

Disciplina de MLP

Expressões e Estruturas de Controle

Instituto de Informática – UFRGS

Tópicos

- I. Expressões e atribuição
- II. Estruturas de controle (de fluxo)

Instituto de Informática – UFRGS

2

I. Expressões

- Definição / conceito de expressão
- Expressões **aritméticas**
- Operadores sobrecarregados
- Conversões de tipo
- Expressões **relacionais** e **booleanas**
- **Avaliação curto-circuito**
- Expressões de **atribuição**
- Atribuição de **modo misto**

Instituto de Informática – UFRGS

3

I. Expressões - definição

- Contexto: linguagens imperativas
- Definições:
 - Meio fundamental para **especificar computações**
 - Frase do programa que **necessita ser avaliada** e que **produz um valor como resultado**

Instituto de Informática – UFRGS

4

I. Expressões - conceitos

■ Elementos de uma expressão:

- **Operadores** (definem a operação)
- **Operandos** (definem valores sobre os quais ela é realizada)
- **Parênteses** (modificam ordem de avaliação/associatividade)
- Chamadas a **funções** (definem novos operadores)

■ Envolverem sintaxe e semântica específicas

■ Importante levar em conta

(no projeto e no estudo de uma linguagem):

- Regras semânticas para a **ordem** e a **forma de avaliação**
- **Efeitos** normais e **colaterais** da ordem ou método de avaliação utilizado

Tipos de operadores

■ Quanto à sua **aridade**:

- **Unários** (um único operando)

`a + (- b) * c` (mudança de sinal)

- **Binários** (dois operandos)

`a + b`

- **Ternários** (três operandos)

`A = (a > b) ? true : false;` (expressão condicional)

- **“Eneários”** (aridade variável)

`printf(char[], ...);` (qualquer número de operandos)

(Expressões condicionais)

■ Instruções de seleção usadas para **executar atribuição de expressão condicional**

■ Válidas em C e Java:

- `media = (cont == 0) ? 0 : soma/cont;`
- // mesmo que:
`if (cont == 0) media = 0;`
`else media = soma / cont;`

Tipos de operadores

■ Quanto à sua **posição**:

- **Pré-fixados** (antes dos operandos)

`+ a b` `++cont`

`-a` `Not a`

`soma(a,b)` // funções são consideradas pré-fixadas

- **Infixados** (entre os operandos)

`a + b`

- **Pós-fixados** (após os operandos)

`cont++`

I. Expressões - tipos

- Literais
- Agregações
- Chamadas de funções
- Expressões de referenciamento
- Expressões aritméticas
- Expressões relacionais
- Expressões booleanas (*bit-a-bit* ou não)
- Expressões de atribuição

I. Expressões - tipos

- Literais:
 - Produzem um valor fixo
 - Exemplos: `'a'` 87 0x54 1.23
- Agregações:
 - Produzem um valor a partir de seus componentes
 - Exemplos:
 - Vetores: `int c = {1, 2, 3};`
 - Registros e Estruturas: `String nome = p.nome;`
- Chamadas de funções:
 - Produzem o resultado da aplicação de uma função sobre um ou mais parâmetros

I. Expressões - tipos

- Expressões de referenciamento
 - Utilizadas para **acessar o conteúdo** de variáveis ou constantes ou para **retornar uma referência** a esses elementos
 - Normalmente, o mesmo operador/símbolo é utilizado para acessar o conteúdo ou obter uma referência a ele (o que diferencia é o contexto onde ele aparece):
`*q = *q + 3;`
 - O `*q` do lado esquerdo referencia a referência (obtem o endereço)
 - O `*q` do lado direito referencia o conteúdo do objeto apontado por q

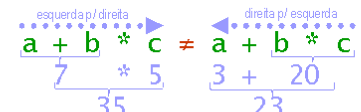
(Alguns operadores de referenciamento em linguagem C)

