Xử lý bit
BỒI DƯỠNG KIẾN THỨC HSG

Đỗ Phan Thuận

Email/Facebook: thuandp.sinhvien@gmail.com
Khoa Học Máy Tính
Đại Học Bách Khoa Hà Nội.

Ngày 6 tháng 7 năm 2023

Các kiểu dữ liệu cơ bản

- Các kiểu dữ liệu phải biết:
  - `bool`: biến bun (boolean) (`true`/`false`)
  - `char`: biến nguyên 8-bit (thường được sử dụng để biểu diễn các ký tự ASCII)
  - `short`: biến nguyên 16-bit
  - `int`/`long`: biến nguyên 32-bit
  - `long long`: biến nguyên 64-bit
  - `float`: biến thực 32-bit
  - `double`: biến thực 64-bit
  - `long double`: biến thực 128-bit
  - `string`: biến xâu ký tự

Các kiểu dữ liệu cơ bản

Loại	Số Byte	Giá trị nhỏ nhất	Giá trị lớn nhất
bool	1		
char	1	-128	127
short	2	-32768	32767
int/long	4	-2148364748	2147483647
long long	8	-9223372036854775808	9223372036854775807
	n	-2^{8n-1}	$2^{8n-1}-1$

Loại	Số Byte	Giá trị nhỏ nhất	Giá trị lớn nhất
unsigned char	1	0	255
unsigned short	2	0	65535
unsigned int	4	0	4294967295
unsigned long long	8	0	18446744073709551615
	n	0	$2^{8n}-1$

Loại	Số Byte	Giá trị nhỏ nhất	Giá trị lớn nhất	
float	4	$\approx -3.4 \times 10^{-38}$	$\approx 3.4 \times 10^{38}$	≈ 7 chữ số
double	8	$\approx -1.7 \times 10^{-308}$	$\approx 1.7 \times 10^{308}$	≈ 14 chữ số

Số nhị phân

- **Số nhị phân** là một số được biểu diễn trong hệ số cơ số 2 hoặc hệ số nhị phân, là một phương thức biểu thức toán học chỉ sử dụng hai ký hiệu: thường là "0" (không) và "1" (một) .
- Ký hiệu, một bit được **đặt**, nếu là 1 và **xóa** nếu là 0.
- Số nhị phân $(a_k a_{k-1} \ldots a_1 a_0)_2$ đại diện cho số:

$$(a_k a_{k-1} \ldots a_1 a_0)_2 = a_k \cdot 2^k + a_{k-1} \cdot 2^{k-1} + \cdots + a_1 \cdot 2^1 + a_0 \cdot 2^0.$$

Ví dụ, số nhị phân 1101_2 đại diện cho số 13:

$$\begin{aligned} 1101_2 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 13 \end{aligned}$$

Số nhị phân

- Máy tính biểu diễn số nguyên dưới dạng số nhị phân.
- Các số nguyên dương (cả có dấu và không dấu) chỉ được biểu diễn bằng các chữ số nhị phân của chúng và các số có dấu âm (có thể dương và âm) thường được biểu thị bằng Phần bù của hai.
- Phần bù của hai đạt được bằng cách:
  - Bước 1: bắt đầu với số dương tương đương.
  - Bước 2: đảo ngược (hoặc lật) tất cả các bit – thay đổi mọi 0 thành 1 và mọi 1 thành 0;
  - Bước 3: thêm 1 vào toàn bộ số vừa đảo ngược, bỏ qua bất kỳ số tràn nào. Tính toán tràn sẽ tạo ra giá trị sai cho kết quả.

```
unsigned int unsigned_number = 13;
assert(unsigned_number == 0b1101);

int positive_signed_number = 13;
assert(positive_signed_number == 0b1101);

int negative_signed_number = -13;
assert(negative_signed_number == 0b1111'1111'1111'1111'1111'1111'1111'0011);
```

Số nhị phân

- CPU thao tác rất nhanh với các bit đó với cơ chế xử lý riêng. Trong nhiều bài toán ta có thể tận dụng cách biểu diễn số nhị phân này để tăng tốc thời gian thực hiện.
- Đối với một số bài toán (thường là trong tổ hợp hoặc quy hoạch động), khi muốn theo dõi những đối tượng nào đã được chọn từ một tập hợp các đối tượng nhất định, ta chỉ cần sử dụng một số nguyên đủ lớn trong đó mỗi chữ số bit đại diện cho một đối tượng và tùy thuộc vào việc chọn hay không để gán 1 hoặc 0.

