AED1 – Aula 01
Breve Revisão de Introdução à Programação

Wanderley de Souza Alencar
wanderleyalencar@ufg.br

Universidade Federal de Goiás - UFG
Instituto de Informática - INF

3 de março de 2020

Sumário

Algoritmos

Algoritmos

Conceito

Um *algoritmo* é qualquer procedimento computacional bem definido que recebe um valor (ou conjunto de valores) como `entrada` e produz algum valor (ou conjunto de valores) como `saída`.

Um *algoritmo* é, assim, uma sequência finita de passos computacionais que transformam a `entrada` em `saída`.

(COMEN, Thomas H. et al., *Introduction to Algorithms*, 3rd. ed., MIT Press, 2009, p. 5).

Algoritmos

Conceito

Um *algoritmo* pode ser entendido como uma *ferramenta* para resolver um `problema computacional` bem especificado.

Por exemplo: Ordenar, de maneira crescente, um conjunto de n números inteiros fornecidos como entrada.

Algoritmos

Exemplo – Ordenação Numérica

Entrada Uma sequência de n números inteiros: $(a_1, a_2, a_3, \ldots, a_n)$;

Saída Uma permutação (reordenamento) $(a'_1, a'_2, a'_3, \ldots, a'_n)$ da sequência de entrada de tal maneira que: $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Algoritmos

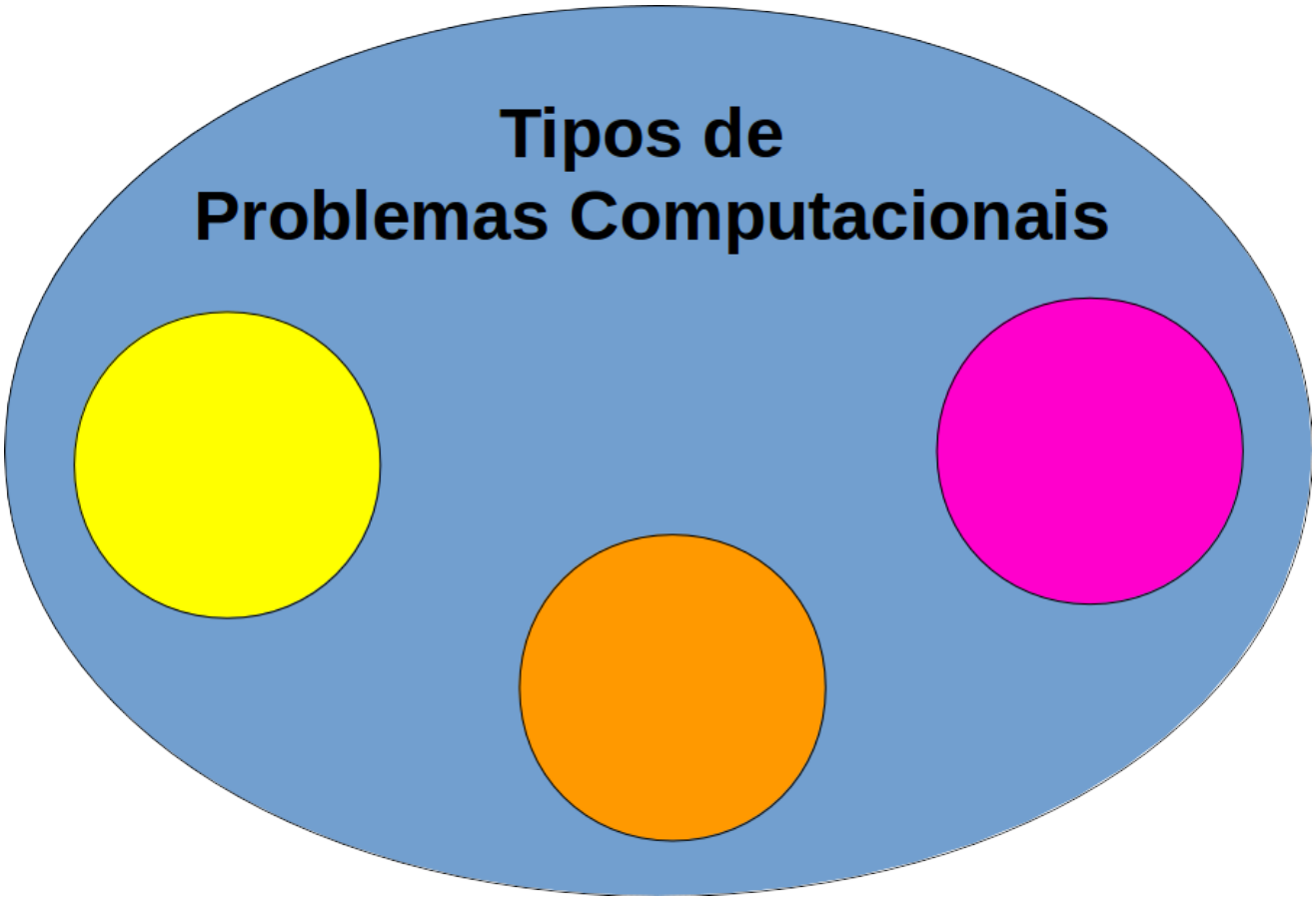
Exemplo – Ordenação Numérica

Entrada $(54, 32, 74, 89, 14, 65, 37, 98)$.

Saída $(14, 32, 37, 54, 65, 74, 89, 98)$.

Temos, acima, uma *instância* do problema de ordenação numérica: um conjunto de valores que representa uma `entrada` específica (54, 32, ..., 98) para a geração de uma solução para o problema.

Problemas Computacionais



Problemas Computacionais

Classes de Problemas

Os *problemas computacionais* podem ser classificados de diversas maneiras, segundo variados critérios de acordo com a abordagem desejada.

A área denominada de *Teoria da Computabilidade* identifica TRÊS CLASSES de problemas computacionais.

Problemas Computacionais

Classes de Problemas

(1) Problemas de Decisão;
(2) Problemas de Localização;
(3) Problemas de Otimização.

Problemas Computacionais

Classes de Problemas: (1) Problema de Decisão

É o problema computacional que pode ser formulado como uma questão para ser respondida com um simples SIM ou NÃO, ou seja, nada mais é necessário na resposta.

Problemas Computacionais

Classes de Problemas: (1) Problema de Decisão

Problemas Computacionais

Classes de Problemas: (1) Problema de Decisão

- O número natural $n \in \mathbb{N}^*$ é *primo*?
- Dados dois números naturais x e y, y divide x de maneira exata?
- É possível desenhar um círculo inscrito num quadrado cujo lado mede x unidades?
- Dado um mapa das vias de certo bairro x de uma cidade C, existe um caminho que conecte dois endereços distintos a e b neste bairro?

...

Problemas Computacionais

Classes de Problemas: (2) Problema de Localização

É o problema cuja enunciação exige a *localização* de pelo menos uma solução que satisfaça a determinados critérios de aceitação.

O termo *localização* significa que a solução deve ser apresentada, ou descrita de tal maneira que seja possível identificá-la.

Assim, uma solução pode ser aceita, mesmo não sendo a *melhor solução* para o problema.

Problemas Computacionais

Classes de Problemas: (2) Problema de Localização

Problemas Computacionais

Classes de Problemas: (2) Problema de Localização

- Apresente um caminho para ir, de automóvel, de um ponto A a outro ponto B, numa certa cidade X, sempre obedecendo as normas de trânsito locais.
- Qual é o máximo divisor comum entre dois números naturais x e y?
- Dada a expressão de uma função real, digamos $f(x)$, definida no intervalo real $[a, b]$, qual é uma valor de x que a torna *negativa*?

...

Problemas Computacionais

Classes de Problemas: (3) Problema de Otimização

É o problema cujo objetivo é encontrar a *melhor solução* para um particular conjunto de valores de entrada, segundo um rol de critério(s) bem estabelecido(s) inicialmente.

