivos

```
//Abre para escrita
FILE *f = fopen ( "file.txt", "rb");
if ( f == NULL ) exit(1);

//Inicializando vetor de cadastro
struct cadastro c;

//Acesso nao sequencial
fseek( f, 2 * sizeof( struct cadastro ), SEEK_SET );
fread( &c, sizeof( struct cadastro ), 1, f );
fclose(f);

//Acessa o registro com dados name == "c" e age == 3
printf( "%s %d\n", c.name, c.age );

return 0;
```

Acesso randômico

- void rewind (FILE *fd)
  - Retorna ao início do arquivo
  - Evita abrir e fechar o arquivo para ir ao início

Acesso randômico

```c
struct cadastro { char name[30], int age };

int main() {

    //Abre para escrita
    FILE *f = fopen ( "file.txt", "rb");
    if ( f == NULL ) exit(1);

    //Inicializando vetor de cadastro
    struct cadastro c;

    //Acesso nao sequencial
    fseek( f, 2 * sizeof( struct cadastro ), SEEK_SET );
    rewind(f);
    fread( &c, sizeof( struct cadastro ), 1, f );
    fclose(f);

    //Acessa o registro com dados name == "a" e age == 1
    printf( "%s %d\n", c.name, c.age );

    return 0;
}
```

Ponteiros

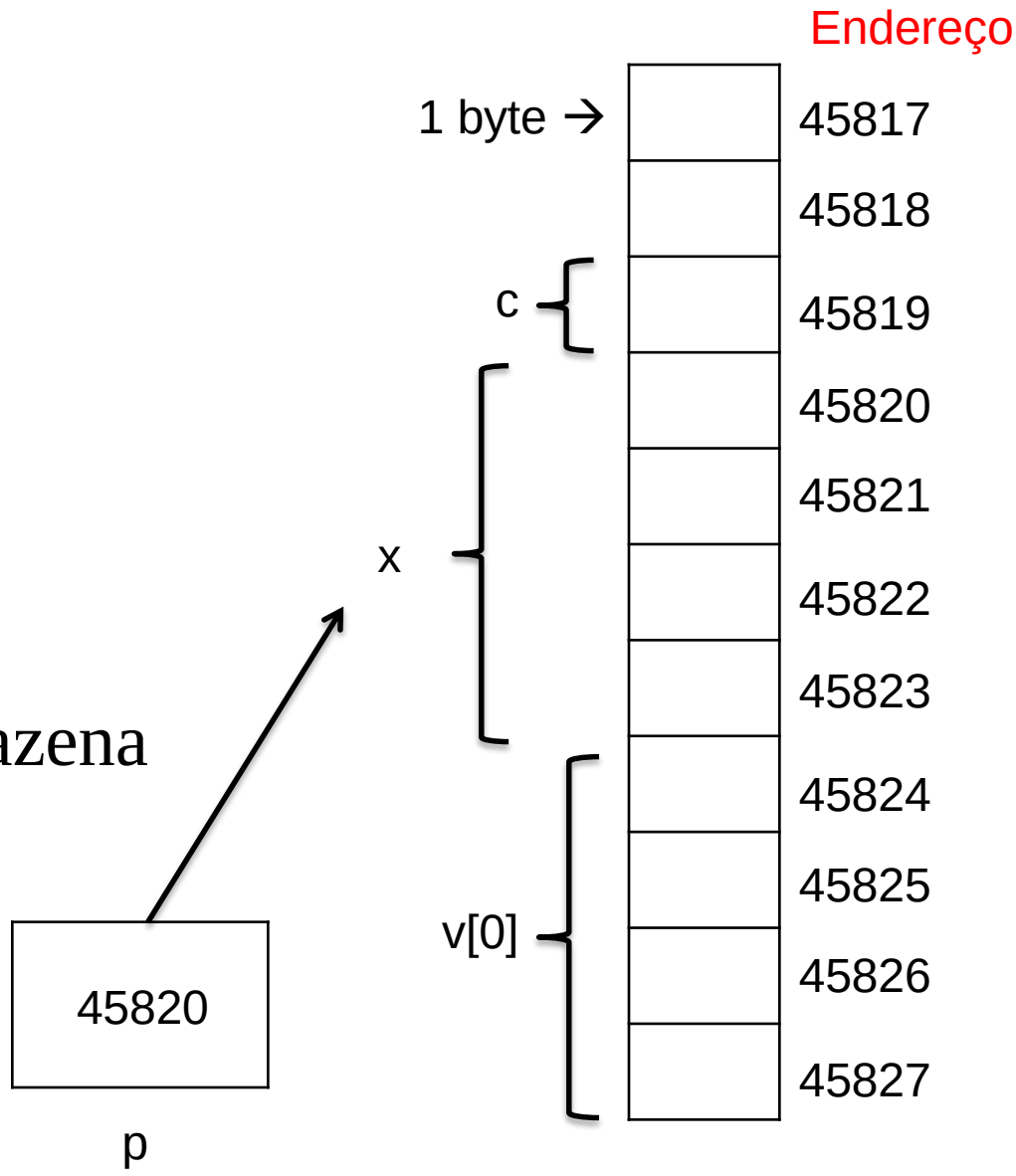
- Ponteiro é uma variável que guarda o endereço de memória de outra variável.

