

Modularização e passagem de parâmetros

Prof. Joaquim Quinteiro Uchôa
Profa. Juliana Galvani Gregghi
Profa. Marluce Rodrigues Pereira
Profa. Paula Christina Cardoso
Prof. Renato Ramos da Silva



Roteiro

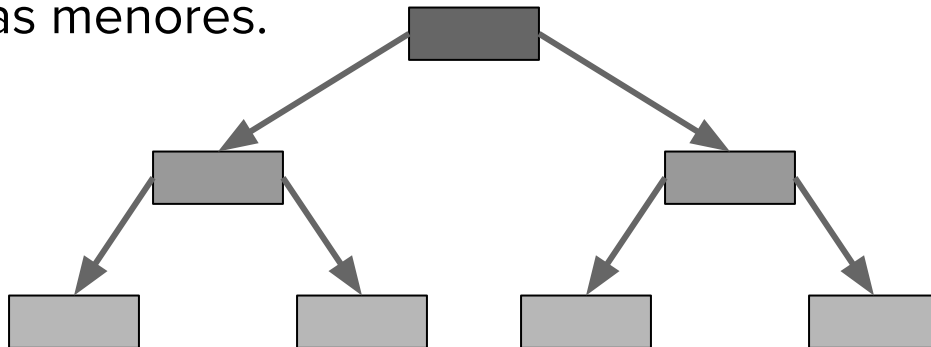
- Contextualização
- Subprogramas
- Fluxo de execução
- Escopo de variáveis
- Passagem de parâmetros
- Passagem por ponteiro e passagem de vetor como parâmetro

Contextualização

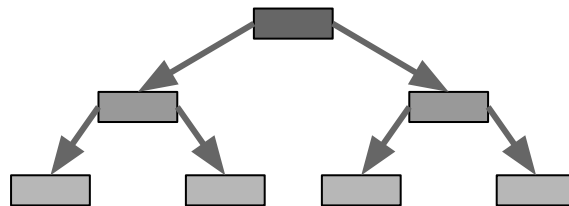


Contextualização

- Necessidade
 - Problemas simples podem ser resolvidos diretamente em um único algoritmo. Contudo, na prática, a maioria das tarefas requerem um planejamento mais eficiente para uma solução, usualmente por meio da decomposição do problema original em problemas menores.



Contextualização



- Necessidade

- Conforme uma tarefa cresce e se torna mais complexa, surgem uma série de situações distintas (subproblemas) que precisam ser resolvidas.
- Cada subproblema pode ser resolvido isoladamente por meio de um algoritmo particular.
- No final, espera-se que as soluções (algoritmos) parciais de cada subproblema possam ser combinadas e juntas sejam capazes de resolver o problema original dado.

Contextualização

- Necessidade
 - Dividir um algoritmo, que envolve um problema grande, em subalgoritmos (soluções parciais), proporciona uma série de vantagens, sendo as principais:
 - Redução da complexidade do código;
 - Redução do tamanho do código;
 - Modificações podem ser feitas apenas nos subalgoritmos, com reflexo no programa como um todo.

Contextualização

- Exemplo

- Problema: imagine que três dos cinco moradores de uma república acabaram de se formar e agora você terá que arrumar novos companheiros de moradia. Para divulgar as vagas, você pensou em calcular a média geral dos gastos da casa. Dessa forma, você decidiu calcular a média dos gastos nos últimos 5 meses com: (1) energia elétrica; (2) água; e (3) despesas com materiais de limpeza.
- Para te ajudar nesta tarefa, você resolveu implementar um programa para fazer todos os cálculos necessários. Em um primeiro momento de projeto, você elaborou o algoritmo em descrição narrativa a seguir.

Contextualização

Algoritmo 1: Média geral de gastos

1. Leia os valores gastos com energia nos últimos 5 meses
2. Calcule a média de gastos com energia
3. Leia os valores gastos com água nos últimos 5 meses
4. Calcule a média de gastos com água
5. Leia os valores gastos com materiais de limpeza nos últimos 5 meses
6. Calcule a média de gastos com materiais de limpeza
7. Informe a média de gastos com energia, água e materiais de limpeza

Contextualização

- Exemplo

- O problema do Algoritmo 1 é que ele geraria um programa extenso, uma vez que ele precisa resolver diversas tarefas (cálculo de três diferentes médias).
- Sendo assim, uma ideia mais organizada de solução seria dividi-lo em subproblemas e resolver cada um destes subproblemas isoladamente, como apresentado a seguir.



Algoritmo 2.1: Média de gastos com energia

1. Leia os valores gastos com energia nos últimos 5 meses
2. Calcule a média de gastos com energia
3. Informe a média de gastos com energia

Algoritmo 2.2: Média de gastos com água

1. Leia os valores gastos com água nos últimos 5 meses
2. Calcule a média de gastos com água
3. Informe a média de gastos com água

Algoritmo 2.3: Média de gastos com materias de limpeza

1. Leia os valores gastos com materiais de limpeza nos últimos 5 meses
2. Calcule a média de gastos com materiais de limpeza
3. Informe a média de gastos com materiais de limpeza

Contextualização

- Exemplo

- Embora o Algoritmo 2 (parte 1, parte 2 e parte 3) seja mais organizado, ele ainda não é muito eficiente, uma vez que estamos realizando uma mesma tarefa (cálculo de média) diversas vezes. Se analisarmos na prática, estamos apenas trocando o tipo de média a ser calculada em cada solução parcial (se é média do consumo com energia, água ou com materiais de limpeza).
- Sendo assim, podemos melhorar nossa solução, como ilustrado no Algoritmo 3 (parte 1 e parte 2) a seguir.