Operador	Significado
<code>[]</code>	Acesso a valor ou retorno de referência de elemento de vetor
<code>*</code>	Acesso a valor ou retorno de referencia de variável ou constante apontada por ponteiro
<code>.</code>	Acesso a valor ou retorno de referência de elemento de estrutura
<code>-></code>	Acesso a valor ou retorno de referência de elemento de estrutura apontada por ponteiro
<code>&</code>	Retorno de referência a qualquer tipo de variável ou constante

Expressões aritméticas

- **Finalidade:** especificar computações aritméticas
- **Origem:** Matemática
 - Define características e estabelece convenções
 - Define os operandos e expressões possíveis
- **Com o que se preocupar:**
 - Ordem de avaliação dos operadores
 - Precedência
 - Associatividade
 - Uso de parênteses
 - Ordem de avaliação dos operandos (e seus efeitos colaterais)
 - Sobrecarga e mesclagem dos modos

Operadores: ordem de avaliação

- **Depende da precedência e da associatividade:**
 - Ordem de avaliação dos operadores é relevante em alguns casos...
 $a=3;b=4;c=5;$

 - Mas o tipo do operador também tem que ser levado em conta: **ordem de precedência**
 - A precedência dos operadores mais comuns (clássicos) é parecida na maioria das linguagens
 - Os projetistas costumam especificar ordens diferentes para os outros

Operadores: regras de precedência

- **Regra geral** (operadores clássicos):
 1. Parênteses
 2. Exponenciação
 3. Multiplicação e divisão
 4. Adição e subtração (binárias)

Operadores: associatividade

- As vezes, a ordem de precedência não é suficiente!
- Como avaliar $a - b + c - d$?
- Regras de associatividade são **aplicadas quando há duas ou mais ocorrências adjacentes de operadores de mesmo nível**
- Tipos:
 - Associatividade à esquerda
 - Associatividade à direita
- Nas linguagens comuns, praticamente sempre acontece à esquerda
 - Exceções:
 - Exponenciação (em Fortran e Ada)
 - ++ e -- prefixados; + e - unários (em C)
 - ++ e --; + e - unários (em C++)

Operadores: associatividade

- E importa a ordem em casos como o seguinte?

$$a + b + c + d$$

- A operação é associativa por natureza matemática...
- Mas os números no computador são discretos (em termos de representação) e os resultados são aproximados!
- No exemplo, se 'a' e 'c' forem números muito grandes e 'b' e 'd' números negativos muito grandes:
 - $a + b \rightarrow \text{OK}$ Ex.: $4.000.001 + -3.999.000$
 - $a + c \rightarrow \text{Overflow}$ Ex.: $4.000.001 + 3.999.000$

- Logo, a ordem importa em alguns casos!

Operadores: ordem de avaliação

- O uso de parênteses muda a ordem

- Em teoria, as linguagens que aceitam parênteses não precisariam especificar regras de precedência

- Problemas:

- Torna a escrita de expressões mais tediosa e
- Compromete a legibilidade do código

- Mas os ambientes atuais de edição de código costumam marcar os pares:

```
(h / ((a + b * (c + b)) + 0.000001))
```

Operandos: ordem de avaliação

- Preocupações quanto ao tipo do operando:

- Variáveis: como minimizar o acesso à memória?
- Constantes: ficam em memória ou fixos no código?
- Uso de parênteses: avaliadas antes das outras
- Funções: seu resultado deve ser recuperado

- Preocupações quanto aos efeitos colaterais (se não houver, não importa a ordem)

Efeito colateral (funcional)

- Ocorre quando uma função modifica um de seus parâmetros ou uma variável global

- Exemplo: $a + \text{fun}(a)$ // supondo que $a = 10$ e $\text{fun}(x) \rightarrow x = x/2$

- Se fun não modifica 'a', a ordem não importa
- Se modifica 'a' e a função for executada primeiro, o valor resultante será um (pois o valor de 'a' será diferente do passado como parâmetro)

```
a + fun(10)
= a + (a=10/2)
= a + 5
= 5 + 5 → 10
```

- Se modifica, mas o valor de 'a' for recuperado primeiro, o resultado será outro:

```
a + fun(a)
= 10 + fun(a)
= 10 + (a=10/2)
= 10 + 5 → 15
```

Efeito colateral (funcional)