Memória

```
char c;
int x;
int v[10];
```

- Um ponteiro armazena endereços

```
int *p = &x;
//p = 45820
```



Sintaxe

- Forma geral: **tipo *identificador;**
  - **tipo**: qualquer tipo válido em C.
  - identificador: qualquer identificador válido em C.
  - *****: símbolo para declaração de ponteiro. Indica que o **identificador** aponta para uma variável do tipo **tipo**.
- Exemplo:
  - int *p;

Operadores de Ponteiros

- Os operadores de ponteiros são:
  - &
  - *

Operador &

- &: operador unário
- Devolve o endereço de memória de seu operando.
  - Usado durante inicializações de ponteiros.
  - Exemplos

```
int *p, acm = 35;
p = &acm;  /*p recebe “o endereço de” acm*/
```

Operador *

- Devolve o valor da variável apontada.
  - Ex.:

```
int *p, q, acm = 35;
p = &acm;
q = *p;     /*   q recebe o conteúdo da variável
                 “no endereço” p   */
```

  - O valor de q é 35

Exercício

- Seja a seguinte seqüência de instruções em um programa C:

  ```
  int *pti;
  int i = 10;
  pti = &i;
  ```

Qual afirmativa é **falsa**?

- **a.** pti armazena o endereço de i
- **b.** *pti é igual a 10
- **c.** ao se executar *pti = 20; i passará a ter o valor 20
- **d.** ao se alterar o valor de i, *pti será modificado
- **e.** pti é igual a 10

Ponteiros genéricos

- Pode apontar para todos os tipos de dados existentes ou que serão criados

```
void *ptr;
```

Ponteiros genéricos

```
void *pp;
int *p1, p2 = 10;

p1 = &p2;
pp = &p2;        //Endereço de int
pp = &p1;        //Endereço de int *
pp = p1;         // Endereço de int
```

Ponteiros genéricos

- Acesso depende do tipo!

```
void *pp;
int p2 = 10;

pp = &p2;
printf ( “%d\n”, *pp );
```

Ponteiros genéricos

- Acesso depende do tipo!

```
void *pp;
int p2 = 10;

pp = &p2;
printf ( “%d\n”, *pp ); //Erro
printf( “%d\n”, * ( (int *) pp ) );
```

Ponteiros genéricos

- Aritmética de ponteiros:

```
void *p = 0x9C4; //2500
p++; //2501 -- Sempre soma 1 byte
p = p + 15; //2516
p--;   // 2515 -- Sempre subtrai um byte
```

O programador deve considerar o tipo

Ponteiros para ponteiros

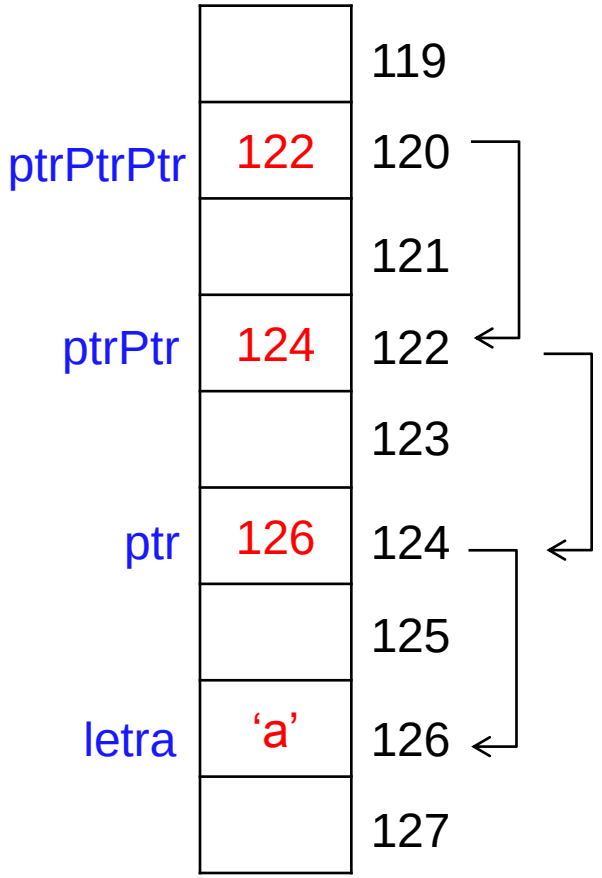
Armazena o endereço de ponteiros

```
int x = 10
int *p = &x;
int **p2 = &p;
```

