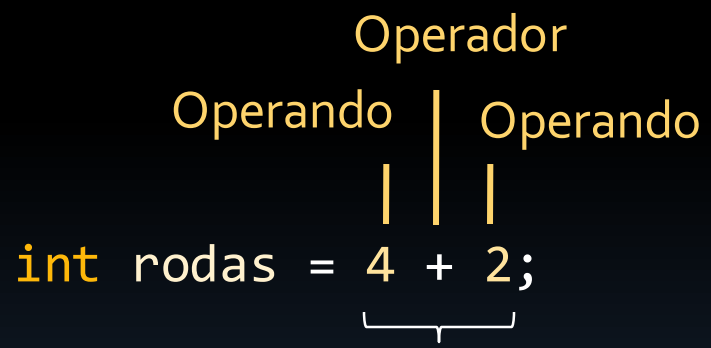
Tipos Básicos de Dados

OPERADORES ARITMÉTICOS

Judson Santiago

Introdução

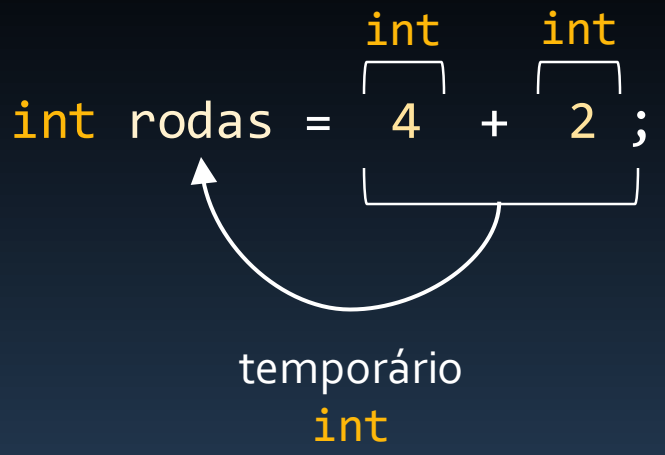
- C++ fornece operadores para cinco cálculos aritméticos básicos:
  - Soma (+)
  - Subtração (-)
  - Multiplicação (*)
  - Divisão (/)
  - Módulo (%)

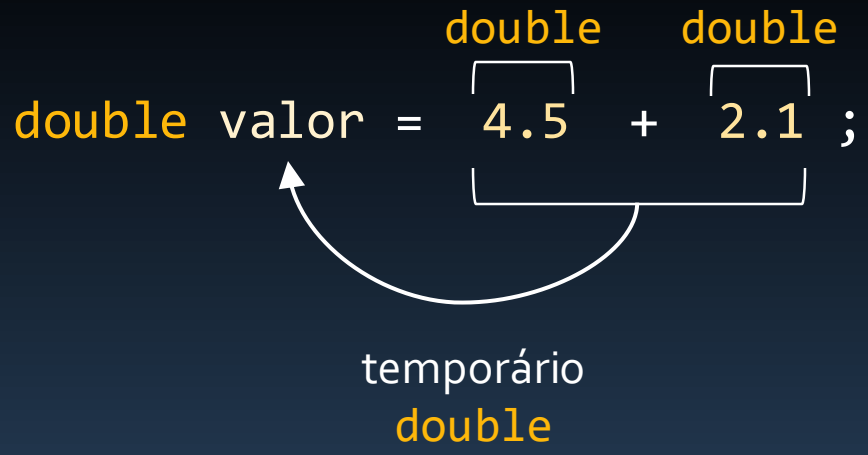


O operador e seus operandos formam uma expressão.