Toán tử theo bit (bitwise)

Tất cả các toán tử ở đây cùng tốc độ tính toán với phép cộng trên CPU đối với các số nguyên có độ dài cố định.

- & : Toán tử AND theo bit so sánh từng bit của toán hạng thứ nhất với bit tương ứng của toán hạng thứ hai. Nếu cả hai bit là 1, thì bit kết quả tương ứng là 1. Trái lại, bit kết quả tương ứng là 0. Ví dụ:

$$\begin{aligned} n &= 01011000 \\ n-1 &= 01010111 \\ \hline n\&(n-1) &= 01010000 \end{aligned}$$

- | : Toán tử OR theo bit so sánh từng bit của toán hạng đầu tiên với bit tương ứng của toán hạng thứ hai. Nếu một trong hai bit là 1, thì bit kết quả tương ứng là 1. Trái lại, bit kết quả tương ứng là 0. Ví dụ:

$$\begin{aligned} n &= 01011000 \\ n-1 &= 01010111 \\ \hline n|(n-1) &= 01011111 \end{aligned}$$

Toán tử theo bit (bitwise)

- $\wedge$: Toán tử XOR theo bit so sánh từng bit của toán hạng đầu tiên với bit tương ứng của toán hạng thứ hai. Nếu một bit là 0 và bit kia là 1, thì bit kết quả tương ứng là 1. Trái lại, bit kết quả tương ứng lad 0. Ví dụ:

$$\begin{aligned} n &= 01011000 \\ n-1 &= 01010111 \\ \hline n \wedge (n-1) &= 00001111 \end{aligned}$$

- $\sim$: Toán tử NOT lật từng bit của một số, nếu bit là 1 thì toán tử sẽ lật thành 0, nếu bit là 0 thì toán tử sẽ lật thành 1. Ví dụ:

$$\begin{aligned} n &= 01011000 \\ \hline \sim n &= 10100111 \end{aligned}$$

Toán tử dịch chuyển

Có hai toán tử để dịch chuyển bit.

- $\gg$: Dịch chuyển một số sang phải bằng cách xóa một vài chữ số nhị phân cuối cùng của số đó. Dịch chuyển một bước tương ứng với một phép chia số nguyên cho 2, do đó, phép dịch phải k thể hiện phép chia số nguyên cho 2^k. Ví dụ:

  $$5 \gg 2 = 101_2 \gg 2 = 1_2 = 1 \text{ giống với } \frac{5}{2^2} = \frac{5}{4} = 1$$

  Đối với máy tính, việc dịch chuyển bit nhanh hơn rất nhiều so với thực hiện phép chia.

- $\ll$: Dịch chuyển một số sang trái bằng cách nối thêm các chữ số 0. Tương tự, phép dịch trái k thể hiện phép nhân với 2^k. Ví dụ:

  $$5 \ll 3 = 101_2 \ll 3 = 101000_2 = 40$$

  giống với $5 \cdot 2^3 = 5 \cdot 8 = 40$

  Lưu ý: đối với một số nguyên có độ dài cố định, dịch trái là loại bỏ các chữ số bên trái nhất và nếu dịch trái quá nhiều, kết quả nhận được sẽ là 0.

Thay đổi một bit

Sử dụng dịch chuyển theo bit và một số thao tác theo bit cơ bản, ta có thể dễ dàng bật, lật hoặc tắt một bit.