Normalmente, quando comparados às classes anteriores, estes são problemas *mais díficeis* de resolver, pois não basta responder SIM ou NÃO, nem localizar "uma solução" qualquer, o que se deseja é a *melhor solução* segundo o rol de critérios estabelecidos.

Problemas Computacionais

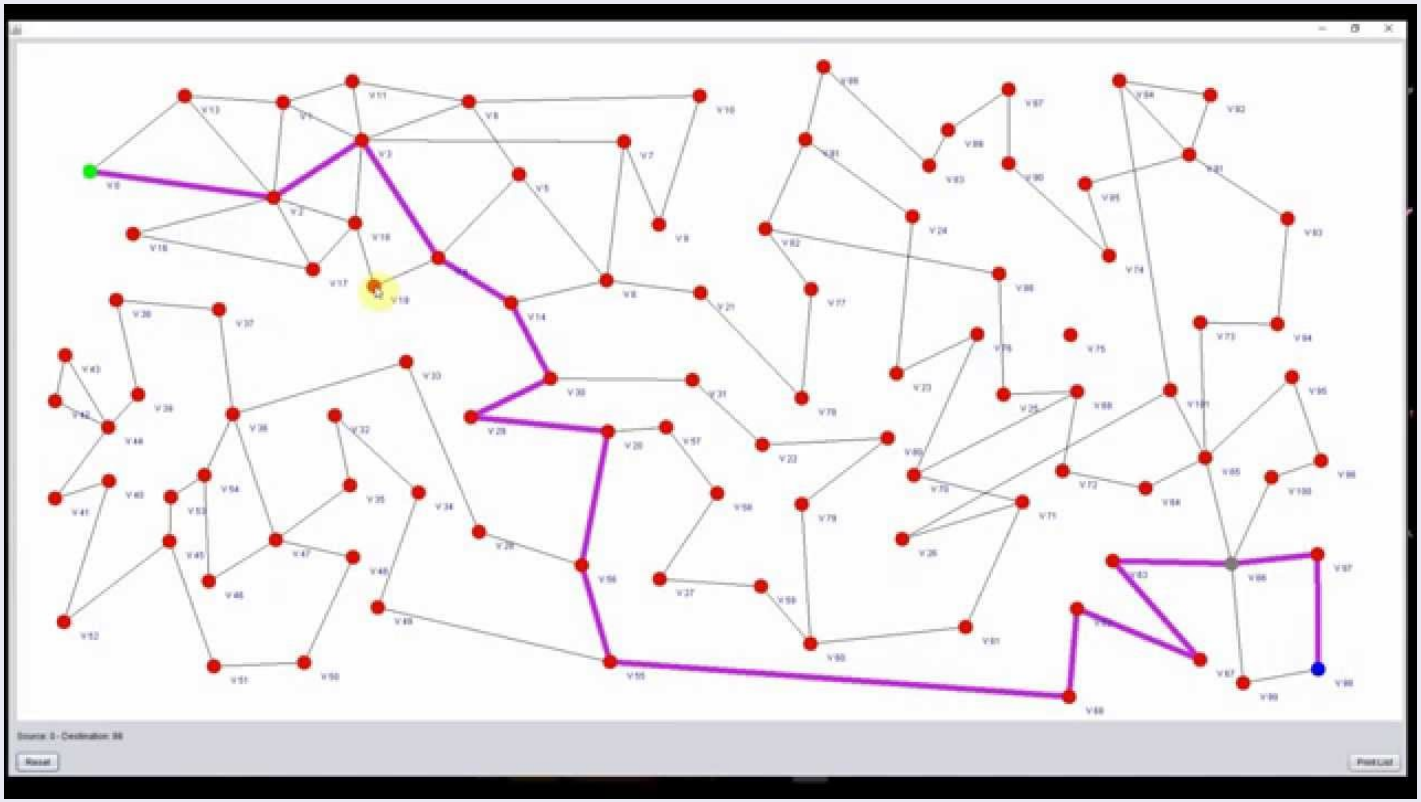
Classes de Problemas: (3) Problema de Otimização

- Qual é o caminho de *menor comprimento* para ir, de automóvel, de um ponto x a outro ponto y, numa certa cidade C, sempre obedecendo as normas de trânsito locais?
- Tendo um fio cujo comprimento é de ℓ unidades, qual é a maior área que se pode obter utilizando-o para desenhar um polígono regular?
- Dada expressão de uma função, digamos $f(x)$, definida no intervalo real $[a, b]$, qual é uma valor de x que a torna *máxima*?

...

Problemas Computacionais

Classes de Problemas: (3) Problema de Otimização



Mais Algoritmos

Mais Algoritmos

Algoritmo (elaboração)

Ao elaborar um *algoritmo*, frequentemente devemos nos preocupar com a sua:

- *eficácia* – ele resolve, completa e corretamente, o problema definido e para o qual foi elaborado?
- *eficiência* – ele resolve o problema definido utilizando eficientemente os recursos computacionais, notadamente *tempo de processamento* e *memória principal*?

Mais Algoritmos

Algoritmo (elaboração)

Ao elaborar um algoritmo que se proponha atingir estes objetivos (eficácia e eficiência), tem-se que dar extrema atenção às:

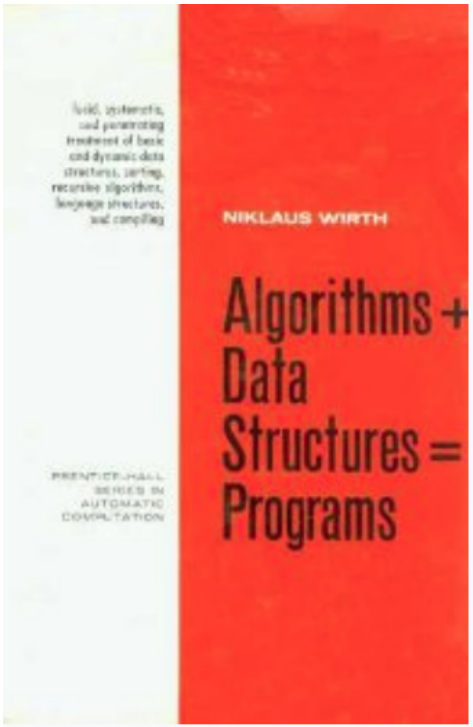
- *estruturas de controle* e
- *estruturas de dados*

utilizadas para a elaboração dele.