Introdução

- O resultado de uma expressão é armazenado em um local temporário de memória
  - O tipo dessa memória depende dos operandos
  - A atribuição é efetuada após a avaliação da expressão





Operadores Aritméticos

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

int main()
{
    float num1, num2;

    cout << "Entre com um número: ";
    cin >> num1;
    cout << "Entre com outro número: ";
    cin >> num2;

    cout << "num1 = " << num1 << "; num2 = " << num2 << endl;
    cout << "num1 + num2 = " << num1 + num2 << endl;
    cout << "num1 - num2 = " << num1 - num2 << endl;
    cout << "num1 * num2 = " << num1 * num2 << endl;
    cout << "num1 / num2 = " << num1 / num2 << endl;
    return 0;
}
```

Operadores.cpp

Operadores Aritméticos

- A saída do programa:

```
Entre com um número: 50.25
Entre com outro número: 11.17
num1 = 50.25; num2 = 11.17

num1 + num2 = 61.42
num1 - num2 = 39.08
num1 * num2 = 561.292      // 561.2925
num1 / num2 = 4.49866      // 4.498657117278424
```

  - Por padrão, cout mostra até 6 dígitos significativos
    - Arredonda se o valor possuir mais que 6 dígitos
    - Passa para notação científica sob certas condições

Precedência de Operadores

- Qual o resultado da expressão abaixo?

```
int total = 3 + 4 * 5;  // 35 ou 23
```

- Quando mais de um operador pode ser aplicado ao mesmo operando, C++ usa regras de precedência para decidir:

```
1º) Multiplicação / Divisão / Módulo
2º) Soma / Subtração

int total = 3 + 4 * 5;  // 3 + (4 * 5) = 23
```

Associatividade de Operadores

- Qual o resultado da expressão abaixo?

```
int total = 120 / 4 * 5;  // 150 ou 6
```

- Se os operadores têm a mesma precedência, C++ usa regras de associatividade (esquerda ou direita) :

Todos os operadores são associativos à esquerda:
Soma / Subtração / Multiplicação / Divisão / Módulo

```
int total = 120 / 4 * 5;  // (120 / 4) * 5 = 150
```

Ordem de Avaliação

- A ordem de avaliação dos operandos é independente da precedência e da associatividade dos operadores

```
int total = f() + g() * h() - i();
```

  - A precedência garante que os resultados de g() e h() serão multiplicados primeiro
  - A associatividade garante que f() será somado ao produto de g() com h() e que esse resultado será subtraído do valor de i()
  - **Não existe garantia para a ordem de chamada das funções**

Precedência e Associatividade

- Tabela de precedência dos operadores aritméticos:
  - Operadores agrupados por precedência
  - Associatividade da esquerda para direita

Operador	Função	Uso
+ -	mais unário menos unário	+expr -expr
* / %	multiplicação divisão módulo	expr * expr expr / expr expr % expr
+ -	adição subtração	expr + expr expr - expr

Operador de Divisão

- O resultado do operador de divisão depende dos operandos
  - Divisão inteira:
    se ambos os operandos são inteiros

```
5 / 2   // o resultado é 2 e não 2.5
```

  - Divisão ponto flutuante:
    se pelo menos um operando é ponto flutuante

```
5.0 / 2      // o resultado é 2.5
5 / 2.0      // o resultado é 2.5
5.0 / 2.0    // o resultado é 2.5
```

Operador de Divisão

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

int main()
{
    // sempre mostra 6 casas após a vírgula
    // cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << fixed;

    cout << "Divisão Inteira: 9/5 = " << 9/5 << endl;
    cout << "Divisão Ponto-Flutuante: 9.0/5.0 = " << 9.0/5.0 << endl;
    cout << "Divisão Mista: 9.0/5 = " << 9.0/5 << endl;

    cout << endl;

    cout << "Constantes double: 1e7/9.0 = " << 1e7/9.0 << endl;
    cout << "Constantes float: 1e7f/9.0f = " << 1e7f/9.0f << endl;
}
```

Divisao.cpp

Operador de Divisão

- A saída do programa:

```
Divisão Inteira: 9/5 = 1
Divisão Ponto-Flutuante: 9.0/5.0 = 1.800000
Divisão Mista: 9.0/5 = 1.800000

Constantes double: 1e7/9.0 = 1111111.111111
Constantes float: 1e7f/9.0f = 1111111.125000
```