END	VARIÁVEL	CONTEÚDO
130	int **p2	132
131		
132	int *p	134
133		
134	int x	10
135		

Ponteiro para ponteiro

```
char letra = ‘a’;

char *ptr = &letra;
char **ptrPtr = &ptr;
char ***ptrPtrPtr = &ptrPtr;
```



Ponteiros para ponteiros

Armazena o endereço de ponteiros

```
int *p;
int **r;
p = &a;
r = &p;
c= **r + b
```

Exercício

1. Escreva uma função mm que receba um vetor inteiro v[0..n-1] e os endereços de duas variáveis inteiras, digamos min e max, e deposite nessas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função main que use a função mm.
2. **Escreva uma função que troque os valores de duas variáveis**

Exercício

- Solução alternativa?

```
void troca( int *i, int *j )  {
   int *temp;
   *temp = *i;
   *i = *j;
   *j = *temp;
}
```

Exercício

- Escreva uma função que troque os valores de duas variáveis

```
void troca( int *p, int *q)
{
    int temp;
    temp = *p; *p = *q; *q = temp;
}
```

Alocação dinâmica

- Para que serve?

Alocação dinâmica

- Cadastro de funcionários de uma empresa?

  struct func_type funcionarios[1000];

  Pode haver desperdício ou faltar espaço.

Malloc (stdlib.h)

```
void *malloc ( unsigned int num );
```

```
//array de 50 inteiros
int *v = (int *) malloc ( 200 );

//string de 200 caracteres
char *c = (char *) malloc( 200 );
```

Para reservar memória dinamicamente

- Para reservar memória dinamicamente num vetor, usa-se ponteiros da seguinte forma:

```
float *x;
x = ( float * ) malloc ( nx * sizeof ( float ) ) ;
```

nx é o numero de elementos que vai ter o array

Para reservar memória dinamicamente

- Para liberar a memória alocada dinamicamente:

```
float *x;
x = ( float * ) malloc ( nx * sizeof ( float ) ) ;
//nx é o numero de elementos que vai ter o array
free(x);
```

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

int main()
{

    //Se não tiver memória suficiente,
    //retorna NULL
    int *p = (int *) malloc( 5 * sizeof(int) );
    if ( p == NULL ) exit(1);

    int i;
    for ( i = 0; i < 5; i++ )
        scanf( "%d", &p[i] );

    //Libera memória
    free(p);

    return 0;
}
```

calloc (stdlib.h)

```
void *calloc (unsigned int num,
              unsigned int size );
```

num: número de unidades alocadas

size: tamanho de cada unidade

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

int main()
{

    //Se não tiver memória suficiente,
    //retorna NULL
    int *p = (int *) calloc( 5, sizeof(int) );
    if ( p == NULL ) exit(1);

    int i;
    for ( i = 0; i < 5; i++ )
        scanf( "%d", &p[i] );

    //Libera memória
    free(p);

    return 0;
}
```

malloc vs. calloc

- Malloc
  - Apenas aloca a memória
- Calloc
  - Aloca a memória
  - Inicializa todos os bits da memória com zero

realloc (stdlib.h)

- Útil para alocar ou realocar memória durante a execução.
  - Ex.: Aumentar a quantidade de memória já alocada.

void *realloc (void *ptr, unsigned int num);

OS DADOS EM PTR **NÃO** SÃO PERDIDOS

realloc

void *realloc (void *ptr, unsigned int num);

- ptr: ponteiro para bloco já alocado
- num: número de bytes a ser alocado
- Retorna ponteiro para primeira posição do array ou NULL caso não seja possível alocar

```
int *v = (int *) malloc ( 50 * sizeof(int) );
v = (int *) realloc( v, 100 * sizeof(int) );
```

realloc

- Se ptr == NULL, então realloc funciona como malloc

```
int *p;
p = (int *) realloc( NULL, nx * sizeof ( int ) );
p = (int *) malloc( nx * sizeof(int) );
```

realloc

- Realloc pode ser utilizado para liberar a memória

```
int *p = ( int * ) malloc ( nx * sizeof ( int ) );
p = ( int * ) realloc ( p, 0 ) ;
```

realloc

- Realloc pode ser utilizado para liberar a memória

```
int *p = ( int * ) malloc ( 50 * sizeof ( int ) );
p = ( int * ) realloc ( p, 100 * sizeof ( int ) );
```