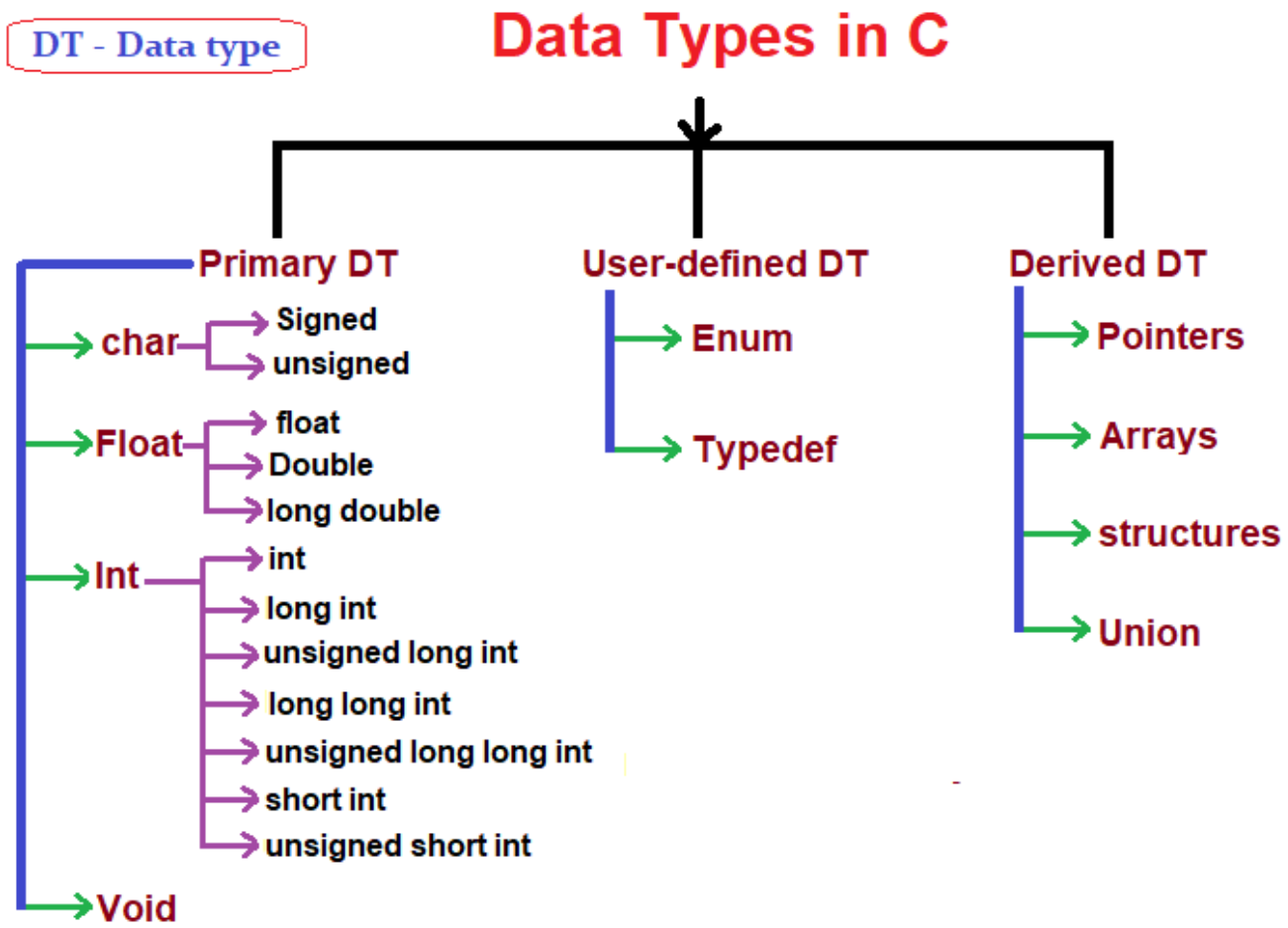
Mais Algoritmos

Niklaus Wirth (1934 – . . .)

Algorithms + Data Structures = Programs.



Tipos de Dados



Tipos de Dados

Relembrando...

Da disciplina *Introdução à Programação* sabemos que há tipos de dados:

(1) *primitivos* (elementares, básicos ou fundamentais);

(2) *construídos* (estruturados ou derivados); e

(3) *definidos pelo usuário*.

Tipos de Dados

(1) Primitivos

São, normalmente, tipos correspondentes à própria estrutura (ou arquitetura) do sistema computacional subjacente em seu nível mais elementar (de *hardware* ou muito próximo a ele). Tipicamente correspondem a:

- números (inteiro, real);
- caracteres individuais;
- valor lógico (verdadeiro/falso).

Tipos de Dados

(1) Primitivos

Na linguagem $\mathbb{C}$, são primitivos:

- char (unsigned, signed);
- int (unsigned, signed | short, long, long long);
- float (double, long double);
- void.

Tipos de Dados

(2) Construídos

São tipos obtidos a partir da combinação de tipos primitivos ou de outros tipos (também construídos):

- vetores e matrizes;
- registros;
- uniões;
- vetores de registros;
- registro de matrizes;

...

Tipos de Dados

(2) Construídos – Registro

Um registro (ou *record*) é um agregado de dados que possuem, entre si, algum peculiar relacionamento de tal maneira que sua agregação é conveniente para sua melhor compreensão e/ou manipulação.

Cada um destes dados é individualmente nomeado de *campo* e, usualmente, são de tipos diferentes entre si.

Os *campos* são também chamados de *membros*, ou *atributos*, notadamente no âmbito da orientação a objetos.

Tipos de Dados

(2) Construídos – Registro

Pode-se comparar um registro a uma "ficha" que possui todos os dados sobre uma determinada entidade, por exemplo:

- registro de um(a) estudante (número de matrícula, nome, médias de provas, médias de notas de laboratório, etc.);
- registro de um produto industrial (código UPC/EAN, nome, nome da empresa fabricante, descrição, etc.).

...

Tipos de Dados

(2) Construídos – Registro (declaração)

Em $\mathbb{C}$, declara-se um registro utilizando a palavra chave `struct`:

```
1 struct {
2    tipo_1 nome_1;
3    tipo_2 nome_2;
4    tipo_3 nome_3;
5    // continua...
6    tipo_n nome_n;
7 }
```

Tipos de Dados

(2) Construídos – Registro (declaração)

```c
#include <stdio.h>

struct aluno {
    int mat; // campo: matricula
    float media; // campo: media do aluno
}

int main() {
     struct aluno j;
     j.mat = 10;
     j.media = 8.5;
     printf("Matricula \%d, media \%f.\n", j.mat, j.media);
}
```

Tipos de Dados

(2) Construídos – Registro (leitura)

Em $\mathbb{C}$, a leitura é realizada *por campo*:

```
1  //
2  // preliminar...
3  //
4  printf ("Digite a matricula do aluno: ");
5  scanf ("\%d", \&j.mat);
6  printf ("Digite a media do aluno: ");
7  scanf ("\%f", \&j.media);
8  //
9  // continua...
10 //
```

Tipos de Dados

(2) Construídos – Registro (cópia)

A cópia pode ser *atômica*, ou seja, realizada com todo o agregado de dados:

```
1  struct registro {
2      tipo_1 nome_1;
3      tipo_2 nome_2;
4      tipo_3 nome_3;
5      // continua...
6      tipo_n nome_n;
7  }
8  int main() {
9       struct registro registro_1, registro_2;
10
11      registro_1 = registro_2;
12 }
```

Tipos de Dados

(2) Construídos – Registro (composição)

Registros podem ser elementos de um vetor:

```
1  #include <stdio.h>
2  struct aluno {
3      int mat;
4      float media;
5  }
6  int main () {
7       struct aluno turma[40];
8
9       turma[0].mat = 10;
10      turma[0].media = 8.5;
11      // continua...
12 }
```

Tipos de Dados

(2) Construídos – Registro (aninhamento)

Registros podem ser campos de outros registros:

```
1  #include <stdio.h>
2  struct aluno {
3      int mat;
4      float media;
5  }
6  int main () {
7       struct notas {
8            float p1;
9            float p2;
10           float p3;
11      }
12      typedef struct notas NotasAluno;
13      struct aluno {
14           int mat;
15           NotasAluno provas;
16      }
17 }
```

Tipos de Dados

(2) Construídos – Registro (sizeof)

Em $\mathbb{C}$, o operador unário `sizeof` calcula o tamanho de qualquer variável ou tipo construído, retornando um número inteiro que o representa.

Dessa forma, o resultado de `sizeof` pode ser apresentado num dispositivo de saída usando o comando `printf`.

Tipos de Dados

(2) Construídos – Registro (sizeof)

```c
#include <stdio.h>

struct aluno {
    int mat; // campo: matricula
    float media; // campo: media do aluno
}

int main() {
     int i;
     struct aluno x;

     printf("%d\n", sizeof(i));
     printf("%d\n", sizeof(x));

}
```

Tipos de Dados

(3) Definidos pelo Usuário

Em diversas linguagens de programação o usuário pode *definir* novos tipos de dados que, a partir daquele momento, passarão a integrar o programa onde foram declarados.

Em $\mathbb{C}$, por exemplo, pode-se definir uma *enumeração*, que é utilizada para associar nomes para constantes inteiras, permitindo que o programa fique mais fácil de ler e manter.

Tipos de Dados

Definidos pelo Usuário – enum

```
1  #include <stdio.h>
2
3  enum Status {EmOperacao = 1, Interrompido = 0};
4  enum Semana {Seg, Ter, Qua, Qui, Sex, Sab, Dom};
5
6  int main() {
7      Status resultadoOp;
8      Semana diaSemana;
9      \\
10     \\ continua...
11     \\
12 }
```

Tipos de Dados

(3) Definidos pelo Usuário

Outro exemplo em $\mathbb{C}$ é quando se utiliza a palavra reservada `typedef` para declarar um novo tipo de dado.

