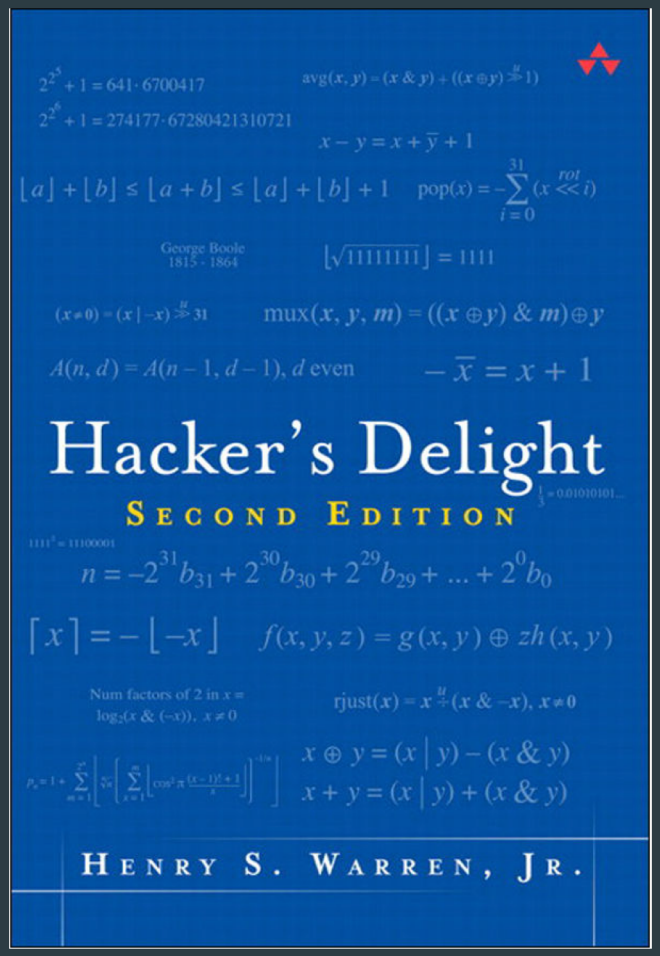
Estrutura de dados 2

Aula 29: *Fun with Bits*

Operadores de bits

- C oferece várias ferramentas para se manipular bits.
- O acesso a bits é importante em várias situações:
  - Software com alto desempenho.
  - Economizar memória com valores *booleanos*.
  - Desenvolvimento de SOs.
  - Software para redes.
  - etc.

Leitura recomendada



- https://www.topcoder.com/community/competitive-programming/tutorials/a-bit-of-fun-fun-with-bits/

Ferramentas básicas

Operador	Nome	Descrição
&	“E” sobre bits	Os bits resultantes valem 1 se os bits correspondentes valem 1 e 0 caso contrário.
|	“Ou” sobre bits	Os bits resultantes valem 1 se pelo menos um dos bits correspondentes valem 1 e 0 caso contrário.
^	“Ou exclusivo” sobre bits.	Os bits resultantes valem 1 se exatamente um dos bits correspondentes valem 1 e 0 caso contrário.
<<	Deslocamento de bits para a esquerda.	Desloca os bits do primeiro operando para a esquerda pelo número de bits especificados no segundo operando. Preenche a partir da direita com bits 0.
>>	Deslocamento de bits para a direita.	Desloca os bits do primeiro operando para a direita pelo número de bits especificados no segundo operando. O preenchimento à esquerda depende da máquina.
~	Complemento de bits.	Os bits 0 são transformados em 1 e os bits 1 são transformados em 0.

```c
int main() {
    unsigned int a = 3;
    unsigned int b = 5;

    imprimeBits(b);
    imprimeBits(a);
    imprimeBits(~a);
    imprimeBits(a | b);
    imprimeBits(a & b);
    imprimeBits(a ^ b);
    imprimeBits(a << 1);
    imprimeBits(a << 2);
    imprimeBits(a >> 1);
    imprimeBits(a >> 2);

    return 0;
}
```

```
00000000 00000000 00000000 00000101
00000000 00000000 00000000 00000011
11111111 11111111 11111111 11111100
00000000 00000000 00000000 00000111
00000000 00000000 00000000 00000001
00000000 00000000 00000000 00000110
00000000 00000000 00000000 00000110
00000000 00000000 00000000 00001100
00000000 00000000 00000000 00000001
00000000 00000000 00000000 00000000
```

Trocar o valor de dois inteiros sem variável auxiliar

Trocar o valor de dois inteiros sem variável auxiliar

```c
int main() {
  int a = 30945867;
  int b = 7872345;
  printf("a = %d\nb = %d\n", a, b);
  imprimeBits(a);
  imprimeBits(b);
  a = a ^ b;
  imprimeBits(a);
  b = b ^ a;
  imprimeBits(b);
  a = a ^ b;
  imprimeBits(a);
  printf("a = %d\nb = %d\n", a, b);
  return 0;
}
```

```
a = 30945867
b = 7872345
00000001 11011000 00110010 01001011
00000000 01111000 00011111 01011001
00000001 10100000 00101101 00010010
00000001 11011000 00110010 01001011
00000000 01111000 00011111 01011001
a = 7872345
b = 30945867
```

Dado um número inteiro *x*, como podemos saber se o *i*-ésimo bit de *x* vale 1?

