

Fundamentos de Programação

Operadores lógicos

Dainf - UTFPR

Profa. Leyza B. Dorini

Prof. Bogdan T. Nassu

Em alguns problemas, é necessário criar estruturas que representem condições mais complexas. Por exemplo, suponha que os requisitos para aprovação na disciplina sejam:

- Frequência mínima de 75%.
- Média da nota das duas provas maior ou igual a 6 ou, caso contrário, nota do exame maior ou igual a 7.

Observe que, para representar a condição “aprovado na disciplina”, é preciso combinar diferentes expressões relacionais (as quais, por sua vez, podem depender de resultados de expressões aritméticas).

Motivação

Mais especificamente, é preciso criar uma estrutura condicional que represente o seguinte:

- $frequencia > 75$ precisa ser verdadeiro.
- Pelo menos uma condição dentre $(notaP1 + notaP2) / 2 \geq 6$ e $notaExame \geq 7$ precisa ser verdadeiro.
- Os dois requisitos acima precisam ocorrer na mesma execução do programa.

Para combinar tais expressões relacionais (de forma a representar a condição necessária para aprovação), é possível explorar estruturas condicionais aninhadas, por exemplo (veja a próxima página).

Uma possível solução...

este if testa o critério
de frequência mínima...

```
1
2  if(frequencia > 75)
3  {
4      if((notaP1+notaP2)/2 >= 6)
5          printf("Aprovado.");
6      else if(notaExame >= 7)
7          printf("Aprovado.");
8      else
9          printf("Reprovado.");
10 }
11 else
12     printf("Reprovado.");
```

...e seu bloco de comandos, que consiste
em uma estrutura condicional aninhada,
testa o critério de nota mínima

Quais as desvantagens desta solução?

Criando condições mais complexas

Note que, no final das contas, queremos expressar a seguinte condição:

esta expressão relacional representa
o critério de frequência mínima...

1

frequencia>75 E ((notaP1+notaP2)/2>=6 OU notaExame>=7)


...e a combinação destas duas
expressões relacionais representa
o critério mínimo de nota.

Criando condições mais complexas

observe que **apenas uma** destas expressões precisa ser verdadeira para o critério de nota ser atendido:

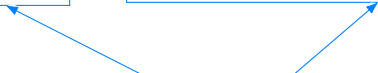
ou a média das provas superior a 6,
ou a nota do exame maior que 7...

1 `frequencia>75` E `((notaP1+notaP2)/2>=6 OU notaExame>=7)`



Criando condições mais complexas

1 `frequencia>75` E `((notaP1+notaP2)/2>=6 OU notaExame>=7)`

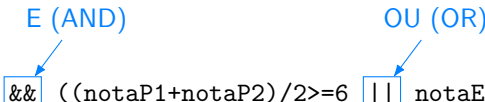


...por outro lado, é preciso que
a primeira E a segunda parte sejam verdadeiras
para que ocorra a aprovação

Operadores lógicos

Operadores lógicos

Para conseguir criar tais condições, vamos utilizar os **operadores lógicos**!



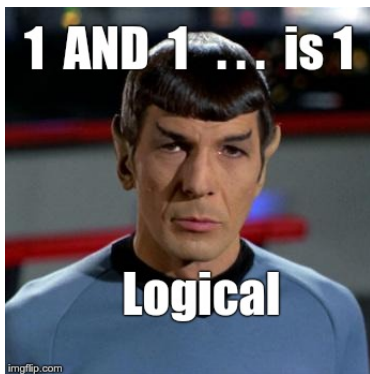
```
1  if(freq>75 && ((notaP1+notaP2)/2>=6 || notaExame>=7)
```

Os operadores lógicos devem utilizar como operandos expressões ou variáveis que produzam resultados lógicos (ou seja, verdadeiro ou falso). Eles são amplamente utilizados em tomadas de decisão e também em repetições.