- $1 \ll x$ là một số chỉ có bit thứ x được bật, trong khi $\sim (1 \ll x)$ là một số có tất cả các bit được bật ngoại trừ bit thứ x.
- $n \mid (1 \ll x)$ bật bit thứ x của số n
- $n \ \wedge \ (1 \ll x)$ lật bit thứ x của số n
- $n \ \& \ \sim (1 \ll x)$ tắt bit thứ x của số n

Kiểm tra một bit được bật hay không

Giá trị của bit thứ x có thể xác định bằng cách dịch x vị trí sang bên phải và thực hiện thao tác & theo bit với 1.

```
1 bool is_set(unsigned int num, int x) {
2     return (num >> x) & 1;
3 }
```

Kiểm tra xem một số nguyên có phải là lũy thừa của 2 không

- Lũy thừa của hai là một số chỉ có một bit duy nhất được bật (ví dụ: $32 = 0010\ 0000_2$), trong khi số liền trước của số đó có bit đó không bật còn tất cả các bit sau nó đều được bật ($31 = 0001\ 1111_2$).
- Phép AND theo bit của một số với số liền trước nó sẽ bằng 0 chỉ trong trường hợp số đó là luỹ thừa của hai. Từ đó ta có phép kiểm tra luỹ thừa của hai:

```
1 bool isPowerOfTwo(unsigned int n) {
2     return n && !(n & (n - 1));
3 }
```

Tắt bit được bật bên phải nhất

- Biểu thức $n \,\&\, (n-1)$ có thể được sử dụng để tắt bit được đặt ngoài cùng bên phải của một số n.
- Giải thích: Biểu thức $n-1$ lật tất cả các bit sau bit được đặt ngoài cùng bên phải của n, bao gồm cả bit được đặt ngoài cùng bên phải. Vì vậy, bằng cách thực hiện phép AND theo bit tất cả các bit đó được tắt thành 0, nghĩa là bit bật ngoài cùng bên phải của số ban đầu n được tắt.
  Ví dụ, xét số $52 = 00110100_2$:

$$\begin{aligned} n &= 00110100 \\ n-1 &= 00110011 \\ - \; - \; - \; - \; - \; - \; - \; - \; - \; - \; - \; - \\ n\&(n-1) &= 00110000 \end{aligned}$$

Thuật toán Brian Kernighan

Đếm số bit được bật với biểu thức trên.

- Ý tưởng là chỉ xem xét các bit đã bật của một số nguyên bằng cách tắt bit đã bật ngoài cùng bên phải của nó (sau khi đếm nó), do đó, lần lặp tiếp theo sẽ xem xét bit ngoài cùng bên phải kế tiếp.

```
1  int countSetBits(int n)
2  {
3      int count = 0;
4      while (n)
5      {
6          n = n & (n - 1);
7          count++;
8      }
9      return count;
10 }
```

Thủ thuật khác

- n & $(n+1)$ tắt tất cả các bit 1 cuối : $0011\ 0111_2 \rightarrow 0011\ 0000_2$.
- $n \mid (n+1)$ bật bit 0 cuối cùng: $0011\ 0101_2 \rightarrow 0011\ 0111_2$.
- n & $-n$ trích bit 1 cuối cùng: $0011\ 0100_2 \rightarrow 0000\ 0100_2$.

Tham khảo thêm nhiều thủ thuật trong cuốn sách Hacker's Delight.

Biểu diễn tập hợp

- Cho một số lượng nhỏ ($n \leq 30$) phần tử
- Gán nhãn bởi các số nguyên $0, 1, \ldots, n-1$
- Biểu diễn tập hợp các phần tử này bởi một biến nguyên 32-bit
- Phần thử thứ i trong tập được biểu diễn bởi số nguyên x nếu bit thứ i của x là 1
- Ví dụ:
  - Cho tập hợp $\{0, 3, 4\}$
  - `int x = (1<<0) | (1<<3) | (1<<4);`

Biểu diễn tập hợp

- Tập rỗng: `0`
- Tập có một phần tử: `1 << i`
- Tập vũ trụ (nghĩa là tập tất cả các phần tử): `(1 << n) - 1`
- Hợp hai tập: `x | y`
- Giao hai tập: `x & y`
- Phần bù một tập: `~x & ((1 << n) - 1)`

Biểu diễn tập hợp

- Kiểm tra một phần tử xuất hiện trong tập hợp:

