

C++

Básico ao Avançado

Qual o resultado?

Heitor Rodrigues Savegnago

UFABC Rocket Design

2017.3

1 Alterações

2 Unário

3 Binário

4 Ternário

5 Precedência

6 Hora de brincar

Modificando variáveis

Modificando variáveis

- Variáveis que não variam não são variáveis

Modificando variáveis

- Variáveis que não variam não são variáveis
- As alterações nas variáveis dependem de seu tipo

Modificando variáveis

- Variáveis que não variam não são variáveis
- As alterações nas variáveis dependem de seu tipo
- Existem três grupos de operadores: unários, binários e ternários:

Modificando variáveis

- Variáveis que não variam não são variáveis
- As alterações nas variáveis dependem de seu tipo
- Existem três grupos de operadores: unários, binários e ternários:

Unário utiliza apenas um valor

Modificando variáveis

- Variáveis que não variam não são variáveis
- As alterações nas variáveis dependem de seu tipo
- Existem três grupos de operadores: unários, binários e ternários:

Unário utiliza apenas um valor

Binário utiliza dois valores

Modificando variáveis

- Variáveis que não variam não são variáveis
- As alterações nas variáveis dependem de seu tipo
- Existem três grupos de operadores: unários, binários e ternários:

Unário utiliza apenas um valor

Binário utiliza dois valores

Ternário utiliza três valores

Modificando variáveis

- Variáveis que não variam não são variáveis
- As alterações nas variáveis dependem de seu tipo
- Existem três grupos de operadores: unários, binários e ternários:

Unário utiliza apenas um valor

Binário utiliza dois valores

Ternário utiliza três valores

- Todo operador retorna o valor de sua operação

Incrementos e decrementos unitários

Incrementos e decrementos unitários

- Variáveis podem ter seu valor acrescido ou decrescido

Incrementos e decrementos unitários

- Variáveis podem ter seu valor acrescido ou decrescido
- Estes operadores unitários o fazem com o valor de 1

Incrementos e decrementos unitários

- Variáveis podem ter seu valor acrescido ou decrescido
- Estes operadores unitários o fazem com o valor de 1
- Podem ser prefixos ou sufixos

Incrementos e decrementos unitários

- Variáveis podem ter seu valor acrescido ou decrescido
- Estes operadores unitários o fazem com o valor de 1
- Podem ser prefixos ou sufixos
- Alteram o valor registrado

Incrementos e decrementos unitários

- Variáveis podem ter seu valor acrescido ou decrescido
- Estes operadores unitários o fazem com o valor de 1
- Podem ser prefixos ou sufixos
- Alteram o valor registrado

```
<nome>++; //Incremento posfixo
++<nome>; //Incremento prefixo

<nome>--; //Decremento posfixo
--<nome>; //Decremento prefixo
```


Incrementos e decrementos unitários

```
1 //...
2 int A(5);    //A vale 5
3 int B(A++);  //A vale 6, B vale 5
4 int C(++B);  //A vale 6, B vale 6, C vale 6
5 C++;         //Também pode ser usado independente do valor de
               retorno
6
7 int D(5);    //D vale 5
8 int E(--D);  //D vale 4, E vale 4
9 int F(E--);  //D vale 4, E vale 3, F vale 4
10 //...
```

Sinalizadores

Sinalizadores

- Equivalem aos sinais matemáticos

Sinalizadores

- Equivalem aos sinais matemáticos
- Fazem o mesmo que multiplicar um número por $+1$ ou -1

Sinalizadores

- Equivalem aos sinais matemáticos
- Fazem o mesmo que multiplicar um número por $+1$ ou -1
- Não alteram o valor registrado

