Tipos Primitivos de Dados

Prof. Edson Alves

2020

Faculdade UnB Gama

Sumário

Inteiros

Variáveis integrais

- Em C/C++ há 3 tipos de dados integrais: **`char`**, **`short`**, **`int`**
- Embora o tipo **`char`** seja utilizado para representar um caractere, efetivamente ele é capaz de armazenar, em geral, um inteiro de 8 *bits*
- Os tamanhos característicos dos tipos **`short`** e **`int`** são de 16 e 32 *bits*, respectivamente
- A extensão **`long long`**, usada para representar inteiros de 64 *bits*, foi incorporado ao C++ a partir do padrão C++11
- Os modificadores **`signed`** e **`unsigned`** modificam o intervalo de valores representáveis pelos tipos integrais

Faixa de valores das variáveis integrais

Tipo	**Tamanho** (em *bits*)	**Intervalo**
`char`	8	$[-128, 127]$
`unsigned char`	8	$[0, 255]$
`short`	16	$[-32768, 32767]$
`unsigned short`	16	$[0, 65535]$
`int`	32	$\approx [-2 \times 10^9, 2 \times 10^9]$
`unsigned int`	32	$\approx [0, 4 \times 10^9]$
`long long`	64	$\approx [-9 \times 10^{18}, 9 \times 10^{18}]$
`unsigned long long`	64	$\approx [0, 10^{19}]$

Observações sobre as variáveis integrais

- Nas variáveis sinalizadas, valores negativos tem o *bit* mais significativo ligado
- Nas variáveis integrais existem divisores de zero, isto é, valores diferentes de zero cujo produto é igual a zero:

```
1 int x = 1024, y = 4194304, z = x * y;
2 bool zero = (z == 0);                    // Verdadeiro
```

- Isto acontece por conta do *overflow*: o resultado não cabe na faixa de valores representáveis, de modo que o "excesso" é descartado
- A multiplicação de dois inteiros, em geral, resulta em *overflow*: nestes casos, o ideal é converter ambos operando para **long long** e guardar o resultado uma variável também do tipo **long long**:

```
1 int x = 1000000, y = x;
2 long long z = x * y;          // z = -727379968 ?
```

Aritmética estendida

- Para armazenar valores inteiros que excedem o limite de variáveis do tipo **`long long`**, é necessário o uso de aritmética estendida
- Até a versão C++17, o C++ não tem suporte nativo à aritmética estendida
- Uma alternativa é utilizar o Python 3:

```python
from math import factorial as f
f(25)   # 15511210043330985984000000 > 2 ** 64 - 1
```

- Outra alternativa é recorrer à classe `BigInteger` do Java:

```java
import java.math.BigInteger;
class BigInt {
    public static void main(String[] args) {
        BigInteger two = BigInteger.valueOf(2);
        BigInteger x = two.pow(80);    // 1208925819614629174706176
    }
}
```

Variáveis em ponto flutuante

Variáveis em ponto flutuante

- Em C e C++ há dois tipos de variável em ponto flutuante: **`float`** e **`double`**
- O tipo **`float`** representa valores em ponto flutuante com precisão simples (7 dígitos de precisão)
- O tipo **`double`** representa valores em ponto flutuante com precisão dupla (15 casas dígitos de precisão)
- O GCC tem suporte para o tipo **`long double`**, com 80 *bits* e precisão superior ao tipo **`double`**
- A precisão extra traz custos de memória e também de performance

Observações sobre variáveis em ponto flutuante

- Nem todo valor pode ser representado exatamente em variáveis em ponto flutuante:

```
1 float x = 123456789.0f;
2 cout.precision(7);
3 cout << fixed << x << endl;          // 123456792.0000000
```

- Multiplicações de valores muito pequenos por valores muito grandes geram erros de precisão

```
1 double x = 1e-50, y = 2e80, z = x * y;
2 // z = 200000000000000039769249677312.000000000000000
```

- Comparações entre valores flutuantes podem gerar resultados incorretos por conta de erros de precisão:

```
1 bool equals = (0.3f * 2 == 0.6);          // Falso
```

Observações sobre variáveis em ponto flutuante

- Se possível, o ideal é evitar o uso de variáveis do tipo ponto flutuante
- Em competições, algoritmos corretos podem receber o veredito `WA` por conta de erros de precisão
- No cálculo da distância entre dois pontos de coordenadas inteiras, o ideal é trabalhar com o quadrado da distância, evitando a extração da raiz quadrada
- No caso de unidades monetárias, melhor trabalhar com múltiplos de centavos, de modo que os valores passam a ser todos números inteiros
- O mesmo vale para unidades de tempo: utilize com a menor unidade de tempo inteira possível (dias, segundos, etc)

Referências

1. A Tutorial on Data Representation. Integers, Floating-point Numbers, and Characters, acesso em 07/03/2020.
2. CppReference. C++ reference, acesso em 07/03/2020.
3. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
4. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.
5. **SKIENA**, Steven S.; **REVILLA**, Miguel A. *Programming Challenges*, 2003.
6. Wikipédia. Floating-point arithmetic, acesso em 07/03/2020.