

C++ para Programação Competitiva - Parte 1

Introdução à Programação Competitiva



**INSTITUTO
FEDERAL**
Brasília

Prof. Daniel Saad Nogueira
Nunes

IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

- 1 Introdução
- 2 Tipos de Dados
- 3 Referências
- 4 Funções
- 5 Entrada e Saída
- 6 Operações bit a bit



Sumário

1 Introdução



Por que usar C++?

- Por que usar C++ para programação competitiva?



Por que usar C++?

Razões

- É rápido: os programas são mais rápidos que os equivalentes em Java ou Python, o que é uma vantagem para problemas com TL apertado.
- Controle da memória: não há coletor de lixo, o controle da memória é determinístico.
- Estruturas de dados e algoritmos: a STL provê uma série de facilidades, evitando a implementação de estruturas de dados e algoritmos conhecidos.
- Possibilita o uso de programação genérica (`templates`).
- Sintaxe similar a do C, diminuindo a curva de aprendizagem para quem já conhece a linguagem.



Disclaimer

- Não estudaremos o C++ por completo.
- É uma linguagem muito complexa.
- Não aprofundaremos em orientação a objetos.
- O objetivo é utilizar a linguagem com foco em programação competitiva.



Compilação

- Os códigos-fonte em C++ possuem a extensão `.cpp`.
- O C++ é uma linguagem compilada, sendo um dos seus compiladores o `g++`. Uma possível forma de compilar o programa `source.cpp` para gerar o executável `source` é a seguinte:
- `g++ source.cpp -Wall -o source`
- Caso exista interesse em habilitar suporte para os padrões mais novos da linguagem, como o C++ 14 ou o C++ 17, podemos utilizar a flag `-std=c++xx`, em que `xx` representa a versão da linguagem.
- `g++ source.cpp -std=c++17 -Wall -o source`



Sumário

2 Tipos de Dados



Tipos primitivos

- O C++ possui alguns tipos primitivos de dados, como: inteiros, caracteres, booleanos e números ponto-flutuante.
- Abordarem a seguir estes tipos primitivos.



Sumário

2 Tipos de Dados

- **Inteiros**
- Caracteres
- Booleanos
- Reais



Inteiros

- Inteiros em C++ podem ou não ter sinais.
- Também possuem representações de tamanhos diferentes.
- Escolher o tipo adequado para o problema é essencial.



Inteiros

Tipo	Tamanho Mínimo	Tamanho Típico	Intervalo de Representação (Típico)
short	2 bytes	2 bytes	-32768 a 32767
unsigned short	2 bytes	2 bytes	0 a 65,535
int	2 bytes	4 bytes	-2147483648 a 2147483647
unsigned int	2 bytes	4 bytes	0 a 4294967295
long int	4 bytes	8 bytes	-9223372036854775808 a 9223372036854775807
unsigned long int	4 bytes	8 bytes	0 a 18446744073709551615
long long int	8 bytes	8 bytes	-9223372036854775808 a 9223372036854775807
unsigned long long int	8 bytes	8 bytes	0 a 18446744073709551615



Inteiros

- Perceba que a linguagem não fixa o tamanho da representação de cada tipo, apenas estipula o tamanho mínimo.
- Caso você queria ter certeza que o inteiro escolhido tenha uma representação específica, podemos recorrer ao cabeçalho `cstdint`.



O cabeçalho `cstdint`

O cabeçalho `cstdint` permite definir tipos inteiros com tamanho da representação desejada:

- `int8_t`: inteiro de 8 bits com sinal
- `int16_t`: inteiro de 16 bits com sinal
- `int32_t`: inteiro de 32 bits com sinal
- `int64_t`: inteiro de 64 bits com sinal
- `uint8_t`: inteiro de 8 bits sem sinal
- `uint16_t`: inteiro de 16 bits sem sinal
- `uint32_t`: inteiro de 32 bits sem sinal
- `uint64_t`: inteiro de 64 bits com sinal



Bases

- O C++ suporta a atribuição de inteiros representados em diferentes bases.
- Utiliza-se o prefixo 0x para hexa e 0b para binário.
- `int x = 0x3f; // x = 63`
- `int x = 0b10010101; // x = 149`



Aspas Simples

- É possível utilizar aspas simples (C++ 14) para separar literais inteiros e melhorar a legibilidade.
- `int x = 1'000'000'000 ;`
- `int x = 0b1001'1101 ;`
- As aspas são completamente ignoradas pelo compilador.