Sinalizadores

- Equivalem aos sinais matemáticos
- Fazem o mesmo que multiplicar um número por +1 ou −1
- Não alteram o valor registrado

```
-<name>; //Só aparece em um contexto onde o retorno é
          utilizado
+<name>;
```

```
1 //...
2 int A(-5);    //A vale -5
3 int B(+A);    //A vale -5, B vale -5
4 int C(-A);    //A vale -5, B vale -5, C vale 5
5
6 int D(5);     //D vale 5,
7 int E(-D);    //D vale 5, E vale -5
8 int F(+D);    //D vale 5, E vale -5, F vale 5
9 //...
```

Negador booleano

Negador booleano

- Retorna o estado inverso de um `bool`

Negador booleano

- Retorna o estado inverso de um `bool`
- Só pode ser usado no tipo `bool`

Negador booleano

- Retorna o estado inverso de um `bool`
- Só pode ser usado no tipo `bool` (Mas funciona em outros tipos)

Negador booleano

- Retorna o estado inverso de um `bool`
- Só pode ser usado no tipo `bool` (Mas funciona em outros tipos)
- Não confundir com fatorial!

Negador booleano

- Retorna o estado inverso de um `bool`
- Só pode ser usado no tipo `bool` (Mas funciona em outros tipos)
- Não confundir com fatorial!
- Não alteram o valor registrado

Negador booleano

- Retorna o estado inverso de um `bool`
- Só pode ser usado no tipo `bool` (Mas funciona em outros tipos)
- Não confundir com fatorial!
- Não alteram o valor registrado

```
!<nome>; //Seu uso só é coerente quando o retorno é
          utilizado
```

Negador booleano

```
1 //...
2 bool A(true);    //A vale 1
3 bool B(!A);      //B vale 0
4 bool C(!B);      //C vale 1
5
6 bool D(!true);   //D vale 1
7 bool E(!false);  //E vale 0
8
9 int F(10);       //F vale 10
10 int G(!F);       //G vale 0
11 int H(!G);       //H vale 1
12 //...
```

Complemento binário

Complemento binário

- Este operador inverte todos os bits de uma variável

Complemento binário

- Este operador inverte todos os bits de uma variável
- Em tipos `signed` o sinal é invertido

Complemento binário

- Este operador inverte todos os bits de uma variável
- Em tipos `signed` o sinal é invertido
- Não pode ser utilizado em números flutuantes

Complemento binário

- Este operador inverte todos os bits de uma variável
- Em tipos `signed` o sinal é invertido
- Não pode ser utilizado em números flutuantes
- Não alteram o valor registrado

Complemento binário

- Este operador inverte todos os bits de uma variável
- Em tipos `signed` o sinal é invertido
- Não pode ser utilizado em números flutuantes
- Não alteram o valor registrado

```
~<nome>; //Novamente, seu uso só é coerente se o retorno é  
utilizado
```

Complemento binário

```
1 //..
2 unsigned char A(0b10100101); //A vale 0b10100101 ou 0xA5
3 unsigned char B(~A); //B vale 0b01011010 ou 0x5A
4 /*
5 unsigned char C(Ã); //Tome cuidado para isso não
6 //acontecer
7 */
8 //..
```

Atribuidor simples

Atribuidor simples

- O operador mais utilizado é o atribuidor simples

Atribuidor simples

- O operador mais utilizado é o atribuidor simples
- Ele atribui o valor a direita à variável a direita

Atribuidor simples

- O operador mais utilizado é o atribuidor simples
- Ele atribui o valor a direita à variável a direita
- Cuidado para não inverter

Atribuidor simples

- O operador mais utilizado é o atribuidor simples
- Ele atribui o valor a direita à variável a direita
- Cuidado para não inverter
- Não atribuir tipos a variáveis de outros tipos

Atribuidor simples

- O operador mais utilizado é o atribuidor simples
- Ele atribui o valor a direita à variável a direita
- Cuidado para não inverter
- Não atribuir tipos a variáveis de outros tipos

```
<alvo> = <item>;
```