Algoritmo 3.1: Controle geral

1. Utilize o subalgoritmo que calcula a média para a despesa com energia
2. Utilize o subalgoritmo que calcula a média para a despesa com água
3. Utilize o subalgoritmo que calcula a média para a despesa com materias de limpeza



Algoritmo 3.2: Média de gastos com uma despesa qualquer

1. Leia os valores gastos com a despesa nos últimos 5 meses
2. Calcule a média de gastos com aquela despesa
3. Informe a média de gastos com aquela despesa

Contextualização

- Exemplo

- Note que esta última versão, Algoritmo 3, é melhor do que as anteriores.
 - Considere, por exemplo, que fosse necessário calcular médias de consumo com outras despesas (gastos com internet, gastos com alimentação, etc.) além das três já tratadas.
 - Para adequarmos o Algoritmo 3 a estas novas médias, bastaria modificarmos as ações na Parte 1 do algoritmo, acrescentando novos usos da Parte 2, responsável por calcular uma média de uma despesa qualquer.
 - Nos Algoritmos 1 e 2, esta adequação seria bem mais trabalhosa.

Contextualização

- Na aula de hoje, aprendemos como construir subalgoritmos. A construção de subalgoritmos é chamada em programação de modularização.
- De modo geral, a modularização envolve o uso de técnicas como subprogramas, métodos, rotinas, módulos, unidades, bibliotecas, entre outras.
- Neste curso, iremos nos focar especificamente no uso de dois tipos de subprogramas, conhecidos como funções e procedimentos.

Subprogramas



Subprogramas

- **Definição:** um subprograma é um conjunto de instruções responsável por realizar uma tarefa **específica** dentro de um programa maior.
 - A esse conjunto de instruções é associado um nome (identificador).
 - Para usar (executar) um subprograma, basta fazer referência ao seu nome (identificador).
 - Em programação, ao usar um subprograma, utiliza-se a expressão **fazer uma chamada** para o subprograma.
 - Uma vez que um subprograma seja criado, o mesmo pode ser executado (chamado) quantas vezes forem necessárias.

Subprogramas

- Subprogramas em C++ (funções ou procedimentos) são definidos de acordo com a seguinte sintaxe:

```
tipo_retorno identificador_subprograma(tipo1 parâmetro1, ..., tipoN parâmetroN)  
{  
    //Corpo (instruções) do subprograma  
    //...  
    return valor_retorno;  
}
```

Vamos agora discutir sobre cada um dos componentes desta sintaxe.

```
tipo_retorno identificador_subprograma(tipo1 parâmetro1, ..., tipoN parâmetroN)  
{  
    //Corpo (instruções) do subprograma  
    //...  
    return valor_retorno;  
}
```

tipo_retorno

- Expressa a natureza da informação computada e retornada pelo subprograma implementado. Por exemplo, se a natureza da informação é **int**, **float**, **char** ou outro tipo de dado qualquer.
- Um subprograma que, por exemplo, calcula a média de uma sequência de números reais e retorna o valor obtido, possuiria como **tipo_retorno** os tipos **float** ou **double**.
- Se um subprograma não retorna nenhuma informação, usa-se o tipo **void** como o **tipo_retorno** do subprograma.

```
tipo_retorno identificador_subprograma(tipo1 parâmetro1, ..., tipoN parâmetroN)  
{  
    //Corpo (instruções) do subprograma  
    //...  
    return valor_retorno;  
}
```

Identificador_subprograma

- Representa o nome do subprograma.
- Segue as mesmas regras de sintaxe para definição de identificadores de variáveis.
- Usado para fazer a chamada do subprograma implementado.

```
tipo_retorno identificador_subprograma(tipo1 parâmetro1, ..., tipoN parâmetroN)
{
    //Corpo (instruções) do subprograma
    //...
    return valor_retorno;
}
```

tipo₁ parâmetro₁, ..., tipo_N parâmetro_N

- Expressa a lista de parâmetros do subprograma (chamados parâmetros **formais**).
 - A lista de parâmetros deve obrigatoriamente aparecer entre símbolos de parênteses, (e).
- A lista de parâmetros denota todas as informações necessárias para que o subprograma possa realizar o seu processamento.
- Cada parâmetro deve ser separado dos demais pelo símbolo de vírgula.
- Caso um subprograma não precise de nenhum parâmetro, ainda deve-se utilizar os símbolos de parênteses (apenas os parênteses).

```
tipo_retorno identificador_subprograma(tipo1 parâmetro1, ..., tipoN parâmetroN)
{
    //Corpo (instruções) do subprograma
    //...
    return valor_retorno;
}
```

tipo₁ parâmetro₁, ..., tipo_N parâmetro_N

- Cada parâmetro é definido por um par de componentes, a saber:
 - Tipo do parâmetro (**tipo₁, ..., tipo_N**): natureza da informação manipulada pelo parâmetro. Por exemplo, se a natureza da informação é **int**, **float**, **char** ou outro tipo de dado qualquer.
 - Identificador do parâmetro (**parâmetro₁, ..., parâmetro_N**): nome do parâmetro que armazena a informação a ser manipulada. Em termos práticos, é um nome de uma variável.

```
tipo_retorno identificador_subprograma(tipo1 parâmetro1, ..., tipoN parâmetroN)  
{  
    //Corpo (instruções) do subprograma  
    //...  
    return valor_retorno;  
}
```

//Corpo (instruções) do subprograma
//...