Dado um número inteiro *x*, como podemos saber se o *i*-ésimo bit de *x* vale 1?

Bit 2 de 5 é 1?

```
  00000000 00000000 00000000 00000101

----------------------------------------
  00000000 00000000 00000000 00000100
```

Bit 7 de 5 é 1?

```
  00000000 00000000 00000000 00000101

----------------------------------------
  00000000 00000000 00000000 00000000
```

Dado um número inteiro *x*, como podemos saber se o *i*-ésimo bit de *x* vale 1?

Bit 2 de 5 é 1?

```
  00000000 00000000 00000000 00000101
& 00000000 00000000 00000000 00000100
-----------------------------------
  00000000 00000000 00000000 00000100
```

Bit 7 de 5 é 1?

```
  00000000 00000000 00000000 00000101
& 00000000 00000000 00000000 10000000
-----------------------------------
  00000000 00000000 00000000 00000000
```

Dado um número inteiro *x*, como podemos saber se o *i*-ésimo bit de *x* vale 1?

Bit 2 de 5 é 1?

```
  00000000 00000000 00000000 00000101
& 00000000 00000000 00000000 00000100
-----------------------------------
  00000000 00000000 00000000 00000100
```

Bit 7 de 5 é 1?

```
  00000000 00000000 00000000 00000101
& 00000000 00000000 00000000 10000000
-----------------------------------
  00000000 00000000 00000000 00000000
```

```
if (x & (1 << i)) {
    // code if yes
} else {
    // code if no
}
```

Como podemos "ligar" o *i*-ésimo bit de um número *x*?

Bit 2 de 5, passar a ser 1

```
  00000000 00000000 00000000 00000101

--------------------------------------
  00000000 00000000 00000000 00000101
```

Bit 7 de 5, passar a ser 1

```
  00000000 00000000 00000000 00000101

--------------------------------------
  00000000 00000000 00000000 10000101
```

Como podemos "ligar" o *i*-ésimo bit de um número *x*?

Bit 2 de 5, passar a ser 1

```
  00000000 00000000 00000000 00000101
| 00000000 00000000 00000000 00000100
-----------------------------------
  00000000 00000000 00000000 00000101
```

Bit 7 de 5, passar a ser 1

```
  00000000 00000000 00000000 00000101
| 00000000 00000000 00000000 10000000
-----------------------------------
  00000000 00000000 00000000 10000101
```

Como podemos "ligar" o *i*-ésimo bit de um número *x*?

Bit 2 de 5, passar a ser 1

```
  00000000 00000000 00000000 00000101
| 00000000 00000000 00000000 00000100
-----------------------------------
  00000000 00000000 00000000 00000101
```

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

Bit 7 de 5, passar a ser 1

```
  00000000 00000000 00000000 00000101
| 00000000 00000000 00000000 10000000
-----------------------------------
  00000000 00000000 00000000 10000101
```

Como podemos “desligar” o *i*-ésimo bit de um número *x*?

Desligar bit 2 de 5

```
  00000000 00000000 00000000 00000101

----------------------------------------
  00000000 00000000 00000000 00000001
```

Desligar bit 7 de 5

```
  00000000 00000000 00000000 00000101

----------------------------------------
  00000000 00000000 00000000 00000101
```

Como podemos “desligar” o *i*-ésimo bit de um número *x*?

Desligar bit 2 de 5

```
  00000000 00000000 00000000 00000101
& 11111111 11111111 11111111 11111011
-----------------------------------
  00000000 00000000 00000000 00000001
```

Desligar bit 7 de 5

```
  00000000 00000000 00000000 00000101
& 11111111 11111111 11111111 01111111
-----------------------------------
  00000000 00000000 00000000 00000101
```

Como podemos “desligar” o *i*-ésimo bit de um número *x*?

Desligar bit 2 de 5

```
  00000000 00000000 00000000 00000101
& 11111111 11111111 11111111 11111011
-----------------------------------
  00000000 00000000 00000000 00000001
```

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

Desligar bit 7 de 5

```
  00000000 00000000 00000000 00000101
& 11111111 11111111 11111111 01111111
-----------------------------------
  00000000 00000000 00000000 00000101
```

Como podemos “imprimir” os bits de um número?