Operadores lógicos

Na linguagem C, os operadores lógicos são:

| Operadores Lógicos | Descrição |
|-------------------------|---------------|
| <code>&&</code> | AND (E) |
| <code> </code> | OR (OU) |
| <code>!</code> | NOT (negação) |



OR (OU): tabela-verdade

`<valor ou expressao> || <valor ou expressao>`: retorna verdadeiro quando pelo menos uma das expressões (valores) é verdadeira. Sua tabela verdade é:

| Op_1 | Op_2 | Ret |
|------|------|-----|
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

AND (E): tabela-verdade

`<valor ou expressao> && <valor ou expressao>`: retorna verdadeiro quando ambas as expressões (ou valores) são verdadeiras. Sua tabela verdade é:

| Op_1 | Op_2 | Ret |
|------|------|-----|
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | F |

Em C, dada uma expressão `Exp1 && Exp2` em que `Exp1` é falsa, o restante não é avaliado.

NOT (NÃO): tabela-verdade

! <valor ou expressao>: Retorna verdadeiro quando a expressão (valor) é falsa e vice-versa. Sua tabela verdade é:

| Op_1 | Ret |
|------|-----|
| V | F |
| F | V |

Não “decore”: entenda!

É mais fácil compreender do que memorizar estas tabelas!! Pense no significado lógico de AND, OR e NOT.

Em resumo, os operadores lógicos AND e OR recebem 2 valores booleanos e produzem um terceiro valor booleano. O retorno é:

- AND – verdadeiro se os 2 valores forem verdadeiros
- OR – verdadeiro se pelo menos 1 dos valores for verdadeiro

O operador lógico NOT recebe e produz 1 valor booleano. O resultado é verdadeiro se o valor for falso, ou vice-versa.

Equivalências

Podemos chegar a equivalências, como:

- $!(a \& \& b)$ é equivalente a $!a \mid \mid !b$.
 - “Se não é verdade que ambos são verdadeiros, então pelo menos um deles é falso!”
- $!(a == b)$ é equivalente a $a != b$.
 - “Se não é verdade que a e b são iguais, então eles são diferentes!”
- Outras:
 - $!(a != b)$ e $a == b$;
 - $!(a > b)$ e $a <= b$.

Atenção!

Entretanto, vale lembrar: sempre procure utilizar a lógica mais simples e objetiva!

Operadores lógicos e valores booleanos

Assim como é o caso para operadores relacionais, ao criar uma expressão com operadores lógicos, o resultado pode ser **verdadeiro** ou **falso**.



Operadores lógicos e valores booleanos

Importante!!

Em C, não há constantes lógicas `true` e `false`: os operadores relacionais retornam 0 para **falso** e 1 para **verdadeiro** (embora qualquer valor diferente de 0 seja interpretado como verdadeiro!).

No exemplo abaixo, a mensagem `Oi!` sempre será impressa na tela! Como 8 é diferente de zero, a condição é considerada verdadeira!

```
1  ...
2
3  if( 5 + 3 )
4      printf("Oi");
5
6  ...
```

Operadores lógicos e valores booleanos

Portanto, como valores booleanos são representados como inteiros, os testes abaixo são equivalentes:

- `if (foo == 0)` é equivalente a `if (!foo)`
- `if (foo != 0)` é equivalente a `if (foo)`

Entretanto, cuidado! Lembre-se sempre que, na linguagem C, **qualquer coisa diferente de 0 é interpretada como verdadeiro!** Portanto...

`if (foo == 1)` **não é equivalente a** `if (foo)`

Isso porque se a variável `foo` tiver outro outro valor valor diferente de 0, a condição também será verdadeira.

Operadores lógicos e valores booleanos

Portanto, como valores booleanos são representados como inteiros, os testes abaixo são equivalentes:

- `if (foo == 0)` é equivalente a `if (!foo)`
- `if (foo != 0)` é equivalente a `if (foo)`

Entretanto, cuidado! Lembre-se sempre que, na linguagem C, **qualquer coisa diferente de 0 é interpretada como verdadeiro!** Portanto...