- Conjunto de instruções que representa o processamento realizado pelo subprograma implementado.
- Sempre que o nome (identificador) de um subprograma for chamado, o corpo de instruções daquele subprograma será executado.
- Pode conter qualquer tipo de instrução ou comando, como por exemplo: declaração de variáveis, estruturas condicionais ou de repetição, chamadas a outros subprogramas, etc.
- Delimitado obrigatoriamente pelos símbolos de chaves, { e }.

```
tipo_retorno identificador_subprograma(tipo1 parâmetro1, ..., tipoN parâmetroN)
{
    //Corpo (instruções) do subprograma
    //...
    return valor_retorno;
}
```

return valor_retorno;

- O comando **return** associa um valor qualquer obtido dentro de um subprograma ao identificador do próprio subprograma. Desta forma, quando uma chamada de subprograma é realizada, recupera-se através do **identificador_subprograma** o valor da informação expressa pelo **valor_retorno**.
- Um subprograma pode apresentar vários comandos do tipo **return** em seu corpo. Contudo, a cada chamada, apenas um deles é efetivamente executado (será executado o primeiro comando **return** seguindo, em tempo de execução, o fluxo de ações expresso no corpo do subprograma).

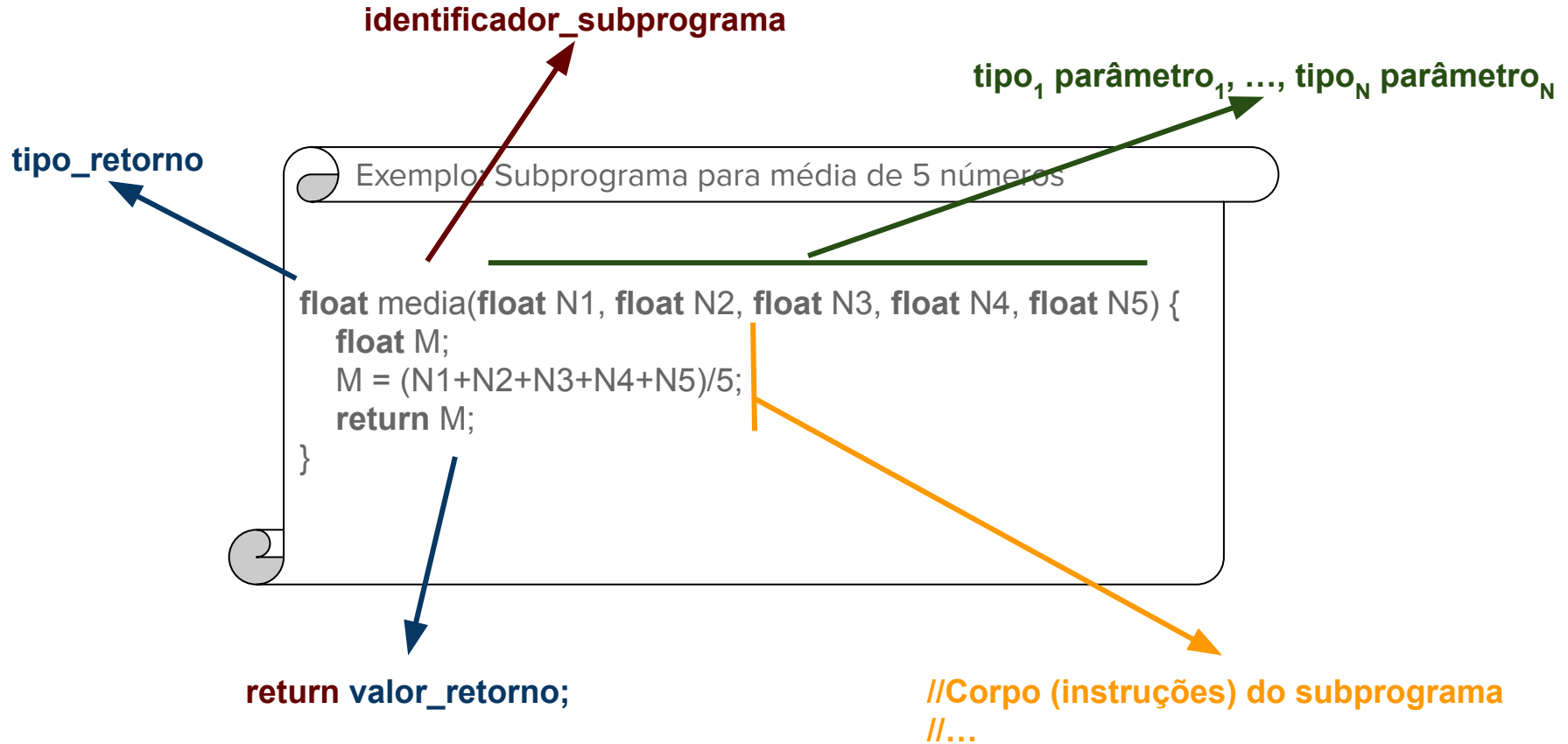
Subprogramas

- Exemplo

- Problema: Construir um subprograma em C++ que calcule a média aritmética de cinco números reais quaisquer.