Sumário

2 Tipos de Dados

- Inteiros
- Caracteres
- Booleanos
- Reais



Caracteres

- Caracteres em C++ são representados internamente da mesma forma que inteiros, mas utilizando apenas 1 byte (8 bits).
- Um valor de um caractere é um inteiro que corresponde ao índice de uma letra na tabela ASCII.
- Apenas um caractere pode ser armazenado em uma variável do tipo `char`.
- O tipo `char` admite o modificador `unsigned`.
- Para representar uma sequência de caracteres (uma palavra), podemos utilizar o tipo `string` (veremos mais tarde).



Caracteres

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]



Caracteres

Tipo	Tamanho	Intervalo de Representação
char	1 byte	-128 a 127
unsigned char	1 byte	0 a 255



Sumário

2 Tipos de Dados

- Inteiros
- Caracteres
- **Booleanos**
- Reais



Booleans

- Nativamente o C++ suporta o tipo `bool`, que pode assumir dois valores:
 - ▶ `true`, equivalente ao valor numérico 1.
 - ▶ `false`, equivalente ao valor numérico 0.



Sumário

2 Tipos de Dados

- Inteiros
- Caracteres
- Booleanos
- Reais



Reais

- Tipo: `float` ou `double`.
- Normalmente as arquiteturas modernas utilizam o padrão IEEE 754 para representação de números reais através de ponto flutuante.
- O tipo `double` admite o modificador `long`.



Reais

float

- 32-bits;
- A grosso modo, possui uma precisão de 6 casas decimais.
- Intervalo de representação está contido em : $[10^{-38}, 10^{38}]$.



Reais

double

- 64-bits;
- A grosso modo, possui uma precisão de 15 casas decimais.
- Intervalo de representação está contido em : $[10^{-308}, 10^{308}]$.



Reais

long double

- 80-bits (tipicamente), 96-bits ou 128-bits;
- Normalmente o intervalo de representação está contido em :
 $[10^{-4951}, 10^{4932}]$.



Sumário

3 Referências



Referências

- O C++ possui suporte às referências, que podem ser vistas como um apelido para outra variável.
- Elas devem ser inicializadas no momento da sua declaração, não podendo ter o seu valor atualizado.
- Utilizamos o símbolo `&` após o nome do tipo para indicarmos que se trata de uma referência para aquele tipo.



Referências

```
1  #include <iostream>
2
3  int main() {
4      int a = 5;
5      int &ref_a = a;
6      ref_a = 10;
7      std::cout << "a = " << a << " ref_a = " << ref_a << std::endl;
8      return 0;
9  }
```



Referências

- Referências são menos poderosas que ponteiros, já que não podem ser alteradas.
- Mas simplificam a escrita do código e evitam sintaxes desnecessárias.
- São usadas normalmente como variáveis do tipo referenciado.



Sumário

4 Funções



Funções em C++

- A linguagem C possibilita apenas a passagem por valor, isto é, uma cópia do valor da variável passada por parâmetro é realizada.
- Em C++ existem dois métodos de passagem de parâmetros: por valor e por **referência**.



Passagem por referência

- Na passagem por referência, uma referência à variável ou objeto é utilizada, fazendo com que qualquer alteração reflita no original.
- Além disto, como uma referência é utilizada, e não é realizada uma cópia, pode-se obter algum ganho de desempenho quando objetos grandes são passados para funções (exemplo, objetos do tipo `vector`).
- Para utilizar a passagem por referência, basta utilizar o `&` após o tipo do parâmetro. Caso o `&` não seja utilizado, é realizada uma passagem por valor (cópia).
- É claro que a escolha do método de passagem de parâmetros vai depender da função e aplicação.



Passagem por referência

```
1 void erastosthenes(std::vector<bool> &sieve, int n) {
2     sieve.assign(n + 1, true);
3     sieve[0] = sieve[1] = false;
4     for (int i = 2; i * i <= n; i++) {
5         if (sieve[i]) {
6             for (int j = i * 2; j <= n; j += i) {
7                 sieve[j] = false;
8             }
9         }
10    }
11 }
```



Passagem por referência

Quando usar passagem por referência?