Tipos de Dados

Definidos pelo Usuário – typedef

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  //
4  //redefinindo os tipos: float e int
5  //
6  typedef float nota;
7  typedef int inteiro;
8  struct tAluno {
9     inteiro matricula ;
10    nota prova1;
11    nota prova2;
12 };
13 //
14 //redefinindo a struct para encurtar o comando na declaraÃ§Ã£o
15 //
16 typedef struct tAluno tAluno;
```

Tipos de Dados

Definidos pelo Usuário – typedef

```
1  int main (void) {
2      tAluno aluno;
3      nota media = 0;
4
5      printf ("Informe o numero de matricula: "); scanf ("%d", &aluno.matricula);
6
7      printf ("Informe a nota da primeira prova: "); scanf ("%f", &aluno.prova1);
8
9      printf ("Informe a nota da segunda prova: "); scanf ("%f", &aluno.prova2);
10
11     media = (aluno.prova1 + aluno.prova2) / 2;
12
13     printf("\nMatricula.....: %d\n", aluno.matricula);
14     printf("Media do aluno: %.2f", media);
15     getch();
16     return (0);
17 }
```

Alocação Dinâmica

Chamada de Funções e Variáveis Locais

Memória Dinâmica

Dados Globais

Instruções

Alocação Dinâmica

Conceito

Existem DUAS maneiras fundamentais de um programa armazenar dados na memória principal:

(1) Alocação Estática: variáveis locais e globais cujo o armazenamento é fixo durante toda a execução do programa e/ou rotina;

(2) Alocação Dinâmica: variáveis cujo armazenamento é variável, controlado em tempo de execução por meio de comandos que o programador insere no código-fonte do programa.

Alocação Dinâmica

Conceito: (1) Alocação Estática

Em $\mathbb{C}$, a *alocação estática* é aquela obtida quando se declara uma variável de algum tipo de dado primitivo (int, float, double, char, etc.), vetores e matrizes de tamanho previamente fixado (int vetor[10], ...), estruturas, uniões, etc.

A variável residirá na memória principal durante toda a execução do programa ou rotina em que foi declarada.

Alocação Dinâmica

Conceito: (2) Alocação Dinâmica

A *alocação dinâmica* habitualmente utiliza *comandos especiais* da linguagens de programação sendo empregada para *alocar* e *liberar* memória principal durante o transcorrer da execução do programa – daí deriva o nome *dinâmica*.

Em $\mathbb{C}$ temos...

Alocação Dinâmica

$\mathbb{C}$ ANSI

O padrão $\mathbb{C}$ ANSI especifica apenas QUATRO funções para alocação dinâmica:

- calloc();
- malloc();
- free();
- realloc();

As funções de alocação dinâmica definidas pelo padrão $\mathbb{C}$ ANSI estão na biblioteca stdlib.h.

Alocação Dinâmica

calloc()

```
void *calloc(size_t num, size_t size);
```

- A função `calloc()` aloca uma quantidade de memória igual a `num` $\times$ `size`. Ou seja, aloca memória suficiente para uma matriz de `num` objetos de tamanho igual a `size` cada um.
- A função devolve um *ponteiro* para o primeiro endereço da região alocada;
- Se não houver memória suficiente é devolvido um ponteiro nulo (`NULL`).

Alocação Dinâmica

calloc()

```
1  #include<stdlib.h>
2  #include<stdio.h>
3
4  float *get_mem(void){
5        float *p;
6
7        p=calloc(100, sizeof(float));
8        if(!p){
9           printf("Erro de alocacao - abortando.");
10          exit(1);
11       }
12       return(p);
13 }
```

Alocação Dinâmica

malloc()

```
void *malloc(size_t size);
```

- A função `malloc()` devolve um ponteiro para o primeiro byte de uma região de memória de tamanho `size` que foi alocada do *heap* de memória;
- Se não houver memória suficiente é devolvido um ponteiro nulo (`NULL`);
- Deve-se sempre verificar se o valor devolvido não é um ponteiro nulo antes de utilizá-lo para armazenar qualquer dado.

Alocação Dinâmica

malloc()

```
1  #include<stdlib.h>
2  #include<stdio.h>
3  struct endereco{
4      char nome[40];
5      char rua[40];
6      char cidade[40];
7      char estado[2];
8  };
9  struct endereco *get_struct(void){
10         struct endereco *p;
11
12         if ((p = malloc(sizeof(struct endereco))) == NULL) {
13             printf("Erro de alocacao");
14             exit(1);
15         }
16         return(p);
17 }
```

Alocação Dinâmica

free()

```
void free(void *ptr);
```

- A função `free()` devolve ao *heap* a memória apontada por `ptr`, tornando a memória novamente disponível para alocação futura;
- `free()` deve ser chamado apenas com um ponteiro que foi previamente alocado com as funções de alocação dinâmica;
- O uso de um ponteiro inválido pode destruir o mecanismo de gerenciamento de memória.

Alocação Dinâmica

free()

```
1  #include<stdlib.h>
2  #include<stdio.h>
3  #include<string.h>
4  int main () {
5      char *str[tam];
6      int i, tam = 3;
7      for (i=0; (i < tam); i++){
8          if ((str[i] = malloc(128)) == NULL){
9              printf("Erro de alocacao"); exit(1);
10         }
11         gets(str[i]); puts(str[i]);
12     }
13     for (i=0; i<tam; i++) {
14         free(str[i]);
15     }
16     return(EXIT_SUCCESS);
17 }
```

Alocação Dinâmica

realloc()

```
void *realloc(void *ptr, size_t size);
```

- A função `realloc()` modifica o tamanho da memória previamente alocada apontada por `prt` para aquele especificado por `size`;
- O valor de `size` pode ser maior ou menor que o original;
- Um ponteiro para o bloco de memória é devolvido porque `realloc()` pode precisar mover o bloco para aumentar o seu tamanho;

...

Alocação Dinâmica

realloc()

```
void *realloc(void *ptr, size_t size);
```

...

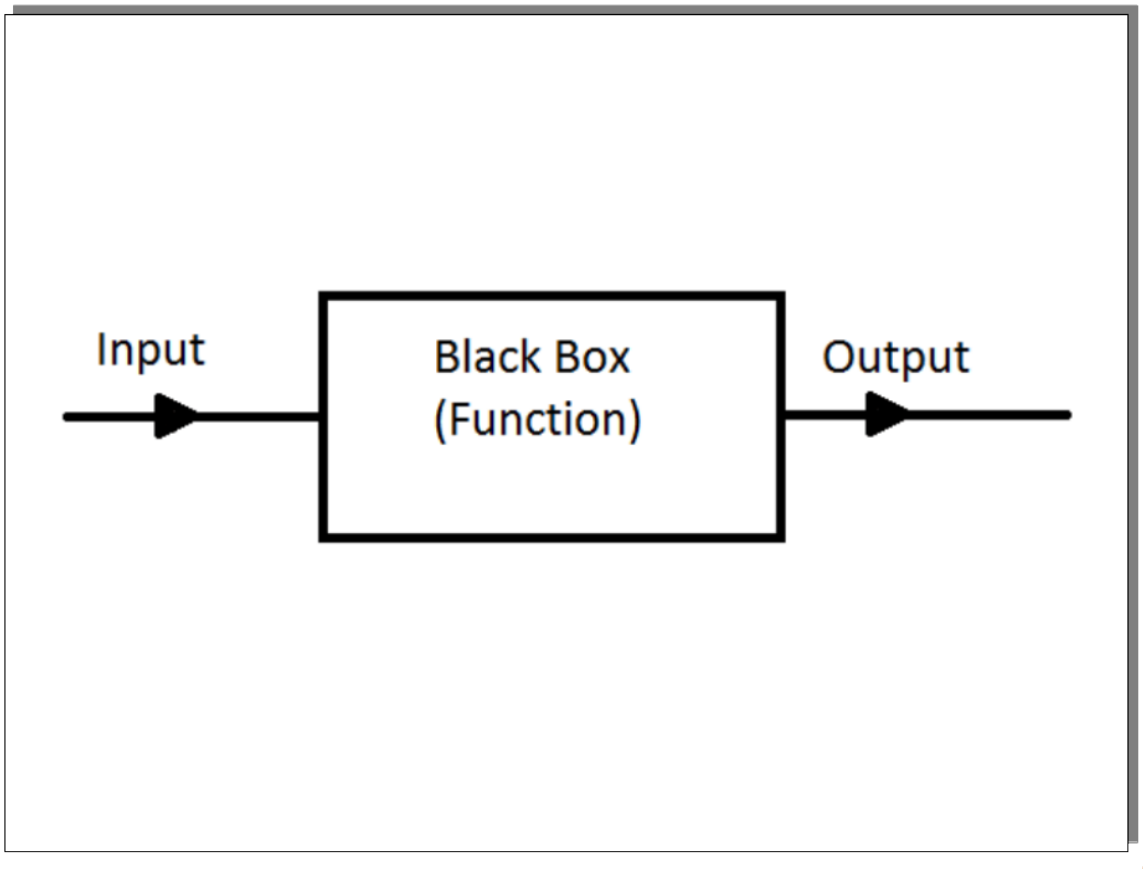
- Se precisar mover o bloco, o conteúdo do bloco antigo é copiado no novo bloco, assim nenhuma informação é perdida;
- Se `size` é zero, a memória apontada por `ptr` é liberada;
- Se não há memória livre suficiente no *heap* para atender ao pedido, devolve-se um ponteiro nulo e o bloco original é deixado inalterado.

Alocação Dinâmica

realloc()