Exemplo: Subprograma para média de 5 números

```
float media(float N1, float N2, float N3, float N4, float N5) {  
    float M;  
    M = (N1+N2+N3+N4+N5)/5;  
    return M;  
}
```

Subprogramas

- Exemplo
 - No exemplo do subprograma que calcula a média de cinco números reais, construímos no corpo do subprograma o conjunto de instruções que resolve o problema de cálculo da média para quaisquer cinco valores, representados pelas variáveis **N1**, **N2**, **N3**, **N4** e **N5**.
 - Para utilizar o subprograma construído, devemos fazer uma chamada ao subprograma. Para isto, basta escrever o identificador do subprograma, seguido pela lista de valores (chamados **parâmetros reais**) para a qual se deseja calcular a média.

Subprogramas

- Assim teríamos, por exemplo:

Exemplo: Subprograma para média de 5 números

```
float media(float N1, float N2, float N3, float N4, float N5) {  
    float M;  
    M = (N1+N2+N3+N4+N5)/5;  
    return M;  
}
```

Declaração do
subprograma **media**

```
media(200,180,210,197,207);
```

Exemplo de chamada
do subprograma **media**

Subprogramas

- Exemplo:

```
float media(float N1, float N2, float N3, float N4, float N5) {  
    float M;  
    M = (N1+N2+N3+N4+N5)/5;  
    return M;  
}  
  
int main() {  
    cout << media(200,180,210,197,207) << endl;  
    return 0;  
}
```

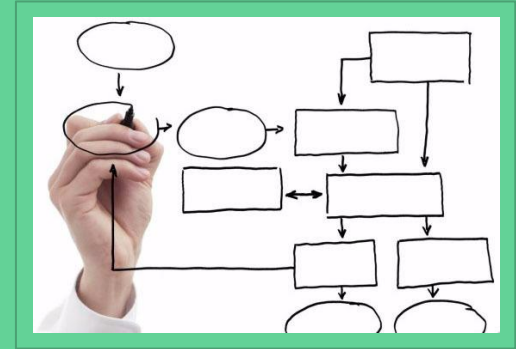


O que será exibido
no comando cout?

Subprogramas

- Chamada de um subprograma
 - Todo subprograma precisa ser declarado antes de ser utilizado.
 - Um subprograma deve ser declarado fora de outros subprogramas.
 - Em C++, a **main** é considerada uma função (subprograma). Portanto, todos os subprogramas criados pelo programador devem ser declarados fora da **main**.
 - Um subprograma pode ser chamado de dentro de outros subprogramas.

Fluxo de execução



Fluxo de execução

- O que acontece quando um subprograma é chamado?
 - Cada argumento da lista de parâmetros é avaliado.
 - Seus tipos são analisados e os valores passados na chamada são atribuídos aos parâmetros formais do subprograma (na ordem na qual foram descritos).
 - O corpo do subprograma é executado.
 - Ao término, o fluxo de execução retorna ao ponto imediatamente após a chamada do subprograma.

Fluxo de execução

- As instruções no corpo do subprograma são executadas até que uma das duas condições a seguir sejam satisfeitas:
 - Se encontre um comando de retorno (**return**).
 - Não existam mais instruções no corpo do subprograma para ser executada.
- O valor do comando **return**, se ele existir, é avaliado e retornado como valor do subprograma ao ponto que chamou sua execução.

Fluxo de execução

- Funções e procedimentos
 - Um subprograma que não retorna, isto é, não produz nenhum valor, tem como retorno o tipo **void**.
 - Em termos de nomenclatura formal, subprogramas com retorno recebem o nome de funções. Enquanto que subprogramas sem retorno recebem o nome de procedimentos.
- A **main()** é uma função (comumente chamada de função principal) responsável por iniciar todos os programas escritos em C++.

Fluxo de execução

- **Observação:** Por que construir um subprograma que não retorna nenhum valor?
 - Dependendo da situação, o importante pode ser a ação colateral do subprograma e não o seu valor de saída.
 - Exemplos:
 - Subprograma que realiza a impressão de uma mensagem.
 - Subprograma responsável por ligar e desligar um componente de hardware.

Fluxo de execução

```
float media(float N1, float N2, float N3, float N4, float N5) {  
    float M;  
    M = (N1+N2+N3+N4+N5)/5;  
    return M;  
}  
  
int main() {  
    cout << media(200,180,210,197,200+7) << endl;  
    return 0;  
}
```

1º.

A execução de um programa C++ sempre começa a partir da função principal.

Fluxo de execução

```
float media(float N1, float N2, float N3, float N4, float N5) {  
    float M;  
    M = (N1+N2+N3+N4+N5)/5;  
    return M;  
}
```

```
int main() {  
    cout << media(200,180,210,197,200+7) << endl;  
    return 0;  
}
```

2º.