QUAL O PROBLEMA DO CÓDIGO ACIMA ?

realloc

- Realloc pode ser utilizado para liberar a memória

```
int *p = ( int * ) malloc ( 50 * sizeof ( int ) );
p = ( int * ) realloc ( p, 100 * sizeof ( int ) );
```

PONTEIRO PODE SER NULL E O VETOR ALOCADO ANTERIORMENTE É PERDIDO

Ponteiros: utilidade

- Retornar mais de um valor em uma função
  - Exemplo: swap(a,b)

- Alias de vetor

```
int v[5] = {1,2,3,4,5}
//v é um ponteiro que aponta para v[0]
// v[3] == *(v+3)
```

Ponteiros e vetores

int v[5] = {1,2,3,4,5}

- v+i equivale a &v[i]
- *(v+i) equivale a v[i]

Ponteiros como aliases

Se v é um vetor de inteiros, então há equivalência?

```
int v[] = {1,2,3};
printf( “%d”, *(v+1) );
printf( “%d”, *(++v) );
```

Ponteiros constantes

Se v é um vetor de inteiros, então há equivalência?

```
int v[] = {1,2,3};
printf( “%d”, *(v+1) );
printf( “%d”, *(++v) );
```

Não há equivalência, pois o nome do vetor é um ponteiro constante

Operações com ponteiros

```
int *px, *py;

....
if( px < py )
px = py + 5;
px – py; //número de variáveis entre px e py
px++;
```

Obs.: se px é ponteiro para int, incrementa o tamanho de um inteiro

Arrays de ponteiros

Exemplo:

Criação de matriz de inteiros

int *vet [size];

Cada posição pode ser alocada dinâmicamente

```c
#include <stdlib.h>

int main()
{

    int *pvet[2];
    int x = 10;
    int y[2] = {20,30};

    pvet[0] = &x;
    pvet[1] = y;

    printf( "pvet[0]: %p\n", pvet[0] ); // &x
    printf( "pvet[1]: %p\n", pvet[1] ); // &y[0]

    printf( "*pvet[0]: %d\n", *pvet[0] ); //x
    printf( "pvet[1][1]: %d\n", pvet[1][1] ); //y[1]

}
```

Exercício

- Faça um programa que permita criar uma lista de alunos (gravados pelo seu número USP), sendo que o número de alunos é determinado pelo usuário.
  - Utilize a **alocação dinâmica** de vetores

Estruturas

- Coleção de variáveis organizadas em um único conjunto.
  - Possivelmente coleção de tipos distintos
- As variáveis que compreendem uma estrutura são comumente chamadas de **elementos** ou **campos**.

Exemplo-estruturas

- Definição x Declaração

```
struct pessoa
{
    char nome[30];
    int idade;
};
```

- Permite declarar variáveis cujo tipo seja **pessoa**.

Usando typedef nas estruturas

```
struct a{
    int x;
    char y;
};

typedef struct a MyStruct;

int main(){
    MyStruct b; /*declaração da var b, cujo tipo é MyStruct*/
}
```

Acesso aos dados da Estrutura

- É feito via o ponto (.)

```
int main (void){
        MyStruct obj;
        obj.x = 10;
        obj.y = ‘a’;
}
```

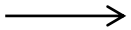
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct pessoa {
    char nome[50], rua[50];
    int idade, numero;
};

int main() {

    struct pessoa p;
    strcpy(p.nome, "Nome" );
    strcpy(p.rua, "Street 4" );
    p.idade  = 27;
    p.numero = 1874;

    return 0;

}
```

INICIALIZAÇÃO CAMPO A CAMPO

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

struct pessoa {
    char nome[50], rua[50];
    int idade, numero;
};

int main() {

    struct pessoa p = { "Nome",
    "Street 4", 27, 1874 };

    //Campos não inicializados
    //explicitamente são inicializados
    //com zero
    struct pessoa p2 = { "Nome2",
    "Street 4", 27 };

    return 0;
}
```

INICIALIZAÇÃO COMO VETOR

ATRIBUIÇÃO COMO VARIÁVEL NORMAL



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

struct ponto {
    int x,y;
};

struct ponto_B {
    int x,y;
};

int main() {

    struct ponto p1, p2 = {1,2};
    struct ponto_B p3 = {3,4};

    p1 = p2; //OK
    p1 = p3; //Erro ! Tipos diferentes

    return 0;
}
```

Aninhamento de estruturas

```
struct tipo_struct1 {...};

struct tipo_struct2 {
    ...
    struct tipo_struct1 nome;
}
```

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

struct endereco {
    char rua[80];
    int numero;
};

struct pessoa {
    char nome[50];
    int idade;
    struct endereco ender;
};
```

```c
int main() {

    struct pessoa p;
    p.idade = 31;
    p.ender.numero = 103;

    return 0;
}
```

Ponteiros para estruturas

- Utilidade?