```
1  #include<stdlib.h>
2  #include<stdio.h>
3  #include<string.h>
4  int main () {
5      char *p;
6      if ((p = malloc(23)) == NULL) {
7          printf("Erro de alocacao"); exit(1);
8      }
9      strcpy(p, "isso sao 22 caracteres");
10     p = realloc(p,24);
11     if (!p) {
12        printf("Erro de alocacao"); exit(1);
13     }
14     strcat(p,"."); printf(p);
15     free(p);
16     return(0);
17 }
```

Funções



Funções

Conceito

Uma *rotina* é um bloco organizado de código-fonte elaborado para realizar uma única, e bem definida, tarefa no contexto de um programa ou de um conjunto de programas.

Se a rotina, ao concluir sua tarefa, nada retornar ao corpo de programa (ou outra rotina) que a invocou, então ela é chamada de *rotina simples*, *subrotina* ou *procedimento*.

Se a rotina retorna ao corpo de programa (ou outra rotina) um determinado valor (de qualquer tipo), então ela é chamada de *função*, por assemelhar-se ao conceito matemático de função.

Funções

Conceito

Na linguagem $\mathbb{C}$ todas as rotinas são consideradas como *funções*.

Apesar disso, há funções que *retornam um valor* (`int`, `float`, etc.) e outras que *não retornam nada*.

Para contornar este "*problema*", a linguagem $\mathbb{C}$ tem um tipo denominado de `void` – que simboliza "nada".

Uma função que retornar `void`, em verdade, não retorna nada para sua chamadora.

Funções

Por que dividir um programa em funções?

Funções

Por que dividir um programa em funções?

Há diversas vantagens advindas da *divisão* do código do programa em funções, dentre elas:

- facilitar a construção/manutenção de código;
- minimizar a possibilidade da ocorrência de erros de programação, pois se trabalha com *peças menores* de código;
- possibilitar a reutilização de código anteriormente escrito e testado;

...

Funções

Formato Geral

Em $\mathbb{C}$, o formato geral de uma função é:

```
1 tipo nomeFuncao (tipo1 nome1, tipo2 nome2, ..., tipoN nomeN ) {
2       //
3       // declaracao das variaveis locais da funcao
4       //
5       // corpo da funcao -- com ou sem um retorne,
6       // dependendo do caso
7       //
8 }
```

Funções

Formato Geral

Habitualmente uma *função* recebe uma lista de argumentos:

```
nome1, nome2, ..., nomeN
```

e executa comandos que utilizam estes argumentos para realizar uma tarefa bem definida retornando, ou não, um resultado para a função chamadora.

Funções

Formato Geral

- A lista de argumentos, também chamados de parâmetros, é uma lista de variáveis, separadas por vírgulas, com seus respectivos tipos associados;
- Não é possível usar uma única definição de tipo para várias variáveis;
- A lista de argumentos pode ser vazia, ou seja, a função não receber nenhum argumento;

...

Funções

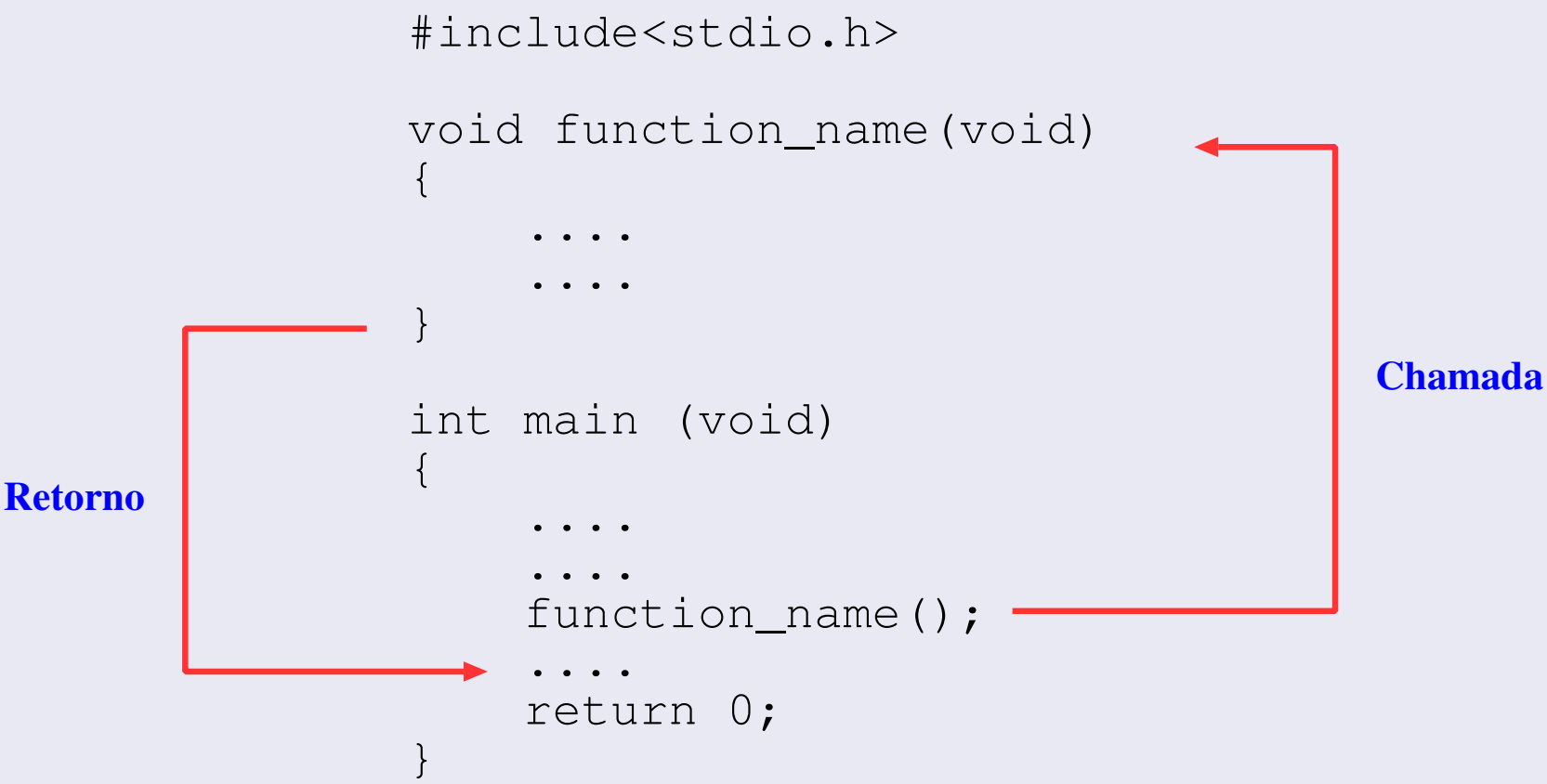
Formato Geral

...

- O nome da função pode ser qualquer identificador válido;
- O `tipo` (que aparece antes do nome da função) especifica o tipo do resultado que será devolvido ao final da execução da função;
- O tipo `void` pode ser usado para declarar funções que não retornam valor algum – rotinas.

Funções

Chamada em $\mathbb{C}$



Funções

Retorno em $\mathbb{C}$

- Há basicamente três maneiras de terminar a execução de uma função X:

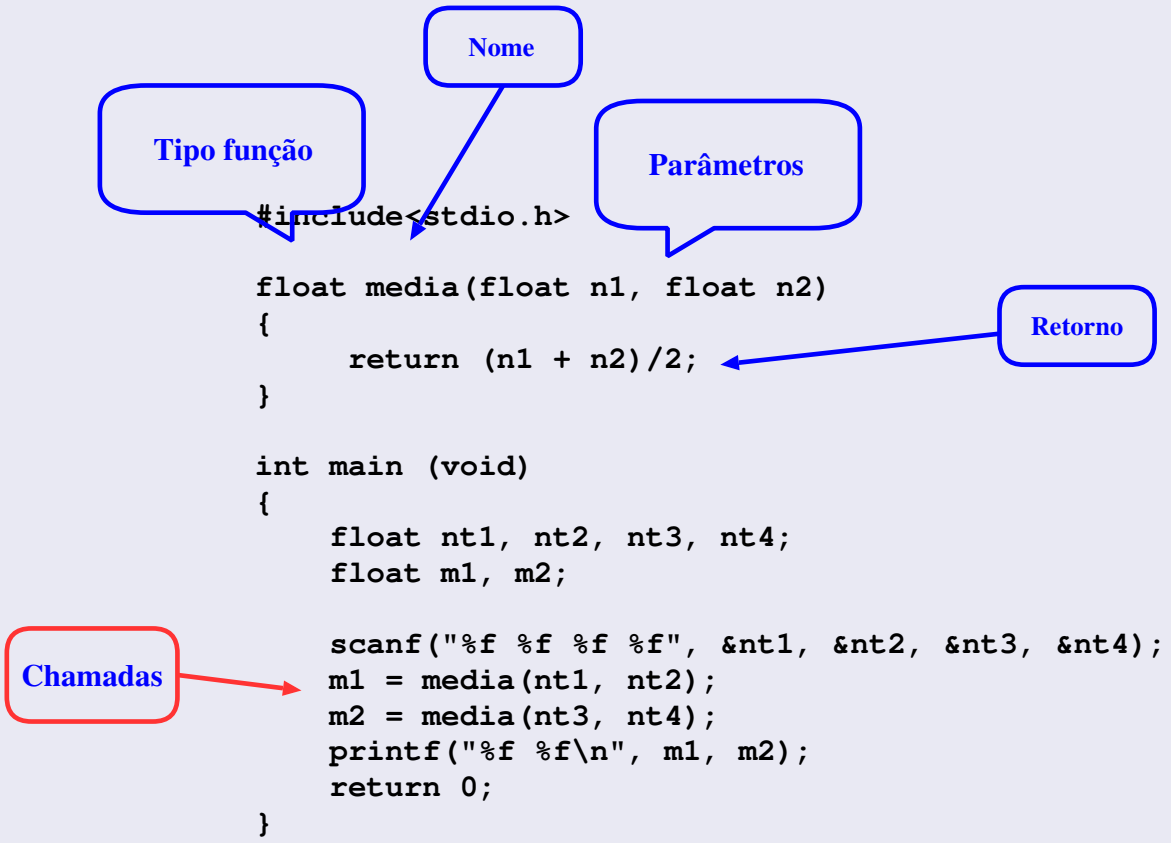
  (1) com `return` para retornar para aquela função que chamou X;

  (2) com `return <expressão>` para retornar o valor da *expressão* para aquela que chamou X;

  (3) atingindo-se, durante a execução, a declaração que indica o término do corpo da função X, desde que ela seja do tipo `void`, ou seja, não retorna *nada* para a chamadora.

Funções

Exemplo 01




```
#include<stdio.h>

float media(float n1, float n2)
{
     return (n1 + n2)/2;
}

int main (void)
{
    float nt1, nt2, nt3, nt4;
    float m1, m2;

    scanf("%f %f %f %f", &nt1, &nt2, &nt3, &nt4);
    m1 = media(nt1, nt2);
    m2 = media(nt3, nt4);
    printf("%f %f\n", m1, m2);
    return 0;
}
```

Funções

Notas

- É importante notar que o nome da função pode aparecer em qualquer lugar onde o nome de uma variável apareceria;
- Além disso os tipos e o número de parâmetros que aparecem na declaração da função e na sua chamada devem estar na mesma ordem e ser de tipos equivalentes;
- Os nomes das variáveis nos programas que usam uma função podem ser diferentes dos nomes usados na definição da função.

Funções

Exemplo 02

```
1  #include<stdio.h>
2  int fatorial (int n) {
3      int f = 1;
4      for ( ; (n > 1); n--) {
5          f *= n;
6      }
7      return(f);
8  }
9  int main (void) {
10     int n, p, c;
11     n = 5; p = 3;
12     c = fatorial(n) / (fatorial(p) * fatorial(n-p));
13     printf("%d \n", c);
14     return (0);
15 }
```

Funções

Protótipos

O padrão ANSI $\mathbb{C}$ estendeu a declaração da função para permitir que o compilador faça uma verificação mais rígida da compatibilidade entre os tipos que a função espera receber e aqueles que lhe são fornecidos.

Protótipos de funções ajudam a detectar erros antes que eles ocorram, ou seja, em tempo de compilação, impedindo que funções sejam chamadas com argumentos inconsistentes durante a execução do programa.

Funções

Protótipos

A forma geral de definição de um protótipo é a seguinte:

```
1 //
2 // Observe o "ponto e virgula" ao final da declaracao.
3 //
4 tipo nomeFuncao (tipo1 nome1, tipo2 nome2, ..., tipoN nomeN);
```

Funções

Exemplo 03

```
1  #include<stdio.h>
2  //
3  // Prototipo da funcao "soma"...
4  //
5  int soma (int a, int b);
6
7  int main() {
8      int a = 5, b = 9;
9      printf("%d\n", soma(a,b));
10     return(0);
11 }
12 //
13 // Corpo, implementacao, da funcao soma...
14 //
15 int soma(int a, int b) {
16     return (a+b);
17 }
```

Funções

Exemplo 04

```
1  #include<stdio.h>
2  //
3  // Ou assim...
4  //
5  int soma (int a, int b);
6
7  int soma(int a, int b) {
8      return (a+b);
9  }
10
11 int main() {
12     int a = 5, b = 9;
13     printf("%d\n", soma(a,b));
14     return(0);
15 }
```

Passagem de Parâmetros

```
#include <stdio.h>

int fatorial (int n)
    int f = 1;
    for (int i = 1; (i <= n); i++) {
        f *= i;
    }
    return(f);
}

int main() {

    int variavel = 5;

    printf("%i\n",fatorial(variavel));
    return(0);
}
```

parâmetro formal

argumento da função

Passagem de Parâmetros

Conceito

- As variáveis que aparecem na lista de parâmetros de uma função são chamadas de *parâmetros formais* (PFs);
- Os PFs são criados no início da execução da função e destruídos ao final dela;
- Parâmetros podem ser passados para funções de duas maneiras em $\mathbb{C}$: *por valor* ou *por referência*.

Passagem de Parâmetros

Passagem por Valor

Uma cópia do valor do argumento é passado para a função.

Assim, a função, ao fazer modificações no parâmetro, não estará alterando o valor original, já que ela possui apenas uma *cópia* dele.

Enfatizando: as alterações locais nos parâmetros não se refletem nas variáveis correspondentes da função chamadora.

Passagem de Parâmetros

Exemplo 07