Ao fazer uma chamada de um subprograma, cada parâmetro é avaliado e atribuído ao parâmetro formal correspondente.

Neste caso, o valor **200** é atribuído a variável **N1**, **180** a **N2**, **210** a **N3**, **197** a **N4** e **207** a **N5**.

Fluxo de execução

```
float media(float N1, float N2, float N3, float N4, float N5) {  
    float M;  
    M = (N1+N2+N3+N4+N5)/5;  
    return M;  
}  
  
int main() {  
    cout << media(200,180,210,197,200+7) << endl;  
    return 0;  
}
```

3º.

O corpo do subprograma chamado é então executado, até que o subprograma não tenha mais instruções em seu corpo ou um comando de retorno seja encontrado.

Neste caso, o subprograma se encerra ao encontrar o comando de retorno da variável **M**, que no caso deste exemplo, estará armazenando o valor **560.8**.

Fluxo de execução

```
float media(float N1, float N2, float N3, float N4, float N5) {  
    float M;  
    M = (N1+N2+N3+N4+N5)/5;  
    return M;  
}  
  
int main() {  
    cout << media(200,180,210,197,200+7) << endl;  
    return 0;  
}
```

4º.

Ao encerrar o subprograma, o fluxo de execução volta para o ponto no qual o subprograma foi chamado.

Neste caso, o fluxo volta a linha do comando **cout** na função principal. Como é retornado o valor **560.8**, o comando de impressão irá exibir o referido valor na tela do computador.

Escopo de variáveis



Escopo de variáveis

- **Definição:** O escopo de uma variável define o conjunto de sentenças (instruções) no qual a variável é visível (acessível).
- Uma variável é visível em uma sentença se ela pode ser referenciada nesta sentença.
- As regras de escopo de uma linguagem de programação determinam como uma ocorrência em particular de um identificador é associada com uma variável.



Escopo de variáveis

- Uma variável é **local** a uma unidade ou a um bloco de programa se ela for lá declarada.
 - Variáveis declaradas dentro de subprogramas são visíveis apenas dentro do subprograma. Ou seja, são locais ao subprograma.
- Variáveis do programa, isto é, aquelas que não são declaradas dentro de nenhum subprograma, são chamadas de variáveis **globais** e podem ser acessadas em qualquer parte do código.

Escopo de variáveis

- De modo geral, deve-se **evitar** o uso de variáveis globais, pois as mesmas:
 - Deixam os subprogramas menos genéricos.
 - Tornam o código menos legível.
 - Tornam o programa menos confiável (suscetível a falhas).
 - Dificultam a manutenção do código.

Escopo de variáveis

- Em C++, variáveis em escopos diferentes podem possuir identificadores idênticos.
 - É importante observar que, embora seja usado o mesmo nome para estas variáveis, elas se referem a endereços de memória completamente diferentes e, portanto, podem armazenar informações completamente distintas (inclusive de tipos de dados diferentes).

Exemplo subprogramas: Maior de 3

```
#include <iostream>
using namespace std;
```

```
int max(int A, int B) {
    if (A > B) {
        return A;
    } else {
        return B;
    }
}
```

```
int maxTres(int A, int B, int C) {
    int maior = A;
    maior = max(maior, B);
    maior = max(maior, C);
    return maior;
}
```

```
int main(){
    float A, B, C;
    cin >> A >> B >> C;
    cout << maxTres(A,B,C) << endl;
    return 0;
}
```

Exemplo subprogramas: Maior de 3

```
#include <iostream>
using namespace std;
```

```
int max(int A, int B) {
    if (A > B) {
        return A;
    }else {
        return B;
    }
}
```

```
int maxTres(int A, int B, int C) {
    int maior = A;
    maior = max(maior, B);
    maior = max(maior, C);
    return maior;
}
```

```
int main(){
    float A, B, C;
    cin >> A >> B >> C;
    cout << maxTres(A,B,C) << endl;
    return 0;
}
```

Escopos diferentes



Teste o programa com
três números reais e veja
o que acontece.

Passagem de parâmetros



Passagem de parâmetros

- Todo subprograma define um processamento a ser realizado.



PROCESSAMENTO

- Esse processamento depende dos valores dos parâmetros do subprograma.
- Logo, para utilizar um determinado subprograma, é necessário obrigatoriamente fornecer a ele os parâmetros adequados para sua execução.

Passagem de parâmetros

- **Definição:** aos mecanismos de informar os valores a serem processados pelo subprograma dá-se o nome de passagem de parâmetros.
- Nesta etapa do curso, iremos utilizar duas formas distintas de passagem de parâmetros:
 - Passagem por cópia, também chamada de passagem por valor.
 - Passagem por referência.

Passagem por cópia ou por valor



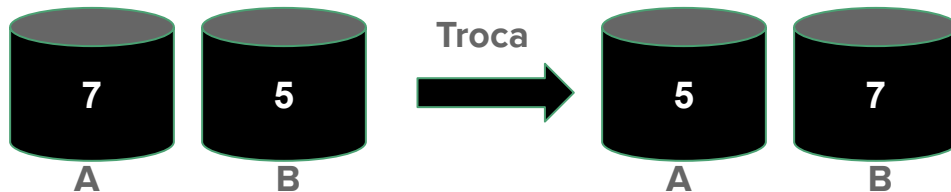
- As variáveis passadas como parâmetro não tem seu valor modificado dentro do subprograma.
- Quando um subprograma recebe um argumento, ele faz uma cópia do valor deste argumento no parâmetro formal correspondente.
 - O parâmetro é uma cópia da variável original. Deste modo, mudar o parâmetro formal não muda a variável original.