Como podemos “imprimir” os bits de um número?

Basta verificar se cada bit é 1 ou 0 e imprimir de acordo

Como podemos “imprimir” os bits de um número?

Basta verificar se cada bit é 1 ou 0 e imprimir de acordo

```
void imprimeBits(unsigned int n) {
	for(int i = 31; i >= 0; i--) {
		if ((n & (1 << i)) != 0)
			printf("1");
		else
			printf("0");
	}
	printf("\n");
}
```

Como podemos “desligar” o bit 1 mais à direita de um número *x*?

Desligar bit mais a direita de 5

```
  00000000 00000000 00000000 00000101

---------------------------------------
  00000000 00000000 00000000 00000100
```

Desligar bit mais a direita de 28?

```
  00000000 00000000 00000000 00011100

---------------------------------------
  00000000 00000000 00000000 00011000
```

Como podemos “desligar” o bit 1 mais à direita de um número *x*?

Desligar bit mais a direita de 5

```
  00000000 00000000 00000000 00000101
& 00000000 00000000 00000000 00000100
-----------------------------------
  00000000 00000000 00000000 00000100
```

Desligar bit mais a direita de 28?

```
  00000000 00000000 00000000 00011100
& 00000000 00000000 00000000 00011011
-----------------------------------
  00000000 00000000 00000000 00011000
```

Como podemos “desligar” o bit 1 mais à direita de um número *x*?

Desligar bit mais a direita de 5

```
  00000000 00000000 00000000 00000101
& 00000000 00000000 00000000 00000100
-----------------------------------
  00000000 00000000 00000000 00000100
```

```
x = x & (x - 1);
```

Desligar bit mais a direita de 28?

```
  00000000 00000000 00000000 00011100
& 00000000 00000000 00000000 00011011
-----------------------------------
  00000000 00000000 00000000 00011000
```

Como podemos descobrir se um número positivo é da forma 2^n?

5 é da forma 2^n?

```
  00000000 00000000 00000000 00000101

----------------------------------------
  00000000 00000000 00000000 00000100
```

128 é da forma 2^n?

```
  00000000 00000000 00000000 10000000

----------------------------------------
  00000000 00000000 00000000 00000000
```

Como podemos descobrir se um número positivo é da forma 2^n?

5 é da forma 2^n?

```
  00000000 00000000 00000000 00000101
& 00000000 00000000 00000000 00000100
-----------------------------------
  00000000 00000000 00000000 00000100
```

128 é da forma 2^n?

```
  00000000 00000000 00000000 10000000
& 00000000 00000000 00000000 01111111
-----------------------------------
  00000000 00000000 00000000 00000000
```

Como podemos descobrir se um número positivo é da forma 2^n?

5 é da forma 2^n?

```
  00000000 00000000 00000000 00000101
& 00000000 00000000 00000000 00000100
-----------------------------------
  00000000 00000000 00000000 00000100
```

128 é da forma 2^n?

```
  00000000 00000000 00000000 10000000
& 00000000 00000000 00000000 01111111
-----------------------------------
  00000000 00000000 00000000 00000000
```

Desligue o bit 1 mais à direita e verifique se o resultado é zero!

```
if ((x & (x - 1)) == 0)
    printf("Power of two!");
```

Errado:

```
if (x & (x - 1) == 0)
    printf("Power of two!");
```

Mais dois...

- Como podemos descobrir se um número é da forma 2^n-1?
- Como podemos isolar o bit 1 mais à direita de um número x ?

Mais dois...

- Como podemos descobrir se um número é da forma 2^n-1?
  - Basta fazer o teste: `((x & (x + 1)) == 0)`
    - `x = 000111`
    - `x+1 = 001000`
    - `x & (x + 1) = 000000`
- Como podemos isolar o bit 1 mais à direita de um número *x* ?
  - Basta fazer: `x & ~(x-1)`
    - `x = 001010`
    - `x-1 = 001001`
    - `x & ~(x-1) = 000010`

Complemento de dois

```
int main() {
  int a = 1;
  int b = -a;

  imprimeBits(a);
  imprimeBits(b);
  imprimeBits(~a + 1);
  imprimeBits(~b + 1);
  return 0;
}
```

```
00000000 00000000 00000000 00000001
11111111 11111111 11111111 11111111
11111111 11111111 11111111 11111111
00000000 00000000 00000000 00000001
```

Valor absoluto

Valor absoluto

```
unsigned int absvalue(int n) {
        return (n & (1 << 31))? ~n + 1: n;
}
```

Cuidado com a interpretação...

```
printf("%d\n", 1 << 0);
printf("%d\n", 1 << 1);
printf("%d\n", 1 << 2);
// ...
printf("%d\n", 1 << 10);
printf("%d\n", 1 << 11);
// ...
printf("%d\n", 1 << 29);
printf("%d\n", 1 << 30);
printf("%d\n", 1 << 31);
```

```
1
2
4

1024
2048

536870912
1073741824
-2147483648
```

Sobre números negativos...