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

Biểu diễn tập hợp

- Tại sao nên làm như vậy mà không dùng set<int>?
- Biểu diễn đỡ tốn khá nhiều bộ nhớ (nén 32,64,128 lần)
- Tất cả các tập con của tập n phần tử này có thể biểu diễn bởi các số nguyên trong khoảng $0 \dots 2^n - 1$
- Dễ dàng lặp qua tất cả các tập con
- Dễ dàng sử dụng một tập hợp như một chỉ số của một mảng

Giới thiệu

- Trong STL C++ có một cấu trúc dữ liệu biểu diễn tập hợp các bit, có thể được sử dụng để hỗ trợ các thao tác xử lý bit rất tốt, đó là `bitset`.
- Về bản chất, `bitset` giống như một mảng/vector kiểu logic, tức là nó gồm các phần tử chỉ mang giá trị 0/1, tuy nhiên nó được cài đặt một cách tối ưu sao cho mỗi phần tử chỉ chiếm đúng một bit.
- Vì vậy, không gian lưu trữ của một `bitset` sẽ nhỏ hơn so với một mảng/vector kiểu bool. Các thao tác trên `bitset` cũng nhanh hơn so với mảng/vector tới 64 lần.

Khai báo bitset

Để khai báo một bitset, ta sử dụng thư viện <bitset> và khai báo theo cú pháp dưới đây:

```
#include <bitset>

using namespace std;

bitset < size > {name}(intialization);
```

Trong đó:

- Tham số size thể hiện kích thước của bitset, tham số này bắt buộc phải có khi khai báo một bitset. Đây cũng là nhược điểm của bitset đó là kích thước phải cố định trước.
- Tham số name thể hiện tên của bitset.

Khai báo bitset

- Tham số intialization thể hiện khởi tạo giá trị ban đầu của bitset. Có 3 cách để khởi tạo:
  - Không khởi tạo: Với cách này, bitset sẽ mặc định toàn bit 0.

```
bitset < 8 > unintialized_bitset; // 00000000.
```

  - Khởi tạo bằng số nguyên hệ thập phân: Với cách làm này, bitset sẽ được khởi tạo bằng với biểu diễn nhị phân của số truyền vào.

```
bitset < 8 > decimal_bitset(15); // 00001111
```

  - Khởi tạo bằng xâu nhị phân: Với cách làm này, bitset sẽ được khởi tạo giá trị bằng với xâu nhị phân truyền vào. Tuy nhiên, xâu truyền vào không được phép có độ dài lớn hơn kích thước của bitset.

```
bitset < 8 > string_bitset(string("1111")); // 000
// Hoặc bitset < 8 > string_bitset("1111");
```

Truy cập các bit của bitset

bitset là một cấu trúc dữ liệu rất đặc biệt, nó vừa giống xâu kí tự lại vừa giống mảng. Giống xâu ở chỗ, nó có thể được khởi tạo từ một xâu nhị phân, và in ra cả dãy bit trực tiếp thông qua câu lệnh cout (chứ không cần for qua từng phần tử rồi in), nhưng lại giống mảng ở chỗ cũng truy cập được vào từng phần tử thông qua toán tử []. Tuy nhiên, các bit trong bitset lại được đánh số thứ tự từ phải qua trái (giống với quy tắc đánh số bit trong hệ nhị phân), bắt đầu từ 0. Ví dụ:

```
1  #include <iostream>
2  #include <bitset>
3  using namespace std;
4  main()
5  {
6      bitset < 8 > sample_bs("00001111");
7      cout << "Bit o vi tri 4 cua sample_bs la " << sample_bs[4];
8      return 0;
9  }
```

Kết quả đoạn code trên sẽ là:

```
Bit tai vi tri 4 cua sample_bs la 0
```

Một số hàm và toán tử của `bitset`

- `set()`, `reset()`, `flip()`, `count()`, `size()`, `any()`, `none()`, `to_string()`, `to_ulong()`, `to_ullong()`
- Các toán tử giống như thao tác theo bit: `&`, `|`, `!`, `«=`, `»=`, `&=`, `|=`, $\wedge$`=`, $\sim$

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

soict.hust.edu.vn/ fb.com/groups/soict