Passagem por cópia ou por valor

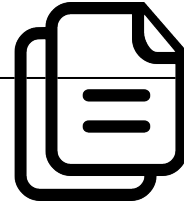


- Exemplo

- Problema: implementar um procedimento em C++ que dado duas variáveis inteiras A e B, troque os valores armazenados nas variáveis entre si. Por exemplo:



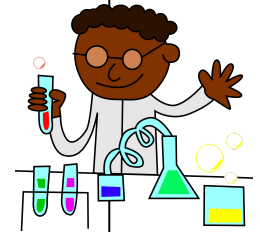
- Utilize passagem de parâmetros por **cópia** no seu subprograma.



```
#include <iostream>
using namespace std;
```

```
void troca(int A, int B) {
    int AUX = A;
    A = B;
    B = AUX;
    cout << "A e B no subprograma troca: " << A << " " << B << endl;
}
```

```
int main() {
    int X, Y;
    cin >> X >> Y;
    cout << "X e Y antes subprograma troca: " << X << " " << Y << endl;
    troca(X, Y);
    cout << "X e Y depois subprograma troca: " << X << " " << Y << endl;
    return 0;
}
```

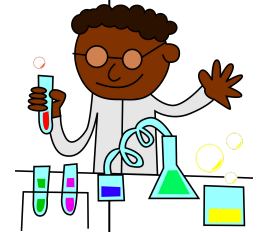
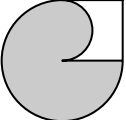




```
#include <iostream>
using namespace std;
```

```
void troca(
    25 32
    X e Y antes subprograma troca: 25 32
    A e B no subprograma troca: 32 25
    X e Y depois subprograma troca: 25 32
)
```

```
int main() {
    int X, Y;
    cin >> X >> Y;
    cout << "X e Y antes subprograma troca: " << X << " " << Y << endl;
    troca(X, Y);
    cout << "X e Y depois subprograma troca: " << X << " " << Y << endl;
    return 0;
}
```



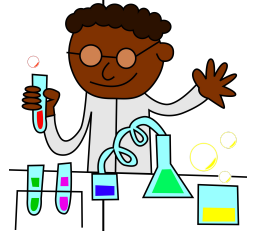


```
#include <iostream>
using namespace std;
```

```
void troca(
25 32
X e Y antes subprograma troca: 25 32
A e B no subprograma troca: 32 25
X e Y depois subprograma troca: 25 32
}
```

```
int main() {
    int X, Y;
    cin >> X >> Y;
    cout << "X e Y antes troca: 25 32" << endl;
    troca(X, Y);
    cout << "X e Y depois troca: 25 32" << endl;
    return 0;
}
```

Valores foram trocados internamente no subprograma, mas não impactaram nas variáveis externas.



Passagem por referência



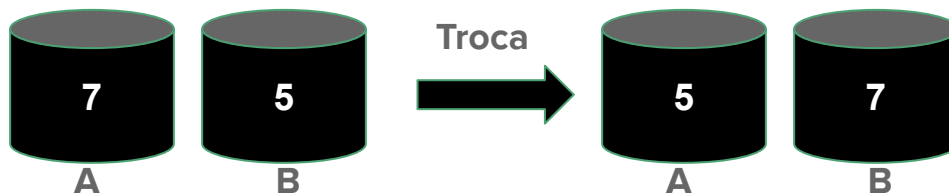
- Não é feita uma cópia da variável. Um parâmetro que seja alterado internamente no subprograma irá refletir essa alteração no código externo ao subprograma.
- Para indicar que um parâmetro está sendo passado por referência, em C++, utiliza-se o símbolo **&**.

Passagem por referência



- Exemplo

- Problema: implementar um procedimento em C++ que dado duas variáveis inteiras A e B, troque os valores armazenados nas variáveis entre si. Por exemplo:

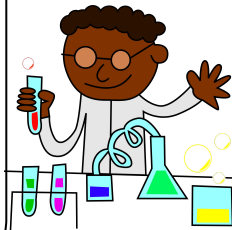
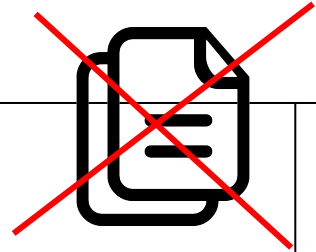


- Utilize passagem de parâmetros por **referência** no seu subprograma.

```
#include <iostream>
using namespace std;
```

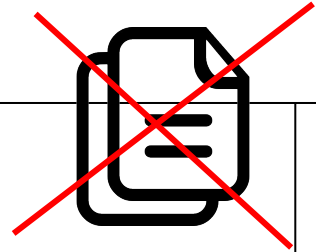
```
void troca(int& A, int &B) {
    int AUX = A;
    A = B;
    B = AUX;
    cout << "A e B no subprograma troca: " << A << " " << B << endl;
}
```

```
int main() {
    int X, Y;
    cin >> X >> Y;
    cout << "X e Y antes subprograma troca: " << X << " " << Y << endl;
    troca(X, Y);
    cout << "X e Y depois subprograma troca: " << X << " " << Y << endl;
    return 0;
}
```