Ponteiros para estruturas

- Utilidade
  1. Evitar overhead em chamadas de funções
  2. Criação de listas encadeadas

Ponteiros para estruturas

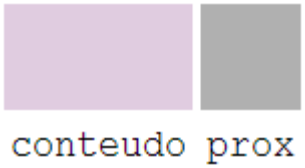
- Exemplo

```
...
struct bal {
  char name[30];
  float balance;
} person;

struct bal *p;


p = &person; //Apontando para
p->balance;  //Acesso ao conteúdo
```

Listas encadeadas

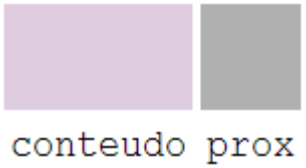
- É uma representação de uma sequência de objetos na memória do computador

```
struct cel {
        int conteudo;
        struct cel *prox;
};
typedef struct cel celula;
```



Operações

```
celula c;
celula *p;

...
p = &c;                    //Endereço da célula
p->conteudo;               //Conteudo atual
p->prox->conteudo;  //Conteudo do próximo
```

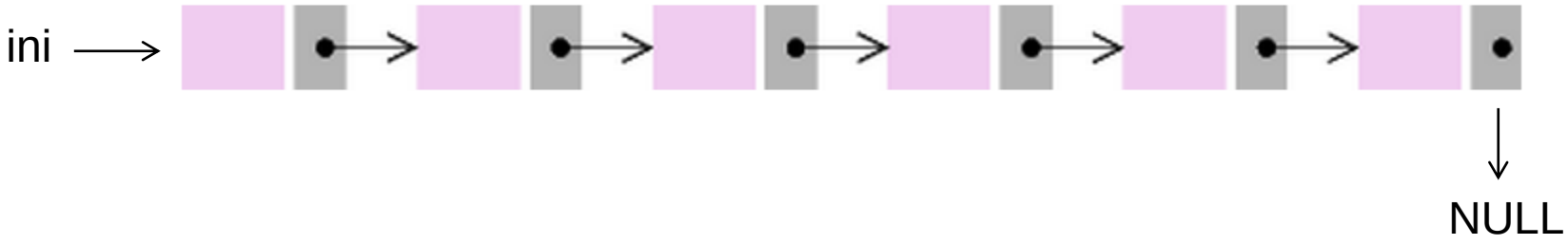


Lista sem cabeça

- Primeiro elemento faz parte da lista

- Inicialização

```
celula *ini;
ini = NULL  //Lista esta vazia
```



Exercício

- Escreva uma função que imprime o conteúdo de uma lista sem cabeça

```
void imprima ( celula *ini );
```

Exemplo

```
void imprima( celula *ini )
{
   celula *p;
   for (p = ini; p != NULL; p = p->prox)
        printf( "%d\n", p->conteudo);
}
```

Exercício

- Escreva uma função que busca o inteiro x na lista. A função devolve um ponteiro para o registro encontrado ou NULL, caso não encontre.

Busca na lista

```
celula *busca( int x, celula *ini)
{
   celula *p = ini;
   while (p != NULL && p->conteudo != x)
         p = p->prox;
   return p;
}
```

Exercício

- Escreva uma que insira uma nova célula com conteúdo x em uma lista ordenada. A lista deve continuar ordenada após a inserção
  - Utilize alocação dinâmica

```c
int insert( celula **ini, int x )
{

    //Aloca nova celula
    celula *new = ( celula * ) malloc ( sizeof ( celula ) );
    if  ( new == NULL ) exit(0);
    new->conteudo = x;

    //Inserçao no começo modifica ini
    if ( *ini == NULL || (*ini)->conteudo > x )
    {
        new->prox = *ini;
        *ini = new;
        return 0;
    }
```

```
//Inserção no meio
celula *q = *ini;
celula *p = q->prox;

//Encontra a posição
while ( p != NULL && p-> conteudo < x )
{
    q = p;
    p = p->prox;
}

//Ajusta os ponteiros
q->prox = new;
new->prox = p;

return 0;
```

Exercício

- Escreva uma função que inverta uma lista

```
int reverse ( celula **ini )
```