- O resultado é double se pelo menos um dos operandos é double e float caso contrário
- O tipo padrão das constantes é double

Operador Módulo

- O operador módulo (%) retorna o resto de uma divisão inteira

```
5 % 2 = 1
```

```
 5 | 2
-4   2
 1
```

```
5 % 3 = 2
```

```
 5 | 3
-3   1
 2
```

```
5 % 6 = 5
```

```
 5 | 6
-0   0
 5
```

Resto da divisão

Operador Módulo

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

int main()
{
	const int CentavosPorReal = 100;
	int valor;

	cout << "Digite um valor em centavos: ";
	cin >> valor;

	int reais = valor / CentavosPorReal;
	int centavos = valor % CentavosPorReal;

	cout << valor << " centavos correspondem a\n" << reais << " Reais e "
		<< centavos << " centavos." << endl;

	return 0;
}
```

Modulo.cpp

Operador Módulo

- A saída do programa:

```
Digite um valor em centavos: 210
210 centavos correspondem a
2 Reais e 10 centavos
```

- Constantes devem ser inicializadas e não podem ter seu valor alterado no programa

```
const int CentavosPorReal = 100;
CentavosPorReal = 200;    X

const int CargaHoraria;   // deve ser inicializado
CargaHoraria = 60;        X
```

#define *versus* const

- Um #define cria uma constante simbólica

```
#define CentavosPorReal 100
const int CentavosPorReal = 100;
```

- Um const é melhor porque:
  - Possui características semelhantes às variáveis
    - Possui um tipo, é possível pegar o seu endereço, pode ser passado para funções, aceita conversões, etc.
  - Possui um escopo
  - Passa pela verificação de tipos

Aplicação do Módulo

- O módulo pode atuar como limitador de um resultado

```
0 % 5 = 0    5 % 5 = 0    10 % 5 = 0    15 % 5 = 0
1 % 5 = 1    6 % 5 = 1    11 % 5 = 1    16 % 5 = 1
2 % 5 = 2    7 % 5 = 2    12 % 5 = 2    17 % 5 = 2
3 % 5 = 3    8 % 5 = 3    13 % 5 = 3    18 % 5 = 3
4 % 5 = 4    9 % 5 = 4    14 % 5 = 4    19 % 5 = 4
```

O resultado de (valor % n) fica sempre na faixa de 0 a n-1

Aplicação do Módulo

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

int main()
{
     cout << "Entre com os valores min e max:\n";
     int min, max;
     cin >> min;
     cin >> max;

     cout << "Sorteando um valor nesta faixa:\n";
     int sorteio = min + rand() % (max - min + 1);
     cout << sorteio << endl;
}
```

RandMod.cpp

Aplicação do Módulo

- A saída do programa:

```
Entre com os valores min e max:
10 20
Sorteando um valor nesta faixa:
18
```

- A função rand( ) retorna valores entre 0 e 32767 (RAND_MAX)
- O módulo coloca estes valores na faixa de 0 a n-1

```
int sorteio = min + rand() % (max - min + 1);
                            \_____________/
                                  n
```

Conversões de Tipo

- A existência de muitos tipos de dados permite ao programador usar o que for mais adequado as suas necessidades
  - bool, short, int, long, long long, char, unsigned char, unsigned short, unsigned int, unsigned long, unsigned long long, float, double, long double

- Para facilitar, a linguagem C++ faz conversões automáticas de tipos em 3 situações

Conversões de Tipo

- Conversões automáticas são feitas:
  - Em atribuições de valores à variáveis, quando o valor é de um tipo diferente da variável

```
int valor = 2.5;                // 2.5 → 2
float resultado = 10;           // 10  → 10.0
```