Atribuidor simples

```
1 //...
2 int A;
3 A = 10; //A passa a valer 10
4 float B = 5.1; //Operadores podem ser usados na declaração.
5
6 float C(B = 13.25); //Todo operador retorna o valor de sua operação
7
8 int D = A = 20;
9 /*
10     Da direita para a esquerda para a direita, A passa a valer 10
11     Então D passa a ter o valor da operação à direita, 10
12 */
13 //...
```

Aritméticos

Aritméticos

- ## ■ As quatro operações básicas da matemática

Aritméticos

- As quatro operações básicas da matemática
- Os símbolos padrão

Aritméticos

- As quatro operações básicas da matemática
- Os símbolos padrão
- Não alteram o valor registrado

Aritméticos

- As quatro operações básicas da matemática
- Os símbolos padrão
- Não alteram o valor registrado
- Cuidado para não dividir por 0

Aritméticos

- As quatro operações básicas da matemática
- Os símbolos padrão
- Não alteram o valor registrado
- Cuidado para não dividir por 0

```
<valor1> + <valor2>; //0 uso só é coerente em casos onde o
    retorno é utilizado
<valor1> - <valor2>;
<valor1> * <valor2>;
<valor1> / <valor2>;
<valor1> % <valor2>;
```

Aritméticos

- As quatro operações básicas da matemática
- Os símbolos padrão
- Não alteram o valor registrado
- Cuidado para não dividir por 0

```

<valor1> + <valor2>; //O uso só é coerente em casos onde o
    retorno é utilizado
<valor1> - <valor2>;
<valor1> * <valor2>;
<valor1> / <valor2>;
<valor1> % <valor2>;
    
```

$$\begin{array}{c|c} E & F \\ \hline G & H \end{array} \Rightarrow \begin{array}{c|c} 13 & 5 \\ \hline 3 & 2 \end{array}$$

UFABC Rocket Design

Deslocadores

Deslocadores

- Os deslocadores são operações bit-a-bit

Deslocadores

- Os deslocadores são operações bit-a-bit
- São operadores que multiplicam o valor registrado por potências de 2

Deslocadores

- Os deslocadores são operações bit-a-bit
- São operadores que multiplicam o valor registrado por potências de 2
- Não alteram o valor registrado

Deslocadores

- Os deslocadores são operações bit-a-bit
- São operadores que multiplicam o valor registrado por potências de 2
- Não alteram o valor registrado
- É mais rápido do que uma multiplicação comum

0770

$V \cdot 2^{-s}$

Deslocadores

```
1 //...
2 unsigned char A(0b01100000);
3 unsigned char B(A>>3); //B vale 0b00001100
4 unsigned char C(B*8); //C vale 0b01100000
5
6 int D(30); //D vale 30
7 int E(D>>1); //E vale 30/(2), ou seja 15
8 int F(D<<2); //F vale 30*(4), ou seja, 120
9 //...
```

Lógicos bit-a-bit

Lógicos bit-a-bit

- Operadores lógicos normais

Lógicos bit-a-bit

- Operadores lógicos normais
- Trabalham separadamente a cada bit

Lógicos bit-a-bit

- Operadores lógicos normais
- Trabalham separadamente a cada bit

```
<valor1> | <valor2>;  
<valor1> & <valor2>;  
<valor1> ^ <valor2>;
```


Lógicos bit-a-bit

- Operadores lógicos normais
- Trabalham separadamente a cada bit

```
<valor1> | <valor2>;  
<valor1> & <valor2>;  
<valor1> ^ <valor2>;
```

Tabela: Tabela verdade para operadores lógicos

A	B	NOT A	NOT B	A OR B	A AND B	A XOR B
0	0	1	1	0	0	0
0	1	1	0	1	0	1
1	0	0	1	1	0	1
1	1	0	0	1	1	0

Lógicos bit-a-bit

```
1 //...
2 unsigned char A(0b10101011);
3 unsigned char B(0b01100100);
4 unsigned char C(A|B);           //C vale 0b11101111
5 unsigned char D(A&B);          //D vale 0b00100000
6 unsigned char E(A^B);          //E vale 0b11001111
7 //...
```