- Alternativas:
 - Impedir / eliminar funções que modifiquem elementos externos
 - Mas como ter funções que “retornam” mais de um valor?
 - Como oferecer troca rápida de dados entre módulos de programação sem variáveis globais?
 - Optar por uma única ordem de avaliação (caso Java)
 - Mas como otimizar código em nível de compilação (que envolve reorganização de expressões e operandos)?
 - Aceitar somente quando as funções não modificarem os valores dos outros operandos na mesma expressão (Fortran 77)
 - Mas como garantir isso?
 - Deixar rolar...
 - Pascal e Ada deixam a escolha para o implementador da linguagem

Sobrecarga: cuidar legibilidade

- Mesmo símbolo, múltiplos propósitos
 - `x = &y;` // retorna o endereço de y
 - `z = x & y;` // faz 'and' bit-a-bit
- Conjuntos distintos, com resultados distintos:
 - `float media;`
`int soma, cont;`
`media = soma / cont;` // qual o problema?
→ divisão de inteiros = inteiro
(e o resultado truncado é convertido para float!)

Conversão (problemas de)

- Estreitamento: “tipo maior para menor”
 - Pode gerar problemas de perda de informação
- Alargamento: “tipo menor para maior”
 - Sem problema aparente, mas pode-se perder precisão, dependendo da implementação da linguagem
 - Exemplo:
 - int (32 bits) → 9 dígitos decimais de precisão
 - float (32 bits) → 7 dígitos decimais de precisão
 - int (32 bits) para float (32 bits) → perda de 2 dígitos na precisão

I. Expressões

- Erros ou exceções relacionados:
 - Tipo incompatível
 - Overflow
 - Underflow
 - Divisão por zero

Expressões relacionais

- Expressões que **envolvem operadores relacionais**: igual, maior (que), menor (que), diferente...
- São **sobrecarregados para diversos tipos de dados** (normalmente números, cadeias e ordinais – tipos primitivos)

Expressões booleanas

- Envolve variáveis, constantes, expressões relacionais e **operadores booleanos**: AND, OR, NOT (em alguns casos também OR exclusivo e equivalência)
- Normalmente possuem **ordem de precedência**:
 1. NOT
 2. AND
 3. OR
- C não trabalha com tipo booleano, e os resultados deste tipo de expressão são números inteiros onde:
 - 0 → falso
 - !0 → verdadeiro
- Além disso, C tem **associação à esquerda** (útil em curtos-circuitos)

Expressões booleanas

- Algumas consequências interessantes em C:
 - Válido: `a > b > c`
 - Válido: `while(!feof(arquivo))`
 - Válido: `while(resultado = fread(arquivo))`

Avaliação em curto-circuito

- **Atalho** que ocorre quando o **resultado de uma expressão é determinado por uma parte dela** (sem avaliar todos os operadores e/ou operandos, pois eles são irrelevantes)
- Exemplos de aplicabilidade:
 - Na expressão: `(13 * a) * (b / 13 - 1)`
Se `a = 0`, o resultado independe de `(b / 13 - 1)`
 - Na expressão: `(a >= 0) AND (b < 10)`
Se `a < 0`, o resultado independe de `(b < 10)`
- Obs.: normalmente expressões matemáticas não seguem este tipo de avaliação, pois ele é complexo e difícil de identificar. A técnica é mais aplicada em expressões booleanas.

Avaliação em curto-circuito

- Considere o exemplo:

```
indice = 1;
while((indice < tamanho) && (lista[indice] <> chave))
    indice++;
```

- Se a linguagem não usa curto-circuito:
 - Avalia ambas as expressões
 - Se o índice for igual ao tamanho, não aumenta o índice, mas testa lista[indice] e gera erro de faixa (pois a lista vai de 0 até tamanho-1)
- Se usa:
 - Avaliaria o primeiro e sairia fora, sem avaliar o segundo

Avaliação em curto-circuito

- Desvantagem: expressões com efeitos colaterais podem não ser executadas

- Exemplo:

- Em: `(a > b) || (b++ / 3)`
- Se `a >= b`, `b++` não ocorre!