  - Em expressões, quando se combinam valores e/ou variáveis de tipos diferentes

```
int total = 2 + 3.5 + 1;        // 6.5 → 6
float resultado = 11 / 2.0;     // 5.5 → 5.5f
```

Conversões de Tipo

- Conversões automáticas são feitas:
  - Na passagem de argumentos para funções, quando os argumentos tem tipos diferentes dos parâmetros da função

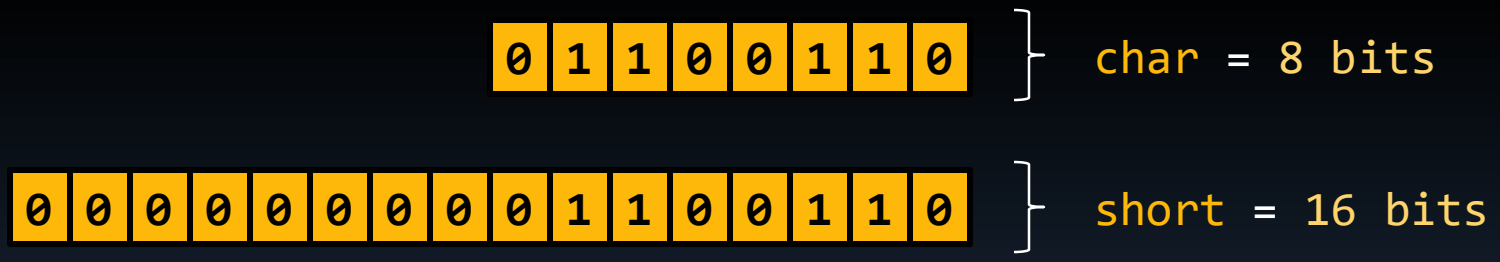
```
double soma (double, double);    // protótipo da função

soma(3, 5);   // chamada da função: 3 → 3.0, 5 → 5.0
```

- Para entender o resultado de alguns programas é preciso entender as conversões

Conversões na Atribuição

- C++ é bastante liberal na atribuição de valores numéricos

```
char  mar = 102;
short sol = mar;  // o tipo char é convertido em short
```



  - Atribuir valor para uma variável de maior capacidade não gera nenhum problema

Conversões na Atribuição

- É preciso tomar cuidado com atribuições para tipos com menor capacidade

```
short sol = 280;
char  mar = sol;   // o valor armazenado é 24
```

  - Apenas os bits de mais baixa ordem são copiados quando o tipo de destino tem uma capacidade menor

Conversões na Atribuição

Tipo de Conversão	Problema Potencial
Tipo ponto flutuante maior para tipo ponto flutuante menor Ex.: double pra float	• Perda de precisão (dígitos significativos) • Valor pode estar fora da faixa do tipo alvo, e neste caso o resultado é indefinido
Tipo ponto flutuante para tipo inteiro Ex.: double para int	• Perda da parte fracionária • Valor pode estar fora da faixa do tipo alvo, e neste caso o resultado é indefinido
Tipo inteiro maior para tipo inteiro menor Ex.: long pra short	• Valor original pode estar fora da faixa para o tipo alvo: apenas os bits de mais baixa ordem são copiados

Conversões na Atribuição

```cpp
#include <iostream>
using namespace std;
int main()
{
     // cout.setf(ios_base::fixed, ios_base::floatfield);
     cout << fixed;

     float tres = 3;          // int convertido para float
     int aposta = 3.9832;     // double convertido para int
     int debito = 7.2E12;     // resultado não definido

     cout << "tres = "   << tres << endl;
     cout << "aposta = " << aposta << endl;
     cout << "debito = " << debito << endl;
}
```

ConvAtrib.cpp

Conversões na Atribuição

- A saída do programa:

```
tres: 3.000000
aposta: 3
debito: 1634811904
```

  - O valor inteiro foi convertido para float
  - O valor float foi truncado (e não arredondado)
  - A variável debito recebeu um valor muito grande (o resultado obtido está errado)