```
1  #include<stdio.h>
2  float eleva(float a, int b) {
3        float res = 1.0;
4        for ( ; (b > 0); b--) res *= a;
5        return(res);
6  }
7  int main() {
8      float numero;
9      int potencia;
10     puts("Entre com um numero"); scanf("%f", &numero);
11     puts("Entre com a potencia"); scanf("%d", &potencia);
12     printf("%f Elevado a %d e igual a %f\n",
13           numero, potencia, eleva(numero, potencia));
14     return(0);
15 }
```

Passagem de Parâmetros

Passagem por Valor

O que será impresso se fizermos `numero = 2` e `potencia = 3`?

Passagem de Parâmetros

Exemplo 08

```
1  #include <stdio.h>
2  void trocar(int a, int b) {
3      int temp;
4      temp = a; a = b; b = temp;
5  }
6  int main(int argc, char *argv[]) {
7      int a = 10, b = 20;
8      trocar(a, b);
9      printf("a = %d, b = %d\n", a, b);
10     return (0_;
11 }
```

Passagem de Parâmetros

Passagem por Valor

O que será impresso pelo programa principal do exemplo anterior?

Passagem de Parâmetros

Passagem por Referência

Neste caso, o que é passado para a função é o endereço do parâmetro e, portanto, a função recebedora pode modificar o valor do argumento da função chamadora.

Para a passagem de parâmetros por referência é necessário o uso de *ponteiros*, também chamados de *apontadores*.

Enfatizando: as alterações locais nos parâmetros SE REFLETEM nas variáveis correspondentes da função chamadora.

Passagem de Parâmetros

Passagem por Referência

Em $\mathbb{C}$, utiliza-se o artifício de passar como argumento o endereço da variável.

Para indicar que será passado o endereço do argumento, usa-se o mesmo tipo que usado para declarar um variável que guarda um endereço:

```
tipo nomeFuncao (tipo1 *parâmetro1, ..., tipoN
*parâmetroN)
```

Passagem de Parâmetros

Passagem por Referência

- Um endereço de uma variável passado como parâmetro não é muito útil. Para acessar o valor de uma variável apontada por um endereço usa-se o operador * (asterisco);
- Ao preceder o nome de uma variável (que contém um endereço) com este operador, obtém-se o equivalente à variável armazenada no endereço em questão.

Passagem de Parâmetros

Exemplo 09

```
1 void troca(int *x, int *y) {
2       int aux;
3
4       aux = *x;
5       *x = *y;
6       *y = aux;
7 }
```

Passagem de Parâmetros

Passagem por Referência

- Outra forma de conseguirmos alterar os valores de variáveis externas às funções é usando variáveis *globais*...
- Nesta abordagem usamos variáveis *globais* ao invés de parâmetros e de valores de retorno...
- Porém, nesta caso, estamos negando uma das principais vantagens de se usar funções: o reaproveitamento de código.

Passagem de Parâmetros

Passagem por Referência

Em $\mathbb{C}$, vetores e matrizes são um caso especial, pois são exceção à regra de que nomes de variáveis como parâmetros indicam passagem *por valor*.

O *nome* de um vetor/matriz corresponde ao endereço do primeiro elemento daquele `array`.

Quando esse nome é passado como parâmetro, em verdade, o endereço do primeiro elemento é que é passado – passagem por referência.

Passagem de Parâmetros

Passagem por Referência

- Fundamentalmente há três maneiras de declarar um vetor/matriz como um parâmetro de uma função.

PRIMEIRA: ele é declarado segundo as regras de declaração de uma variável do tipo `vetor`.

Passagem de Parâmetros

Exemplo 10

```
1 #include <stdio.h>
2 #define DIM 80
3
4 int conta (char v[DIM], char c);
```

Passagem de Parâmetros

Exemplo 10

```
int main() {
    char c, linha[DIM];
    size_t  bufferSize = DIM;
    int maiusculas[26], minusculas[26];
    gets(linha);
    for (c = 'a'; c <= 'z'; c++) minusculas[c-'a'] = conta(linha, c);
    for (c = 'A'; c <= 'Z'; c++) maiusculas[c-'A'] = conta(linha, c);
    for (c = 'a'; c <= 'z'; c++)
        if (minusculas[c-'a'])
            printf("%c apareceu %d vezes\n", c, minusculas[c-'a']);
    for (c = 'A'; c <= 'Z'; c++)
        if (maiusculas[c-'A'])
            printf("%c apareceu %d vezes\n", c, maiusculas[c-'A']);
    return(0);
}
```

Passagem de Parâmetros

Exemplo 11

```c
#include <stdio.h>
#define DIM 80

int conta (char v[DIM], char c) {
    int i = 0, vezes = 0;
    while (v[i] != '\0') {
        if (v[i++] == c) {
            vezes++;
        }
    return(vezes);
}
```

Passagem de Parâmetros

Passagem por Referência

- Fundamentalmente há três maneiras de declarar um vetor/matriz como um parâmetro de uma função.

SEGUNDA: apenas o endereço do vetor é passado: o parâmetro é declarado como um vetor sem dimensão – é perfeitamente possível porque a função somente precisa receber o endereço onde se encontra o vetor.

Como $\mathbb{C}$ não confere os limites de vetores, a função precisa do endereço inicial do vetor e uma maneira de descobrir o final do vetor.

Passagem de Parâmetros

Exemplo 12

```
1  #include<stdio.h>
2  #define DIM 6
3
4  void Le_vetor (int v[], int tam);
5  void Imprime_vetor (int v[], int tam);
6  void Inverte_vetor (int v[], int tam);
7
8  int main() {
9      int v[DIM];
10
11     Le_vetor(v, DIM);
12     Imprime_vetor (v, DIM);
13     Inverte_vetor (v, DIM);
14     Imprime_vetor (v, DIM);
15     return(0);
16 }
```

Passagem de Parâmetros

Exemplo 12

```
1  void Le_vetor (int v[], int tam) {
2      int i;
3      for (i = 0; (i < tam); i++) { printf("%d = ? ", i); scanf("%d", &v[i]); }
4  }
5  void Imprime_vetor (int v[], int tam) {
6      int i;
7      for (i = 0; (i < tam); i++) { printf("%d = %d\n", i, v[i]); }
8  }
9  void Inverte_vetor (int v[], int tam) {
10     int i, temp;
11     for (i = 0; (i < tam/2); i++) {
12         temp = v[i];
13         v[i] = v[tam-i-1];
14         v[tam-i-1] = temp;
15     }
16 }
```

Passagem de Parâmetros

Passagem por Referência

- Fundamentalmente há três maneiras de declarar um vetor/matriz como um parâmetro de uma função.

TERCEIRA: é necessário o emprego de *ponteiros*.

Passagem de Parâmetros

Exemplo 13

```
1  #include <stdio.h>
2
3  void display (int *num, int tam);
4
5  void display (int *num, int tam) {
6      int i;
7      for (i = 0; (i < tam); i++) {
8          printf("%d\n",*(num + i));
9      }
10 }
```

Escopo de Variáveis