- Para modificar a variável original, passada para a função.
- Para passar variáveis grandes: realizar uma cópia é muito mais custoso do que utilizar uma referência.
- Para permitir polimorfismo: se a função recebe uma referência da classe base, ao invocá-la com um objeto da classe derivada, o comportamento será o da classe derivada.



Sumário

5 Entrada e Saída



Entrada e Saída

- O C++ possui suporte às funções `scanf` e `printf` legadas do C, pertencentes ao cabeçalho `cstdio`.
- Contudo, também existe a possibilidade de utilizar os mecanismos de entrada e saída próprios do C++. Estes mecanismos estão descritos no cabeçalho `iostream`.
- Através dos objetos `cin` e `cout` podemos ler dados da entrada padrão e imprimir dados na saída padrão.



Sumário

- 5 Entrada e Saída
 - cin e cout
 - Fast I/O



cin

- `cin` é o chamado *stream* padrão para leitura dos dados. Ele possibilita ler dados da entrada padrão e armazená-los em variáveis.
- Através do operador `>>` é possível realizar diversas leituras com uma única linha de código.



cin

```
1  #include <iostream>
2
3  int main() {
4      int a;
5      float b;
6      double c;
7      std::cin >> a >> b >> c;
8      return 0;
9  }
```



cout

- `cout` é o chamado *stream* padrão para escrita dos dados. Ele possibilita imprimir o conteúdo das variáveis na saída padrão.
- Através do operador `<<` é possível realizar diversas escritas com uma única linha de código.



cout

```
1  #include <iostream>
2
3  int main() {
4      int a = 5;
5      float b = 3.14;
6      char c = 'd';
7      std::cout << a << ' ' << b << ' ' << c << '\n';
8      return 0;
9  }
```



Sumário

- 5 Entrada e Saída
 - cin e cout
 - Fast I/O



Fast I/O

- Como dito anteriormente, o C++ suporta tanto as funções do C como os *streams* `cin` e `cout`.
- Inclusive é possível utilizar ambos os mecanismos simultaneamente.
- Notavelmente, o desempenho dos mecanismos do C++ são mais lentos que os do C, o que pode ocasionar um TLE.
- Ao desabilitar a sincronia entre os dois mecanismos, `cin` e `cout` tornam-se bem mais eficientes.
- Contudo, ao desabilitar a sincronia, não será mais possível misturar métodos de I/O das duas linguagens.
- Para desabilitar a sincronia, basta adicionar a linha `std::ios::sync_with_stdio(false);` no início do programa.



Fast I/O

```
1  #include <iostream>
2
3  int main() {
4      std::ios::sync_with_stdio(false);
5      /**
6       * Programa ...
7       */
8      return 0;
9  }
```



Sumário

6 Operações bit a bit



Operações bit a bit

- As operações bit a bit operam sobre inteiros.
- Suponha que x , y e i são inteiros. Os possíveis operadores são:
 - ▶ `~x` : obtém a representação binária complementar de x .
 - ▶ `x & y` : realiza a operação de **E** bit a bit entre x e y
 - ▶ `x | y` : realiza a operação de **OU** bit a bit entre x e y .
 - ▶ `x ^ y` : realiza a operação de **XOR** bit a bit entre x e y .
 - ▶ `x << i` : realiza a operação de deslocamento à esquerda (shift left) de i posições.
 - ▶ `x >> i` : realiza a operação de deslocamento à direita (shift right) de i posições.



~: complemento de um

- O operador `~` realiza o complemento de um da representação binária de um inteiro.
- `uint8_t x = 170; // x = 0b1010'1010`
- `uint8_t y = ~x; // y = 0b0101'0101 (85)`



~: complemento de um

```
1  #include <stdint>
2  #include <stdio>
3
4  int main() {
5      uint8_t x = 170;
6      printf("%hhu %hhu\n", x, ~x);
7      return 0;
8  }
```



~: complemento de um

- O operador `~` realiza o complemento de um da representação binária de um inteiro.
- `int8_t x = -86; // x = 0b1010'1010`
- `int8_t y = ~x; // y = 0b0101'0101 (85)`



~: complemento de um

```
1  #include <stdint>
2  #include <stdio>
3
4  int main() {
5      int8_t x = -86;
6      printf("%hhd %hhd\n", x, ~x);
7      return 0;
8  }
```



Complemento de dois

- Os números negativos são representados via complemento de dois.
- O complemento de dois é obtido a partir do complemento de um somado de um.
- Propriedade:** `-x == ~x+1`