Conversões em Expressões

- O que acontece se tipos diferentes forem misturados em uma expressão aritmética?
  - Alguns tipos são promovidos sempre que são usados em expressões

```
char a = 90;
char b = 70;
int val = a + b;   // char é promovido para int
```

  - Alguns tipos são convertidos quando combinados com outros tipos

```
float total = 2.50 * val;  // val é convertido para double
```

Conversões em Expressões

- Conversão automática em expressões
  - Os tipos char e short são sempre promovidos para int (isso inclui as versões signed e unsigned)

```
char ch1 = '%';          // código ASCII % = 37
char ch2 = '&';          // código ASCII & = 38
short galinhas = 20;
short patos = 35;

// char's para int e o resultado int para char
char ch = ch1 + ch2;     // código ASCII K = 75

// short's para int e o resultado int para short
short aves = galinhas + patos;
```

Conversões em Expressões

- Conversão automática em expressões
  - Os tipos char e short são sempre promovidos para int (isso inclui as versões signed e unsigned)

```
unsigned char estado = 1;    // 00000001

// exibe int
cout << ~estado << endl;     // 11111111111111111111111111111110 = -2

// exibe unsigned char
estado = ~estado;
cout << estado << endl;      // 11111110 = 254
```

Conversões em Expressões

- Quando uma operação envolve dois tipos, o menor é convertido para o maior

```
// o valor 5 é convertido para double
float total = 9.0 / 5;

// o valor 5 é convertido para long long
long long val = 163481190409292 + 5;
```

- Quando os tipos são iguais não ocorre conversão
  - Mas é preciso tomar cuidado:

```
// o resultado não cabe em um int
long long erro = 100000000 * 2009;   // -963462912
```

Conversões em Expressões

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

int main()
{
     long long a, b, c, d;

     a = 200903280945;
     b = 100000000 * 2009;
     c = 100000000LL * 2009;
     d = 10000000000 * 2009;

     cout << "a: " << a << endl;
     cout << "b: " << b << endl;
     cout << "c: " << c << endl;
     cout << "d: " << d << endl;
}
```

ConvExpr.cpp

Conversões em Expressões

- A saída do programa:

```
a: 200903280945
b: -963462912
c: 200900000000
d: 2009000000000
```

  - Em b, os operandos são inteiros mas o resultado da operação é muito grande para um inteiro
  - Em d, um dos operandos é muito grande para um inteiro, sendo armazenado em long long e fazendo o resultado ser long long

Conversões em Funções

- Os protótipos das funções controlam as conversões nas passagens de argumento

```
float soma(float, float);
```

  - Aplicam-se as mesmas regras usadas na atribuição

```
// int → float
float a = soma(3,4);

// double → float nos argumentos e float → int no retorno
int b = soma(3.0, 4.0);

// nenhuma conversão
float c = soma(3.0f, 4.0f);
```

Type Casts

- A linguagem permite ao programador forçar conversões

```
float parcial = 5.4;
int resultado = int (3.8) + int (parcial);

int total = int (parcial); // estilo C++
int total = (int) parcial; // estilo C

cout << int ('A');
cout << char (65);

long long grande = long long (100000000) * 2009;
```

Declarações auto

- **C++11** introduziu a possibilidade de deduzir o tipo a partir do valor de inicialização

```
auto n = 100;       // n é int
auto x = 1.5f;      // x é float
auto y = 1.3e5;     // y é double
```

- Entretanto, esta dedução automática foi criada para casos mais complexos:

```
vector<pair<double,double>> vet;
vector<pair<double,double>>::iterator i = vet.begin();
auto i = vet.begin();
```

Resumo

- C++ dispõem de cinco operadores aritméticos:
  - Soma
  - Subtração
  - Multiplicação
  - Divisão (Inteira e Ponto Flutuante)
  - Módulo
- Conversões são feitas automaticamente mas também podem ser forçadas via type cast