Lógicos bit-a-bit

```
1 //...
2 unsigned char A(0b10101011);
3 unsigned char B(0b01100100);
4 unsigned char C(A|B);           //C vale 0b11101111
5 unsigned char D(A&B);           //D vale 0b00100000
6 unsigned char E(A^B);           //E vale 0b11001111
7 //...
```

10101011

OR

01100100

11101111

10101011

AND

01100100

00100000

10101011

XOR

01100100

11001111

Atribuidor composto

Atribuidor composto

- Uma simplificação de operações que atribuem valores

Atribuidor composto

- Uma simplificação de operações que atribuem valores
- Operações mais simples são sempre escritas desta maneira

Atribuidor composto

- Uma simplificação de operações que atribuem valores
- Operações mais simples são sempre escritas desta maneira

```
<nome> = <nome> <operador> <valor>; //Onde nome é uma variável
    vel
<nome> <operador>= <valor>;           //Operador de atribuição
    composta
//...
```

Atribuidor composto

Tabela: Relação de operadores de atribuição composta e seus equivalentes

Composto				Equivalente			
A	+=	B;	⇔	A	=	A +	B;
A	-=	B;	⇔	A	=	A -	B;
A	*=	B;	⇔	A	=	A *	B;
A	/=	B;	⇔	A	=	A /	B;
A	%=	B;	⇔	A	=	A %	B;
A	>>=	B;	⇔	A	=	A >>	B;
A	<<=	B;	⇔	A	=	A <<	B;
A	=	B;	⇔	A	=	A	B;
A	&=	B;	⇔	A	=	A &	B;
A	^=	B;	⇔	A	=	A ^	B;

Atribuidor composto

```

1  //...
2  int A(0), B(10);    //A vale 0,    B vale 10
3  A += 1;             //A vale 1,    B vale 10
4  B /= 2;             //A vale 1,    B vale 5
5  A *= 100;           //A vale 100, B vale 5
6  B <=< 3;             //A vale 100, B vale 40
7  B &= A;              //A vale 100, B vale 32
8  A %= B;             //A vale 4,    B vale 32
9  //...

```

Comparadores

Comparadores

- Verificar se valores são iguais ou diferentes

Comparadores

- Verificar se valores são iguais ou diferentes
- Descobrir se um valor é maior que outro

Comparadores

- Verificar se valores são iguais ou diferentes
- Descobrir se um valor é maior que outro

```
<valor1> == <valor2>; //Só é coerente quando o retorno é
    utilizado
<valor1> != <valor2>;
<valor1> < <valor2>;
<valor1> > <valor2>;
<valor1> <= <valor2>;
<valor1> >= <valor2>;
```

Comparadores

```

1  //...
2  bool A(true);
3  int B(10), C(10), D(15);
4
5  bool E(B>C);    //E vale 0
6  bool F(A==E);   //F vale 0
7
8  bool G(D>=B);   //G vale 1
9  bool H(E!=G);   //H vale 1
10 bool I(B==C);   //I vale 1
11 //...

```

Lógicos Booleanos

Lógicos Booleanos

- Servem para fazer junção de tipos `bool`

Lógicos Booleanos

- Servem para fazer junção de tipos `bool`
- Montar expressões de dependências lógicas mais compostas

Lógicos Booleanos

- Servem para fazer junção de tipos `bool`
- Montar expressões de dependências lógicas mais compostas

```
<valor1> || <valor2>; //Coerente apenas quando o retorno é utilizado  
<valor1> && <valor2>;
```

Lógicos Booleanos

```
1 //...
2 bool T(true), F(false);
3
4 bool A(T || F); //A vale 1
5 bool B(T && F); //B vale 0
6 //...
```

Operador ternário

Operador ternário

- Não tem nome próprio

Operador ternário

- Não tem nome próprio ☹

Operador ternário

- Não tem nome próprio ☹
- Faz escolhas a partir de decisões

Operador ternário

- Não tem nome próprio ☹
- Faz escolhas a partir de decisões
- Não altera o fluxo do código