&: E bit a bit

- Seja o inteiro x composto pelos bits $x_0x_1 \dots x_{n-1}$ e seja y composto pelos bits $y_0y_1 \dots y_{n-1}$.
- O operador $\&$ aplica o operador de **E** bit a bit entre x_i e y_i para cada $0 \leq i < n$: isto é

$$\begin{array}{cccc}
 & x_0 & x_1 & \dots & x_{n-1} \\
 \& & y_0 & y_1 & \dots & y_{n-1} \\
 \hline
 & x_0 \& y_0 & x_1 \& y_1 & \dots & x_{n-1} \& y_{n-1}
 \end{array}$$

a	b	a&b
0	0	0
0	1	0
1	0	0
1	1	1



&: E bit a bit

```
1  #include <stdint>
2  #include <stdio>
3
4  int main() {
5      uint8_t x = 0b1010'1010; // x = 170
6      uint8_t y = 0b1100'1100; // y = 204
7      uint8_t z = x & y;      // z = 0b1000'1000 (136)
8      printf("%hhu\n", z);
9      return 0;
10 }
```



|: **OU** bit a bit

- Seja o inteiro x composto pelos bits $x_0x_1 \dots x_{n-1}$ e seja y composto pelos bits $y_0y_1 \dots y_{n-1}$.
- O operador $|$ aplica o operador de **OU** bit a bit entre x_i e y_i para cada $0 \leq i < n$: isto é

$$\begin{array}{cccc}
 & x_0 & x_1 & \dots & x_{n-1} \\
 | & y_0 & y_1 & \dots & y_{n-1} \\
 \hline
 & x_0|y_0 & x_1|y_1 & \dots & x_{n-1}|y_{n-1}
 \end{array}$$

a	b	a b
0	0	0
0	1	1
1	0	1
1	1	1



|: **OU** bit a bit

```
1  #include <stdint>
2  #include <stdio>
3
4  int main() {
5      uint8_t x = 0b1010'1010; // x = 170
6      uint8_t y = 0b1100'1100; // y = 204
7      uint8_t z = x | y;      // z = 0b1110'1110 (238)
8      printf("%hhu\n", z);
9      return 0;
10 }
```



^ XOR bit a bit

- Seja o inteiro x composto pelos bits $x_0x_1 \dots x_{n-1}$ e seja y composto pelos bits $y_0y_1 \dots y_{n-1}$.
- O operador \wedge aplica o operador de **XOR** bit a bit (ou exclusivo) entre x_i e y_i para cada $0 \leq i < n$: isto é

$$\begin{array}{cccc}
 & x_0 & x_1 & \dots & x_{n-1} \\
 \wedge & y_0 & y_1 & \dots & y_{n-1} \\
 \hline
 & x_0 \wedge y_0 & x_1 \wedge y_1 & \dots & x_{n-1} \wedge y_{n-1}
 \end{array}$$

a	b	a \wedge b
0	0	0
0	1	1
1	0	1
1	1	0



\wedge : **XOR** bit a bit

```
1  #include <stdint>
2  #include <stdio>
3
4  int main() {
5      uint8_t x = 0b1010'1010; // x = 170
6      uint8_t y = 0b1100'1100; // y = 204
7      uint8_t z = x ^ y;      // z = 0b0110'0110 (102)
8      printf("%hhu\n", z);
9      return 0;
10 }
```



«: shift-left

- Seja o inteiro x composto pelos bits $x_0x_1 \dots x_{n-1}$ e i um inteiro.
- O operador « desloca todos os bits de x , i posições para a esquerda.
- A cada deslocamento, temos que

$$x_0 \leftarrow x_1, x_1 \leftarrow x_2, \dots, x_{n-2} \leftarrow x_{n-1} \text{ e } x_{n-1} \leftarrow 0.$$

```
1  uint8_t x = 0b1001'1001; // x = 153
2  uint8_t y = x << 1;      // y = 0b0011'0010 (50)
3  uint8_t z = x << 2;      // z = 0b011'00100 (100)
4  uint8_t w = x << 3;      // w = 0b1100'1000 (200)
```