`if (foo == 1)` **não é equivalente a** `if (foo)`

Isso porque se a variável `foo` tiver outro outro valor valor diferente de 0, a condição também será verdadeira.

Um exemplo clássico...

Um ano é bissexto quando é divisível por 400, ou quando é divisível por 4 mas não por 100. Como testar se um ano dado é bissexto?

A resposta longa...

```
1
2 if (ano % 400 == 0) /* Divisível por 400. */
3     printf ("Eh bissexto\n");
4 else if (ano % 4 == 0) /* Divisível por 4. */
5 {
6     if (ano % 100 == 0) /* Divisível por 100. */
7         printf ("Nao eh bissexto\n");
8     else
9         printf ("Eh bissexto\n");
10 }
11 else
12     printf ("Nao eh bissexto\n");
```

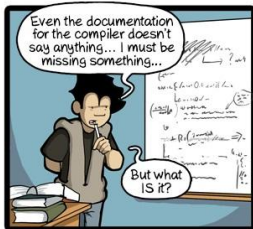
... a resposta compacta ...

```
1
2 // Divisível por 400,
3 //OU divisível por 4 E não divisível por 100.
4 if ((ano%400 == 0) || ((ano%4 == 0)&&(ano%100 != 0)))
5     printf ("Eh bissexto\n");
6 else
7     printf ("Nao eh bissexto\n");
```

... e a resposta mais compacta ainda!

```
1
2 // Divisível por 400,
3 //OU divisível por 4 E não divisível por 100.
4 if (!(ano%400) || (!(ano%4)&&(ano%100)))
5     printf ("Eh bissexto\n");
6 else
7     printf ("Nao eh bissexto\n");
```

Cuidado com condições “sem sentido”



CommitStrip.com

Exemplos de condições sem sentido

Outros exemplos:

```
if (x > y || x <= y) // Sempre verdadeira
```

```
if (x > 10 && x <= 3) // Nunca é verdadeira
```

```
if (x || !x) // Sempre é verdadeira
```



Cuidado ao combinar expressões relacionais

Suponha que precisamos comparar se uma idade está entre 18 e 60. A seguinte comparação gera um erro de execução:

```
if(18 <= idade <= 60)
```

Neste caso as comparações são realizadas da esquerda para a direita: `18 < idade` será avaliado como verdadeiro (retornando 1) ou falso (0). Portanto, o que será comparado com o valor 60 será 1 ou 0, e não o conteúdo de `idade`, como esperado.

A solução!

Portanto, ao comparar um valor com outros dois (operadores relacionais), é preciso separar as comparações com operadores lógicos. Para exemplo anterior: `if(18 <= idade && idade <= 60)`

Cuidado ao combinar expressões relacionais

Suponha que precisamos comparar se uma idade está entre 18 e 60. A seguinte comparação gera um erro de execução:

```
if(18 <= idade <= 60)
```

Neste caso as comparações são realizadas da esquerda para a direita: `18 < idade` será avaliado como verdadeiro (retornando 1) ou falso (0). Portanto, o que será comparado com o valor 60 será 1 ou 0, e não o conteúdo de `idade`, como esperado.

A solução!

Portanto, ao comparar um valor com outros dois (operadores relacionais), é preciso separar as comparações com operadores lógicos. Para exemplo anterior: `if(18 <= idade && idade <= 60)`

Precedência e avaliação de operadores

Nos exemplos anteriores, observe que intuitivamente avaliamos os operadores relacionais antes dos lógicos! Agora vamos formalizar isso!