Operador ternário

- Não tem nome próprio ☹
- Faz escolhas a partir de decisões
- Não altera o fluxo do código
- É simpático

Operador ternário

- Não tem nome próprio ☹
- Faz escolhas a partir de decisões
- Não altera o fluxo do código
- É simpático

```
<condicional> ? <valor1> : <valor2>;
```

Operador ternário

```
1 //...
2 bool A(true);
3 int C(10), D(15);
4 bool E(C>=D);      //E vale 0
5 bool F(A==E);      //F vale 0
6 int G(F?10:50);    //G vale 50
7 ///...
```

Precedência

Precedência

- Os operadores tem preferência de ordem

Precedência

- Os operadores tem preferência de ordem
- $1 + 1 + 1 + 1 * 0 = ?$

Precedência

- Os operadores tem preferência de ordem
- $1 + 1 + 1 + 1 * 0 = ?$

```
A = 10 + 10 - 100 / 5;
```

Precedência

- Os operadores tem preferência de ordem
- $1 + 1 + 1 + 1 * 0 = ?$

```
A = 10 + 10 - 100 / 5;
```

```
B = - 2 * - 2 + 12 << 40 / 20 + 30 % 8;
```


Precedência

Precedência

- Ambiguidades

Precedência

- Ambiguidades
- Mudando a precedência

Precedência

- Ambiguidades
- Mudando a precedência
- Operador de preferência

Precedência

- Ambiguidades
- Mudando a precedência
- Operador de preferência
- Igual a matemática

Precedência

Tabela: Ordem de precedência de operadores

Operador	Descrição
()	preferencial

Precedência

Tabela: Ordem de precedência de operadores

Operador	Descrição
()	preferencial
++, --	posfixo

Precedência

Tabela: Ordem de precedência de operadores

Operador	Descrição
()	preferencial
++, --	posfixo
++, --	prefixo

Precedência

Tabela: Ordem de precedência de operadores

Operador	Descrição
()	preferencial
++, --	posfixo
++, --	prefixo
~, !	lógico
+, -	senalizadore

Precedência

Tabela: Ordem de precedência de operadores

Operador	Descrição
()	preferencial
++, --	posfixo
++, --	prefixo
~, !	lógico
+, -	senalizador
*, /, %	aritimético
+, -	aritimético
<<, >>	deslocador

Operador	Descrição
()	preferencial
++, --	posfixo
++, --	prefixo
~, !	lógico
+, -	senalizadore
*, /, %	aritmético
+, -	aritmético
<<, >>	deslocador
<, <=, >=, >	comparador
==, !=	comparador

Precedência

Tabela: Ordem de precedência de operadores

Operador	Descrição
()	preferencial
++, --	posfixo
++, --	prefixo
~, !	lógico
+, -	senalizadore
*, /, %	aritimético
+, -	aritimético
<<, >>	deslocador
<, <=, >=, >	comparador
==, !=	comparador
&	lógico
^	lógico
	lógico
&&	lógico
	lógico

Precedência

Tabela: Ordem de precedência de operadores

Operador	Descrição
()	preferencial
++, --	posfixo
++, --	prefixo
~, !	lógico
+, -	sinalizadore
*, /, %	aritimético
+, -	aritimético
<<, >>	deslocador
<, <=, >=, >	comparador
==, !=	comparador
&	lógico
^	lógico
	lógico
&&	lógico
	lógico
?:	ternário
=	atribuidor
+=, -=, *=, /=, %=, &=, ^=, =, <<=, >>=	atribuidor


```
1 //...
2 int A = 25 * 40;
3 int B = 1 << 4;
4 A /= B + 4;
5
6 A = A + B;
7 B = A - B;
8 A = A - B;
9
10 float C = A > 200 ? A * (50.0f - 0.003f) : B % 5;
11 //...
```

Vamos testar!