≫: shift-right

- Seja o inteiro x composto pelos bits $x_0x_1 \dots x_{n-1}$ e i um inteiro.
- O operador \gg desloca todos os bits de x , i posições para a esquerda.
- A cada deslocamento, temos que

$$x_{n-1} \leftarrow x_{n-2}, x_{n-2} \leftarrow x_{n-3}, \dots, x_1 \leftarrow x_0 \text{ e } x_0 \leftarrow 0.$$

```
1  uint8_t x = 0b1001'1001; // x = 153
2  uint8_t y = x >> 1;      // y = 0b0100'1100 (76)
3  uint8_t z = x >> 2;      // z = 0b0010'0110 (38)
4  uint8_t w = x >> 3;      // w = 0b0001'0011 (19)
```



Ligar o i -ésimo bit

- Para ligar o i -ésimo bit **menos significativo** de um inteiro x , basta fazer:

```
x = x | (1 << i)
```

- Que abreviadamente é:

```
x |= (1 << i)
```

- Isto funciona pois $1 \ll i$ é exatamente o número em que todos os bits são zeros, exceto aquele que ocupa a posição i . Ex:

```
1 << 5 == 0b0000...100000
```

Assim, ao aplicar o operador $|$, apenas o i -ésimo bit menos significativo é ligado.



Ligar o i -ésimo bit

```
1  uint8_t x = 0b1001'1001; // x = 153
2  uint8_t y = x | (1 << 1); // y = 0b1001'1011
3  uint8_t z = x | (1 << 7); // z = 0b1001'1001
4  uint8_t w = y | (1 << 5); // w = 0b1011'1011
```



Desligar o i -ésimo bit

- Para desligar o i -ésimo bit **menos significativo** de um inteiro x , basta fazer:

```
x = x & ~(1 << i)
```

- Que abreviadamente é:

```
x &= ~(1 << i)
```

- Isto funciona pois $1 << i$ é exatamente o número em que todos os bits são zeros, exceto aquele que ocupa a posição i . Ex:

```
~(1 << 5) == 0b1111...011111
```

Assim, ao aplicar o operador $\&$, apenas o i -ésimo bit menos significativo é desligado.



Desligar o i -ésimo bit

```
1  uint8_t x = 0b1001'1001; // x = 153
2  uint8_t y = x & ~(1 << 1); // y = 0b1001'1001
3  uint8_t z = x & ~(1 << 7); // z = 0b0001'1001
4  uint8_t w = z & ~(1 << 4); // w = 0b0000'1001
```



Flipar o i -ésimo bit

- Para inverter (flipar) o i -ésimo bit **menos significativo** de um inteiro x , basta fazer:

```
x = x ^ (1 << i)
```

- Que abreviadamente é:

```
x ^= (1 << i)
```

- Isto funciona pois `1<<i` é exatamente o número em que todos os bits são zeros, exceto aquele que ocupa a posição i . Ex:

```
1<<5 == 0b0000...100000
```

Assim, ao aplicar o operador \wedge , apenas o i -ésimo bit menos significativo é invertido.



Flipar o i -ésimo bit

```
1  uint8_t x = 0b1001'1001; // x = 153
2  uint8_t y = x ^ (1 << 1); // y = 0b1001'1011
3  uint8_t z = x ^ (1 << 7); // z = 0b0001'1001
4  uint8_t w = z ^ (1 << 4); // w = 0b0000'1001
```



Obter a potência 2^n

- Toda a potência de 2 em binário é da forma $00 \dots 10 \dots 0$.
- Para ser mais preciso, temos que 2^i possui todos os bits desligados, com exceção do i -ésimo bit menos significativo. Ex: $2^0 = 1$, $2^4 = 10000$.
- Assim, para obter a potência 2^i , basta fazer `1 << i`.



Multiplicar um número por 2^n

- Seguindo a mesma lógica anterior, para multiplicar um número x por 2^n , basta fazer: `x <<= n`.
- Exemplo: $5 \cdot 32 = 5 \cdot 2^5$, que é `5 << 5 // 0b10100000 (160)`.
- Em outras palavras, deslocar um inteiro x em n posições para a esquerda, equivale à multiplicá-lo por 2^n .



Dividir um número por 2^n

- A lógica da divisão por 2^n é simétrica a da multiplicação por 2^n . Para dividir um número x por 2^n , basta fazer: `x >>= n`.
- Exemplo: $160/32 = 160/2^5$, que é `160 >> 5 // 0b101 (5)`.
- Em outras palavras, deslocar um inteiro x em n posições para a direita, equivale à dividi-lo por 2^n .