- The left shift and right shift operators should not be used for negative numbers
- If the number is shifted more than the size of integer, the behavior is undefined

- https://www.geeksforgeeks.org/left-shift-right-shift-operators-c-cpp/

Conteúdo de um byte

- Dado um inteiro (4 bytes) como pegar
  - O primeiro byte?
  - O segundo byte?
  - O terceiro byte?
  - O quarto byte?

Conteúdo de um byte

```
int main() {
  int a = 47523567; // exemplo
  imprimeBits(a);   // bits de a
  imprimeBits(255); // máscara

  imprimeBits(a & 255);          // primeiro
  imprimeBits((a >> 8) & 255);   // segundo
  imprimeBits((a >> 16) & 255); // terceiro
  imprimeBits((a >> 24) & 255); // quarto
  return 0;
}
```

```
00000010 11010101 00100110 11101111
00000000 00000000 00000000 11111111
00000000 00000000 00000000 11101111
00000000 00000000 00000000 00100110
00000000 00000000 00000000 11010101
00000000 00000000 00000000 00000010
```

Representação de conjunto

- Podemos utilizar número inteiros e operações binárias para representar conjuntos, subconjuntos e operações de conjuntos
- Exemplo, considere o conjunto universo:
  - `U = {1, 3, 7, 9}`
  - Vamos assumir que estes elementos estão armazenados em um vetor:
    - `U[0] = 1`; `U[1] = 3`; `U[2] = 7`; e U[3] = 9
  - Com 4 bits, conseguimos representar qualquer subconjunto de U
- Atenção à limitação de tamanho

Representação de conjunto

- `U = {1, 3, 7, 9}`
- Seja `x` um número inteiro
  - Se o `i`-ésimo bit de `x` for 1, então o elemento `U[i]` pertence ao subconjunto representado por `x`
  - Se o bit for 0, então `U[i]` não pertence ao subconjunto

Representação de conjunto

- U = {1, 3, 7, 9}
  - 5 em binário é 0101, representa {1, 7}
  - 1 em binário é 0001, representa {1}
  - 0 em binário é 0000, representa {}
  - 2 em binário é 0010, representa {3}
  - 15 em binário é 1111, representa {1, 3, 7, 9}

Representação de conjunto

- `U = {1, 3, 7, 9}`
  - `5` em binário é `0101`, representa `{1, 7}`
  - `2` em binário é `00010`, representa `{3}`

- União, operador `|`

  `5 | 2 = 0101 | 0010 = 0111: {1, 3, 7}`

Representação de conjunto

- Demais operações:
  - Interseção:
  - Diferença:
  - Pertinência:
  - Adição:
  - Remoção:

Representação de conjunto

- Demais operações:
  - Interseção: `A & B`
  - Diferença: `A & ~B`
  - Pertinência: `(A & 1 << i) != 0`
  - Adição: `A |= 1 << i`
  - Remoção: `A &= ~(1 << i)`

Enumerando subconjuntos

```c
#include <stdio.h>
int U[4] = {1, 3, 7, 9};
int N = 4;

void subconjuntos(int *v, int n) {
  // cada x é um conjunto
  for (int x = 0; x < 1 << n; x++) {
    printf("{ ");
    for (int i = 0; i < n; i++) {
      // verificando se U[i] está em x
      if ((x & (1 << i)) != 0)
        printf("%d ", U[i]);
    }
    printf("}\n");
  }
}

int main() {
  subconjuntos(U, N);
  return 0;
}
```

```
{ }
{ 1 }
{ 3 }
{ 1 3 }
{ 7 }
{ 1 7 }
{ 3 7 }
{ 1 3 7 }
{ 9 }
{ 1 9 }
{ 3 9 }
{ 1 3 9 }
{ 7 9 }
{ 1 7 9 }
{ 3 7 9 }
{ 1 3 7 9 }
```

Outras funções do C -- Eficientes

- `__builtin_ctz(n)` retorna quantos zeros há depois do bit mais à direita do número inteiro `n`
- `__builtin_clz(n)` retorna quantos zeros há antes do bit mais à esquerda do número inteiro `n`
- `__builtin_popcount(n)` retorna o número de bits 1 do inteiro `n`
- `__builtin_parity(n)` retorna 1 se o número de bits 1 de `n` é ímpar

- https://www.geeksforgeeks.org/builtin-functions-gcc-compiler/