Precedência de operadores

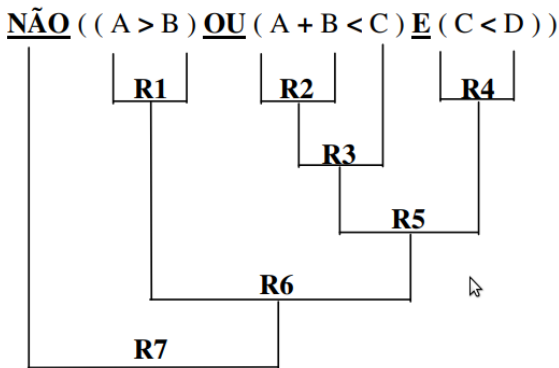
Como já vimos, existe uma ordem de prioridade de cálculo em uma expressão. Ao considerar também operadores lógicos temos:

- ① !
- ② *, /, %
- ③ +, -
- ④ <, >, <=, >=
- ⑤ ==, !=
- ⑥ &&
- ⑦ ||

Esta ordem de prioridade só pode ser quebrada com o uso de parênteses.

Exemplos

Para a expressão **NÃO** ((A > B) **OU** (A + B < C) **E** (C < D)) a ordem de avaliação seria:



<http://www.din.uem.br/~yandre/>

Apesar do operador **NÃO** ter a maior prioridade, ele foi aplicado por último devido ao uso de parênteses.

Exemplos

```
!(!(true || false) && (false && true))
```

Solução:

```
!(! true && (false && true))
```

```
!(! true && false)
```

```
!(false && false)
```

```
!false
```

```
true
```

Exemplos

```
!(5 != 10 / 2 || true && 2 - 5 > 5 - 2 || true)
```

Solução :

```
!(5 != 5 || true && -3 > 3 || true)
```

```
!(false || true && false || true)
```

```
!(false || false || true)
```

```
!(false || true)
```

```
!true
```

```
false
```


Avaliação de operadores lógicos

Muitos compiladores só avaliam uma expressão contendo operadores lógicos até o seu resultado ser conhecido.

Por exemplo, na expressão `a || ((b > 10) && c)`:

- Se `a` for diferente de 0, já sabemos que a expressão é verdadeira, e o termo `((b > 10) && c)` não precisa ser avaliado.
- Se `a` for igual a 0 e `b` for menor que 10, já sabemos que a expressão é falsa, e o termo `c` não precisa ser avaliado.

Precedência *versus* legibilidade

Conhecendo as precedências, sabemos a ordem em que as operações são executadas. Entretanto, muitas vezes o uso de parênteses para agrupar as expressões pode ajudar **muito** na legibilidade. Considere a seguinte expressão lógica:

```
x < 10 || x < y * 4 && x - 3 > 6 + y / 2
```

Neste caso, é mais fácil compreender o que está sendo executado a partir da seguinte expressão:

```
x < 10 || ((x < y * 4) && (x - 3 > 6 + y / 2))
```

1

¹A discussão continua no próximo slide...

Precedência *versus* legibilidade

O exemplo anterior ilustra um erro de interpretação muito comum: como o operador `&&` tem precedência sobre o `||`, a ordem de avaliação é

```
EXP1 || (EXP2 && EXP3)
```

... e não lendo os operadores lógicos da esquerda para a direita...
Se você quiser este efeito, precisa fazer

```
(EXP1 || EXP2) && EXP3
```

Profe, mas faz diferença? Me diga você: `1 || (1&&0)` é equivalente a `(1 || 1)&&0`?

Precedência *versus* legibilidade

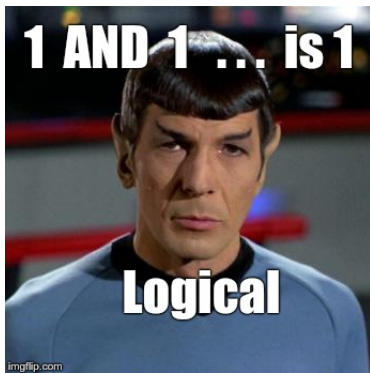
Entretanto, não exagere no uso de parênteses... Isso também compromete a legibilidade do código!!

Não faça isso!!!

```
((x) < 10) || ((x < (y*4)) && ((x-3) > ((6+y) / 2)))
```

Por fim...

Agora o meme faz sentido, não é?



Até a próxima...