Desligar o bit 1 menos significativo

- Como desligar o bit 1 menos significativo de um número?
- Exemplo: se $x = 010010100$, então a resposta seria $x' = 010010000$.
- Basta fazer `x = x & (x-1)`.
- Ou, abreviadamente: `x &= x-1`
- Por que isso funciona?



Desligar o bit 1 menos significativo

- Seja $x = x_{n-1} \dots x_0$ um inteiro e seja i a posição 1 menos significativo.
- Isto é, x é da forma $x_{n-1} \dots \underbrace{x_i 0 \dots 0}_{i+1}$.
- $x - 1$ será da forma: $x_{n-1} \dots \underbrace{01 \dots 1}_{i+1}$.
- Fazendo com que `x & (x-1)` dê como resultado:
 $x_{n-1} \dots x_{i+1} \underbrace{0 \dots 0}_{i+1}$, isto é, x com o bit da posição i desligado.



Verificar se um número é potência de dois

- Uma potência de 2 tem sua representação binária com um único bit ligado.
- Assim, para saber se um número é potência de dois, basta verificar se o bit 1 menos significativo for desligado o resultado seja zero:
`x & (x-1) == 0`.
- Contudo, com esta condição o zero seria considerado uma potência de 2.
- Para contornar, basta testar se x é diferente de zero.
- Expressão resultante: `x && (x & (x-1))`.



Verificar se um número é potência de dois

```
1 bool is_power_of_two(int x){  
2     return x && (x & (x-1));  
3 }
```



Restos de potência de dois

- Para obter o resto, pode-se usar o operador %.
- Contudo, quando o divisor é potência de dois, podemos utilizar operações bit a bit, muito mais baratas que uma divisão.
- Se x é uma potência de dois, o resto r da divisão de y por x pode ser calculado como `r = y & (x-1)`.
- Razão: o resto de uma divisão por 2^i sempre corresponde aos i bits menos significativos do número.
- Os i bits menos significativos nos dão a faixa de inteiros $[0, 2^i - 1]$, os possíveis valores de resto por x .



Contar o número de bits ligados

- Uma forma eficiente de contar quantos bits 1 estão ligados é desligar o bit 1 menos significativo até que o número valha zero:

```
1 int count_1(int x) {  
2     int c;  
3     for (c = 0; x; c++) {  
4         x &= x - 1;  
5     }  
6     return c;  
7 }
```

- O laço executa tantas vezes quanto existem bits 1.



Verificar se dois inteiros possuem o mesmo sinal

- Como os inteiros em C++ são representados via complemento de dois, temos que:
 - ▶ Um número positivo possui 0 no bit mais significativo.
 - ▶ Um número negativo possui 1 no bit menos significativo.
- Ao aplicar um XOR bit a bit entre um número negativo e um número positivo, o bit mais significativo do resultado estará ligado, isto é, o resultado será negativo.



Verificar se dois inteiros possuem o mesmo sinal

```
1 bool diff_sign(int x,int y){  
2     return (x ^ y) < 0;  
3 }
```



Logaritmo inteiro rápido

- Dado um inteiro x , como computar $\lfloor \lg x \rfloor$ rapidamente?
- Podemos usar uma tabela pré-computada para contar o número de bits suficientes necessários para representar x .



Logaritmo inteiro rápido

```
1  #define LT(n) n, n, n, n, n, n, n, n, n, n, n, n, n, n, n, n
2  const char log_table_256[256] = {
3      -1,    0,    1,    1,    2,    2,
4      2,    2,    3,    3,    3,    3,
5      3,    3,    3,    3,    LT(4), LT(5),
6      LT(5), LT(6), LT(6), LT(6), LT(6), LT(7),
7      LT(7), LT(7), LT(7), LT(7), LT(7), LT(7), LT(7)
8  };
```




Logaritmo inteiro rápido

```
1  int fast_log2(int x) {
2      int r,tt;
3      if (tt = x >> 24) {
4          r = 24 + log_table_256[tt];
5      } else if (tt = v >> 16) {
6          r = 16 + log_table_256[tt];
7      } else if (tt = v >> 8) {
8          r = 8 + log_table_256[tt];
9      } else {
10         r = log_table_256[x];
11     }
12     return r;
13 }
```



Mais operações bit a bit

<https://graphics.stanford.edu/~seander/bithacks.html>