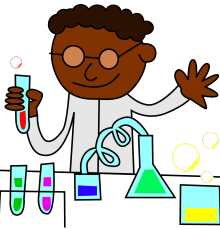

```
#include <iostream>
using namespace std;
```

Passagem por
referência



```
void troca(int& A, int &B) {
    int AUX = A;
    A = B;
    B = AUX;
    cout << "A e B no subprograma troca: " << A << " " << B << endl;
}

int main() {
    int X, Y;
    cin >> X >> Y;
    cout << "X e Y antes subprograma troca: " << X << " " << Y << endl;
    troca(X, Y);
    cout << "X e Y depois subprograma troca: " << X << " " << Y << endl;
    return 0;
}
```

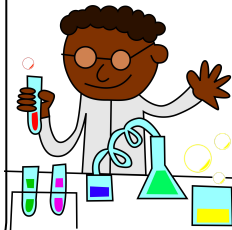
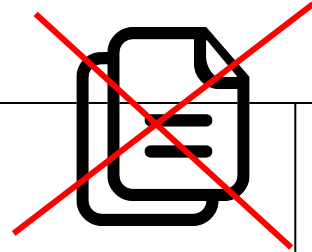


```
#include <iostream>
using namespace std;
```

```
void troca(int& A, int &B) {
    int AUX = A;
    A = B;
    B = AUX;
    cout << "A e B no subprograma troca: " << A << " " << B << endl;
}

int main() {
    int X, Y;
    cin >> X >> Y;
    cout << "X e Y antes subprograma troca: " << X << " " << Y << endl;
    troca(X, Y);
    cout << "X e Y depois subprograma troca: " << X << " " << Y << endl;
    return 0;
}
```

O & pode vir
tanto junto do
tipo como do
identificador!

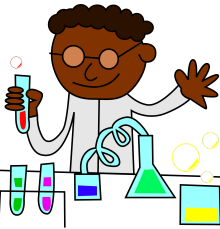
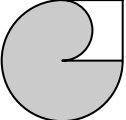




```
#include <iostream>
using namespace std;
```

```
void troca(int X, int Y) {
    A X e Y antes subprograma troca: 25 32
    B A e B no subprograma troca: 32 25
    C X e Y depois subprograma troca: 32 25
}
```

```
int main() {
    int X, Y;
    cin >> X >> Y;
    cout << "X e Y antes subprograma troca: " << X << " " << Y << endl;
    troca(X, Y);
    cout << "X e Y depois subprograma troca: " << X << " " << Y << endl;
    return 0;
}
```



```
#include <iostream>
using namespace std;
```

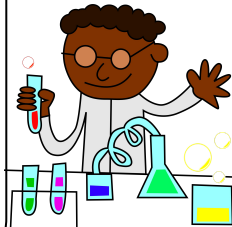
```
void troca(int X, int Y) {
    A 25 32
    B X e Y antes subprograma troca: 25 32
    C A e B no subprograma troca: 32 25
    D X e Y depois subprograma troca: 32 25
}
```

```
int main() {
    int X, Y;
    cin >> X >> Y;
    cout << "X e Y antes subprograma troca: ";
    troca(X, Y);
    cout << "X e Y depois subprograma troca: ";
    return 0;
}
```

Valores foram trocados internamente no subprograma, com impacto nas variáveis externas.



Resultados diferentes



Passagem de parâmetros

- Por padrão, parâmetros de tipos fundamentais escalares (**int**, **float**, **bool**, etc.) são sempre passados por cópia para um subprograma.
 - Para passá-los por referência, o programador deve explicitamente utilizar o símbolo **&** no parâmetro formal do subprograma de interesse, como apresentado anteriormente.
- No futuro, iremos estudar outros tipos de dados, os quais, por padrão, são passados por referência, sem a necessidade de utilizar o símbolo **&** para tal.

Passagem por ponteiro e passagem de vetor como parâmetro



Passagem de Parâmetros

Relembrando do que já vimos:

Passagem por valor/cópia:

```
int foo1(int a, int b);
```

⇒ As variáveis passadas como parâmetro não tem seu valor modificado dentro da função.

Passagem por referência:

```
int foo2(int &a, int &b);
```

⇒ As variáveis passadas como parâmetro refletem fora da função as alterações que sofrerem dentro dela.

Passagem por ponteiros

Passagem por ponteiros:

```
int foo3(int *a, int* b);
```

⇒ Esse é um caso específico de passagem por cópia, mas a cópia é do endereço da variável. Assim, os valores também podem ser alterados (como na passagem por referência), uma vez que se tem acesso à variável (por conta de seu endereço). Essa passagem também recebe o nome de *passagem por referência indireta*.

Passagem por ponteiros versus referência

Vários desenvolvedores preferem utilizar passagens por ponteiros que por referência, pelo fato que a passagem por ponteiro é mais legível:

```
int foo2(int &a, int &b);  
int foo3(int *a, int* b);  
int x, y;  
foo2(x,y); // cópia ou referência?  
foo3(&x,&y);
```

A chamada por ponteiros deixa explícito que o valor pode ser alterado pela função.