```
#include <stdio.h>

int variavel;

int main() {

    int i;
    variavel = 5;
    for (i=0;(i<10);i++) {
       // codigo
    }
    {
       int j;

       j = 18;
    }
    return(0);
}
```

Escopo de Variáveis

```
#include <stdio.h>

int variavel;

int main() {

    int i;
    variavel = 5;
    for (i=0;(i<10);i++) {
       // codigo
    }
    {
       int j;

       j = 18;
    }
    return(0);
}
```

Variável *global* “*variavel*”

Variável *local* “*i*”

Variável *local* (de bloco) “*j*”

Escopo de Variáveis

Conceito

Num programa, variáveis podem ser definidas para serem usadas somente dentro de uma função particular, ou, ao contrário, precisem ser acessíveis a diversas funções diferentes;

Por esta razão temos que apresentar os locais onde as variáveis de um programa podem ser definidas e, a partir destes locais, podermos inferir onde elas estarão disponíveis...

o que é captado pelo conceito de escopo de uma variável.

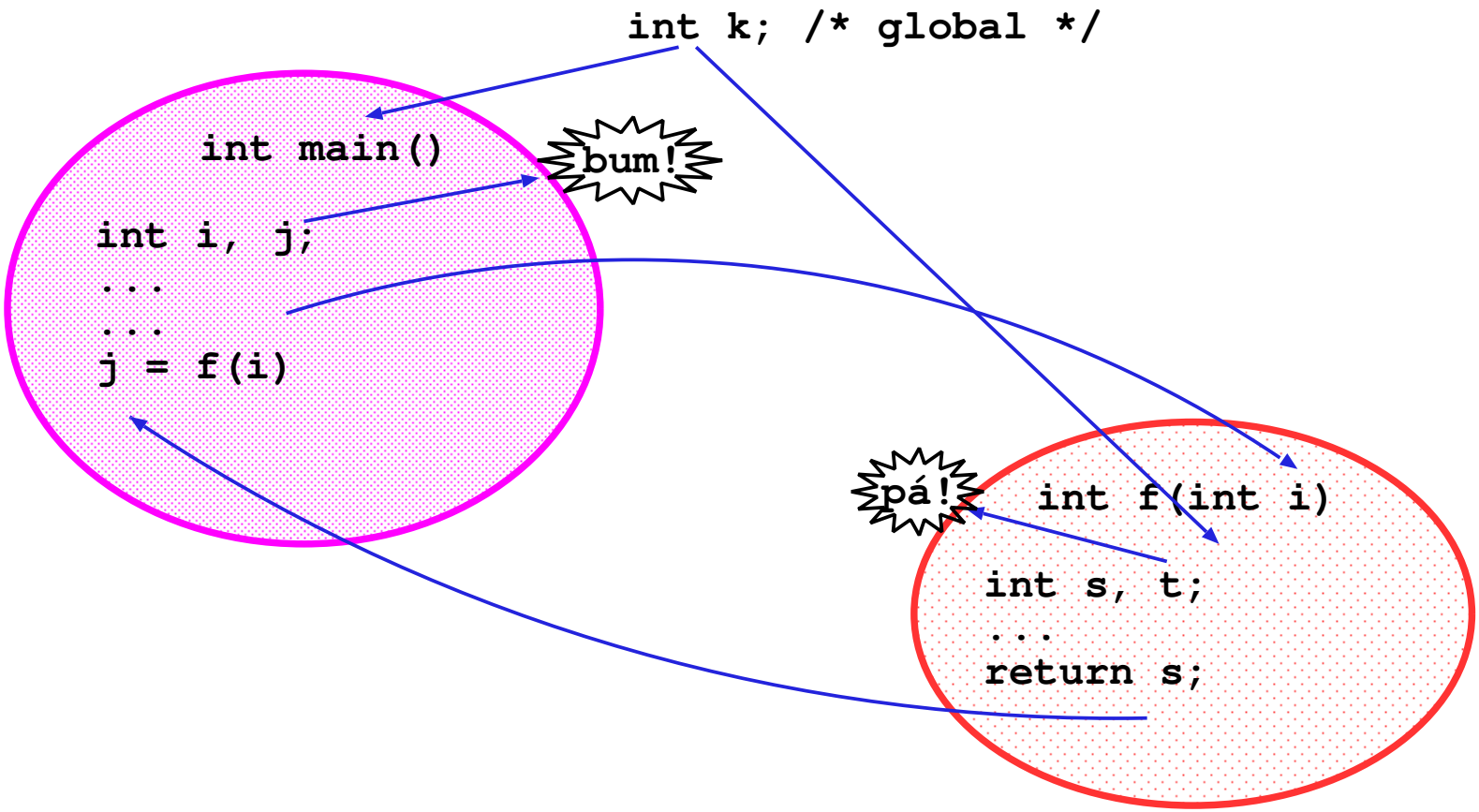
Escopo de Variáveis

Conceito

Em $\mathbb{C}$, variáveis podem ser declaradas basicamente em três locais:

- fora de todas as funções – variáveis globais;
- dentro de funções – variáveis locais;
- na lista de parâmetros das funções – parâmetros formais.

Escopo de Variáveis



Escopo de Variáveis

(1) Variáveis Globais

Em $\mathbb{C}$, as variáveis *globais*...

- são definidas fora de qualquer função e são, portanto, disponíveis para qualquer função do programa;
- podem servir como meio de comunicação entre funções, uma maneira de transferir valores entre elas.
  Assim, se duas funções devem partilhar dados, mas uma não chama diretamente a outra, uma variável *global* pode ser usada para transferir dados entre elas.

Escopo de Variáveis

(2) Variáveis Locais

Em $\mathbb{C}$, as variáveis *locais*...

- são aquelas declaradas dentro de uma função ou de um bloco de comandos;
- somente podem ser referenciadas dentro da função/bloco onde foram declaradas;
- são invisíveis para outras funções do mesmo programa.
- passam a existir no início da execução da função/bloco de comandos onde foram definidas e são destruídas ao final da execução desta(e);

Escopo de Variáveis

Exemplo 05

```
1  #include <stdio.h>
2
3  int i; // variavel global
4
5  void soma1(void) {
6      i += 1;
7      printf("Funcao soma1: i = %d\n", i);
8  }
9
10 void sub1(void) {
11     int i = 10; // variavel local
12     i -= 1;
13     printf("Funcao sub1: i = %d\n", i);
14 }
15 //
16 // continua...
17 //
```

Escopo de Variáveis

Exemplo 05

```
1 int main (int argc, char *argv[]) {
2     i = 0;
3     soma1();
4     sub1();
5     printf("Funcao main: i = %d\n", i);
6     return(0);
7 }
```

Escopo de Variáveis

Exemplo 05

O resultado da execução deste programa é o seguinte:

- impressão de `Funcao soma1:  i = 1`
- impressão de `Funcao sub1:  i = 9`
- impressão de `Funcao main:  i = 1`

Escopo de Variáveis

Exemplo 05

Note que:

- a variável global `i` recebe o valor `0` (zero) no início de `main`;
- `soma1`, ao executar, incrementa em uma unidade a variável global `i`;
- `sub1`, ao executar, define uma variável local também chamada de `i` – a alteração que realiza afeta somente esta variável local;
- a função `main` imprime o valor final da variável global `i`.

Escopo de Variáveis

Exemplo 06

```
1  #include <stdio.h>
2  void pares(void) {
3      int i;
4      for (i = 2; (i <= 10); i += 2) { printf("%d: ", i); }
5  }
6  void impares(void) {
7      int i;
8      for (i = 3; (i <= 11); i += 2) { printf("%d: ", i); }
9  }
10 int main(int argc, char *argv[]) {
11     pares();
12     printf("\n");
13     impares();
14     return (0);
15 }
```

Escopo de Variáveis

(3) Parâmetros Formais

As *variáveis locais* que aparecem na lista de parâmetros de uma função são chamadas de parâmetros formais.

Por isso escopo destes parâmetros é idêntico ao das variáveis locais da função.

Indicações de Bibliografia

- Obras indicadas nas bibliografias da disciplina (básica, complementar e sugerida);
- Guias e/ou manuais da linguagem $\mathbb{C}$;
- Guias e/ou manuais do compilador usado por você;
- Videoaulas disponíveis no YouTube a respeito dos temas abordados.

Referências Bibliográficas

• Obras indicadas nas bibliografias da disciplina (básica, complementar e sugerida);

Fim

????, by Mark Kostabi