- Obs.:

- C, C++ e Java avaliam && (and) e || (or) por curto-circuito!
- Em ADA,
 - há operadores tradicionais: 'and' e 'or'
 - e os que usam curto-circuito: 'and then' e 'or else'

Fazer Exercício

(Operador booleano vs *bit-a-bit*)

- Em Java, AND (&&) e OR (||) são booleanos (trabalham com expressões booleanas):

```
int a; boolean b;
if((a>10) && (b)) { ... } // ok
if(a && b) { ... } // Erro: 'a' não é expressão nem é tipo booleano
```

- Mas "&" e "|" trabalham com bits:

```
int a = 1, b=2;
if(a & b) { ... } // Erro: o resultado não é um valor booleano!
```

Instruções de atribuição

- Permitem mudar o valor de uma variável
- Papel fundamental nas linguagens imperativas
- Tipos:

- Atribuições simples
- Atribuições com alvos múltiplos
- Atribuições com alvos condicionais
- Operadores de atribuição (compostos, unários e expressões de atribuição)

Instruções de atribuição

■ Atribuições simples:

- Sintaxe: `<alvo> <operador> <expressão>`
- Exemplos: `a = b;`
`a := c + d;`
- Problema que ocorre quando o operador de atribuição é igual ao de igualdade:
`a = b = c;` // colocaria em 'a' o resultado do teste de igualdade

■ Alvos múltiplos:

- Exemplos: `a = b = 0;` // c++, Java
`soma, total = 0;` // PL/I

Instruções de atribuição

■ Alvos condicionais (em C++):

- A expressão (pois gera um *lvalue*):
`(flag ? cont1 : cont2) = 0;`
- É o mesmo que:
`if(flag!=0) cont1 = 0; else cont2 = 0;`

■ Operadores de atribuição compostos:

- `soma += valor;`
- `resultado /= numero;`

Instruções de atribuição

■ Operadores de atribuição unários (C, Java):

`cont++;` → `cont = cont + 1;`

Atenção ao efeito colateral:

`soma = ++cont;` → `cont = cont + 1; soma = cont;`
`soma = cont++;` → `soma = cont; cont = cont + 1;`

Instruções de atribuição

■ Atribuição como expressão:

- Em C e Java, a instrução de atribuição produz um resultado (que é o mesmo valor atribuído ao alvo)
- Exemplos:
 - `a = b + (c = d / b++) - 1;`
 - `if(x = y)` // cuidar! não seria `x == y` ?
Obs.: em Java esse efeito colateral que poderia ocasionar erros nos ifs não ocorre, pois ela só aceita expressões booleanas em ifs

Atribuição de modo misto

- São aquelas que **aceitam uma expressão que não é do mesmo tipo do alvo**
- Resolvidas através de coersão
- Questão: quando ou onde fazer a coersão?
 - Ou seja: `real = int1 / int2`
 - seria lido como: `real = (float)(int1 / int2) // perda!`
 - ou como: `real = ((float)int1 / (float)int2)`

II. Estruturas de controle

- Definição:
 - Servem para **agregar expressões e comandos**;
 - **Controlar o fluxo** de execução do programa
 - Podem ser primitivos (i.e., simples) ou compostos
- Tipos:
 - **Sequenciais** (bloco que agrega comandos sequenciais)
 - **De seleção** (comandos condicionais)
 - **Iterativas** (comandos iterativos)
 - **De desvio incondicional**
 - **Comandos protegidos**

Estruturas condicionais

- Caminho condicionado:
`if(a<0) { printf("erro"); getch(); }`
- Caminho duplo:
`if(a<0) { printf("erro"); getch(); }
else printf("valor correto");`
- Caminhos múltiplos:
 - ifs aninhados (else if)
 - comando de seleção múltipla (switch, case)

```
switch(mes){  
    case 3: case 4: case 5: printf("outono"); break;  
    case 6: case 7: case 8: printf("inverno"); break;  
    case 9: case 10: case 11: printf("primavera"); break;  
    case 12: case 1: case 2: printf("verão"); break;  
    default: printf("Mês inválido");  
}
```