Passagem de vetores

- Por padrão, parâmetros de tipos fundamentais escalares (**int**, **float**, **bool**, etc.) são sempre passados por cópia para um subprograma.
 - Para passá-los por referência, o programador deve explicitamente utilizar o símbolo **&** no parâmetro formal do subprograma de interesse, como apresentado anteriormente.
- Porém, quando temos como parâmetro do subprograma uma variável do tipo composta homogênea (vetores/arranjos ou matrizes), um caso particular de grande importância acontece.

Passagem de vetores

- A passagem de parâmetros com vetores ou matrizes, mesmo que sem o operador **&**, sempre permite a alteração dos dados armazenados nas variáveis.
 - Ou seja, as alterações nos valores contidos nos parâmetros formais do tipo vetor/matriz são refletidas fora do subprograma.
- Excepcionalmente, funciona como se fosse sempre uma passagem de parâmetros por referência.
 - Isto é feito como uma forma de economizar tempo e memória do computador.

Passagem de vetores

- A passagem de parâmetros com vetores ou matrizes, mesmo que sem o operador **&**, sempre permite a alteração dos dados armazenados nas variáveis
 - Ou form
- Excepc
passag

**Isso ocorre principalmente pelo fato que, em C/C++ todo vetor é um ponteiro!
Então o processo aqui é similar à
passagem de parâmetros por ponteiros.**

parâmetros
rograma.

mpre uma

Passagem de parâmetros - Forma 1

- Para especificar que um parâmetro formal de um subprograma é uma variável do tipo vetor, basta colocar um par de colchetes ao lado do identificador do parâmetro que é daquele tipo.
- Como exemplo, teste o subprograma a seguir, chamado **dobra()**, responsável por dobrar os valores armazenados em um vetor de 5 posições.

```
#include <iostream>
using namespace std;
```

```
void dobra(int vet[], int tam){
    int i = 0;
    while (i < tam){
        vet[i] = 2*vet[i];
        ++i;
    }
}
```

```
int main(){
    int vetor[5] = {1,2,3,4,5};
    int i = 0;
    //primeira exibição
    while (i < 5){
        cout << vetor[i] << " ";
        i++;
    }
}
```

```
cout << endl;
```

```
dobra(vetor,5);
i = 0; //reinicia contador
```

```
//segunda exibição
while (i < 5){
    cout << vetor[i] << " ";
    i++;
}
cout << endl;
return 0;
}
```



```
#include <iostream>
using namespace std;
```

```
void dobra(int vet[], int tam){
    int i = 0;
    while (i < tam){
        vet[i] = 2*vet[i];
        ++i;
    }
}
```

```
int main(){
    int vetor[5] = {1,2,3,4,5};
    int i = 0;
    //primeira exibição
    while (i < 5){
        cout << vetor[i] << " ";
        i++;
    }
}
```

```
cout << endl;
```

```
dobra(vetor,5);
i = 0; //reinicia contador
```

```
//segunda exibição
while (i < 5){
    cout << vetor[i] << " ";
    i++;
}
cout << endl;
return 0;
}
```



Este é um exemplo de especificação de um parâmetro do tipo vetor. Note que, neste momento, não devemos indicar entre os colchetes o tamanho (número de elementos) do vetor.

Passagem de parâmetros - Forma 2

- Outra forma de passar um vetor como parâmetro é utilizar o fato que ele também é um ponteiro. Assim, utiliza-se a notação de ponteiros.
- Como exemplo, teste o subprograma a seguir, chamado **dobra()**, responsável por dobrar os valores armazenados em um vetor de 5 posições.


```
#include <iostream>
using namespace std;

void dobra(int* vet, int tam){
    int i = 0;
    while (i < tam){
        vet[i] = 2*vet[i];
        ++i;
    }
}

int main(){
    int vetor[5] = {1,2,3,4,5};
    int i = 0;
    //primeira exibição
    while (i < 5){
        cout << vetor[i] << " ";
        i++;
    }
```

```
    cout << endl;

    dobra(vetor,5);
    i = 0; //reinicia contador

    //segunda exibição
    while (i < 5){
        cout << vetor[i] << " ";
        i++;
    }
    cout << endl;
    return 0;
}
```



```
#include <iostream>
using namespace std;
```

```
void dobra(int* vet, int tam){
    int i = 0;
    while (i < tam){
        vet[i] = 2*vet[i];
        ++i;
    }
}
```

```
int main(){
    int vetor[5] = {1,2,3,4,5};
    int i = 0;
    //primeira exibição
    while (i < 5){
        cout << vetor[i] << " ";
        i++;
    }
}
```

```
cout << endl;
```

```
dobra(vetor,5);
i = 0; //reinicia contador
```

```
//segunda exibição
while (i < 5){
    cout << vetor[i] << " ";
    i++;
}
cout << endl;
return 0;
```

```
}
```

Este é um exemplo de especificação de um parâmetro do tipo vetor como ponteiros.



Sobre o Material



Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Janderson Rodrigo de Oliveira
- Joaquim Quinteiro Uchôa
- Juliana Galvani Greggi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.

Esta obra está licenciado com uma Licença [Creative Commons Atribuição 4.0 Internacional](https://creativecommons.org/licenses/by/4.0/).