Comandos iterativos (repetição)

- Número indefinido de repetições:
 - Pré-teste
 - `while(<teste_verdadeiro>) { <comandos> }`
 - Pós-teste (pelo menos 1 execução)
 - `do{ <comandos> } while(<teste_verdadeiro>);`
 - `Repeat <comandos> until <teste_falso>;`
- **Problema: e se necessitarmos de outros pontos de saída?**
- **Solução: desvio incondicional**

Comandos iterativos (repetição)

■ Número definido de repetições:

- Número de iterações conhecido ou determinado previamente
- Necessita de variável de controle (contador/iterador)
- Pode-se determinar o valor inicial, final e o incremento
- Exemplos:
 - `for i := 1 to 10 do vetor[i] := i*2;`
 - `for i := 1 to 100 Step 2 do a := a + i;`
 - `for i := 10 DOWNTO 0 Step 2 do a := a + i;`
 - `for(i=0 ; i<10 ; i++) vetor[i] = i*2;`
- Também se pode utilizar os comandos apresentados anteriormente (de preferência os que possuem a condição no começo, e.g., *while*)

Comandos iterativos (repetição)

■ Considerações sobre número definido de repetições:

- Número de repetições deve ser fixo (apesar de algumas linguagens permitirem modificar o contador dentro do laço) para manter a legibilidade e evitar problemas (Ada não permite)
- A variável de controle deve ser discreta
- O comando for em linguagem C é na verdade um comando de repetição indefinida
- A variável de controle deveria ter o escopo limitado ao laço:
 - Ada limita
 - Em C++, é possível definir a variável dentro do 'for' (mas ela vale depois)
 - Em Java também, mas ela é limitada ao escopo do 'for'

Comando for em C, C++, Java

■ Sintaxe:

```
for (<expr1> ; <expr2> ; <expr3>) <comando>;
```

■ Parâmetros do laço (todos são opcionais!):

- <expr1>: - inicialização, avaliada uma vez
- <expr2>: - controle do laço, avaliada a cada iteração (pré-teste)
 - se igual a 0, termina
 - se não existe, *default* é *true* (!0)
- <expr3>: - executada em cada iteração, após o corpo do laço

■ Observações:

- As expressões podem conter declarações, comandos ou seqüências de comandos separados por vírgula
- O corpo do laço pode ser vazio, conter um comando simples, uma composição ou um bloco

Comando for em C, C++, Java

- Cada expressão pode conter múltiplos comandos (valor da expressão é o valor do último comando)

```
for (conta1 = 0, conta2 = 0.0;  
    conta1 <= 10 && conta2 <= 100.0;  
    soma = ++conta1 + conta2, conta2 *=2.5);
```

■ Exemplos :

- `for (int i=0; i <n; i++) ...`
- `for (i = 0, int j = 10; j == i; i++) ...`
- `for (; ;) ...laço infinito...`

Iteradores

- **Motivação:** como utilizar tipos de dados não convencionais (vetores, árvores, pilhas, etc.) para o controle de um laço?
- LPs mais atuais permitem possuem operações pré-definidas (*built-in*) ou permitem definir iteradores para os mais diferentes TADs
- Também chamados de “cursors”

Iteradores

- E.g., em Java, toda classe que define uma coleção deve implementar a interface `Iterator` que exporta os métodos `hasNext()` e `next()`
- Exemplo 1 (cancela todas as tarefas de tempo da coleção 'c'):

```
void cancelAll(Collection<TimerTask> c) {  
    for(Iterator<TimerTask> i = c.iterator(); i.hasNext();)  
        i.next().cancel();  
}
```
- Exemplo 2 (define e mostra todos os sabores da coleção 'sabores'):

```
List<String> sabores = new ArrayList<String>();  
sabores.add("chocolate");  
sabores.add("morango");  
sabores.add("limão");  
Iterator<String> itemSabor = sabores.iterator();  
while(itemSabor.hasNext()){  
    System.out.println( itemSabor.next() );  
}
```

Iteradores

- **Comando foreach** (Java 1.5, C#, PHP...):
 - Exemplo 1 (cancela tarefas de tempo):

```
void cancelAll(Collection<TimerTask> c) {  
    for (TimerTask t : c) t.cancel();  
}
```

OBS: lê-se o ':' como 'in' (em), ou seja, "para cada tarefa de tempo *t* em *c*, cancele-a"!
 - Exemplo 2 (acumula valores de um array):

```
int sum(int[] a) {  
    int result = 0;  
    for (int i : a) result += i;  
    return result;  
}
```

Desvios incondicionais

- **Tipos:**
 - Irrestritos
 - Restritos

Desvios incondicionais

■ Irrestritos (evitar)

□ Comando goto

- Favorece programação macarrônica
- Em alguns casos, o ganho de eficiência pode compensar a legibilidade (em fluxos de controles aninhados)
- Exemplo:

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        if(a[i]==b[j]) goto saida;
saida:
if(i<n) printf("achou"); else printf("não achou");
```

Desvios incondicionais

■ Restritos – Escapes

- Permitem finalização diferente da usual
- Desvios disciplinados e estruturados
- Não entram em repetições nem desviam o fluxo para um lugar qualquer
- Comandos típicos:
 - Break : termina um bloco de laço mais interno
 - Continue : pula a iteração corrente do laço mais interno
 - Return : sai do subprograma
 - Exit : sai do programa (termina-o)

Atenção: break e continue só funcionam dentro de um laço (não servem para blocos condicionais)!

Desvios incondicionais

■ Restritos – Escapes

□ Exemplo com break:

```
s=0;
for(;;){ // lê um número inteiro natural
    printf("n: ");
    scanf("%d", &n);
    if(n <= 0) break; // vai para o fim do laço e sai
    s+=n;
}
```

Desvios incondicionais

■ Restritos – Escapes

□ Exemplo com continue:

```
s=0; i = 0;
while(i<10){ // soma 10 números naturais
    printf("n: ");
    scanf("%d", &n);
    if(n < 0) continue; // volta para o início do laço
    s+=n;
    i++;
}
```

Desvios incondicionais

■ Restritos – Escapes

- Exemplo com `return` e com `exit`:

```
void trata (int erro) {
    if(erro==0){
        printf("nada a tratar");
        return; // interrompe função
    }
    if(erro<0){
        printf("erro grave");
        exit(1); // sai do programa
    }
    printf("erro tratado");
}
```

Desvios incondicionais

■ Escapes rotulados (Java e ADA):

- Exemplo 1 (em Java):

```
busca:
    for(i=0; i<arrayOfInts.length; i++) {
        for(j=0; j<arrayOfInts[i].length; j++) {
            if (arrayOfInts[i][j] == searchfor) {
                encontrado = true;
                break busca;
            }
        }
    }
}
```

Atenção: O `break` termina o comando rotulado, que no caso é o conjunto de comandos `for` aninhados (não volta para o label `busca`!). Se o rótulo não tivesse sido especificado, o `break` terminaria o `for` interno!

Desvios incondicionais

■ Escapes rotulados (Java e ADA):

- Exemplo 2 (em Java):

```
Loop_externo:
    for(;;) {
        Loop_interno:
            for(;;){
                /* comandos */
                break Loop_interno;
                /* comandos */
                continue Loop_externo;
            }
    }
```

No exemplo, `break` termina o `loop` interno e o `continue` pula a iteração dos dois `loops` (interno e externo)!

Leitura recomendada

- Sebesta, R. W. Expressões e Instruções de Atribuição (capítulo 7). In: Sebesta, R. W. Conceitos de Linguagens de programação. 5a edição. Porto Alegre: Bookman, 2003.
- Sebesta, R. W. Estruturas de Controle no Nível da Instrução (capítulo 8). In: Sebesta, R. W. Conceitos de Linguagens de programação. 5a edição. Porto Alegre: Bookman, 2003.
- Tutorial Java sobre "Branching Statements":
<http://java.sun.com/docs/books/tutorial/java/nutsandbolts/branch.html>
- Tutorial Java sobre operadores:
<http://72.5.124.55/docs/books/tutorial/java/nutsandbolts/operators.html>