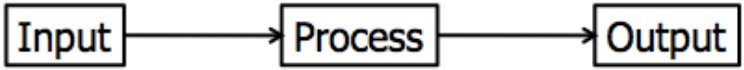
Data Processing

Lau Chi Yung

9 Feb, 2019

Data Processing



Outline

Outline

Integers

unsigned 8-bit integer	
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
...	
01111110	126
01111111	127
10000000	128
10000001	129
...	
11111110	254
11111111	255

signed 8-bit integer	
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
...	
01111110	126
01111111	127
10000000	-128
10000001	-127
...	
11111110	-2
11111111	-1

Integers

- Underlying addition procedure is the same for both types, the only difference is the representation of the outcome

unsigned			signed
129		10000001	-127
1	+	00000001	1
130		10000010	-126

Integers

- Underlying addition procedure is the same for both types, the only difference is the representation of the outcome

unsigned			signed
129		10000001	-127
1	+	00000001	1
130		10000010	-126

- Beware of overflow

8-bit unsigned	8-bit signed
9 - 10 = 255	127 + 127 = -2

Integers

C/C++	Pascal	Range
unsigned char	byte	0 255
unsigned short	word	0 65535
unsigned int unsigned	longword	0 4294967295
unsigned long long	qword	0 18446744073709551615

- Unsigned integers are rarely used, unless the task constraints are deliberately designed

Integers

C/C++	Pascal	Range
signed char	shortint	-128 127
short	smallint	-32768 32767
int long[1]	**longint**	-2147483648 2147483647
long long	int64	-9223372036854775808 9223372036854775807

- Use **int**/**longint** for most cases
- Use **long long**/**int64** to avoid overflow
- `https://plus.google.com/+YouTube/posts/BUXfdWqu86Q`

[1] different on 32/64-bit machines

Outline

Decimal numbers

- Want to store real numbers corrected to 2 decimal places within $[-100000.00, 100000.00]$

Decimal numbers

- Want to store real numbers corrected to 2 decimal places within $[-100000.00, 100000.00]$
- Just use 32-bit signed integer :D

Decimal numbers

- Want to store real numbers corrected to 2 decimal places within $[-100000.00, 100000.00]$
- Just use 32-bit signed integer :D

- Now want to store real numbers corrected to 6 significant figures within $[-10^{20}, 10^{20}]$

Decimal numbers

Fixed-point

- 100.00
- 203.75
- 0.02

Floating-point

- 100.00032
- 20.3
- 3254×10^{15}

Floating points

Size	C/C++	Pascal	Range
32 bits	float	single	$\pm 3.40 \times 10^{38}$
64 bits	double	double	$\pm 1.78 \times 10^{308}$

Floating points

Size	C/C++	Pascal	Range
32 bits	float	single	$\pm 3.40 \times 10^{38}$
64 bits	double	double	$\pm 1.78 \times 10^{308}$

How many integers are in the range [0, 1]?

2

How many real numbers are in the range [0, 1]?

Infinite

Then how to store $\pm 3.40 \times 10^{38}$ in 32 bits?

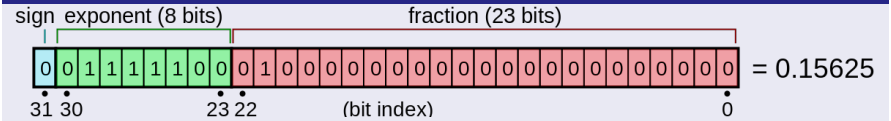
Floating points

- Break a real number into **sign**, **significand** and **exponent**
- $3.14159 = 314159 \times 10^{-5}$
-

sign	significand	exponent
+	314159	-5

Floating points

IEEE-754 Single-precision floating-point format

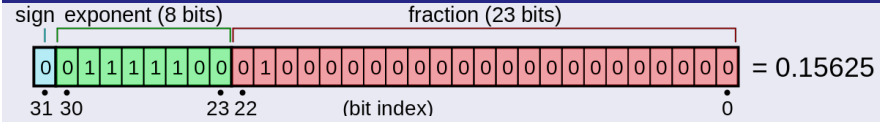


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits

[2] only need 23 bits storage

Floating points

IEEE-754 Single-precision floating-point format

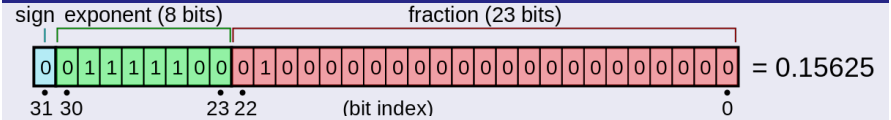


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits
- Can **float** represent integers in $\pm(2^{24}-1)$ exactly?
- Can **float** represent 0.5 exactly?
- Can **float** represent 0.3 exactly?

[2] only need 23 bits storage

Floating points

IEEE-754 Single-precision floating-point format

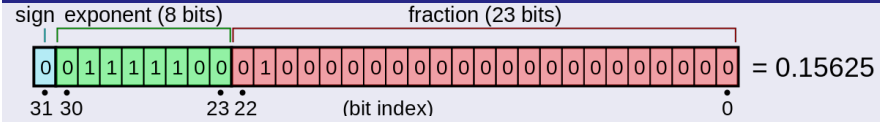


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits
- Can **float** represent integers in $\pm(2^{24}-1)$ exactly? yes
- Can **float** represent 0.5 exactly?
- Can **float** represent 0.3 exactly?

[2] only need 23 bits storage

Floating points

IEEE-754 Single-precision floating-point format

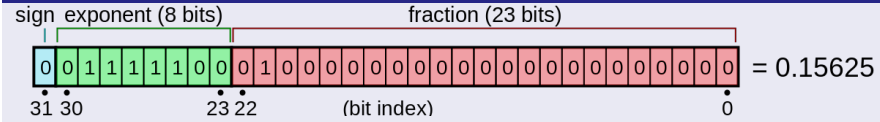


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits
- Can **float** represent integers in $\pm(2^{24}-1)$ exactly? yes
- Can **float** represent 0.5 exactly? yes
- Can **float** represent 0.3 exactly?

[2] only need 23 bits storage

Floating points

IEEE-754 Single-precision floating-point format

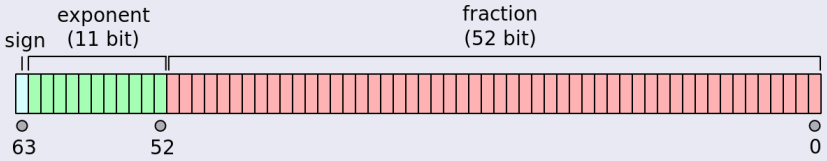


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits
- Can **float** represent integers in $\pm(2^{24}-1)$ exactly? yes
- Can **float** represent 0.5 exactly? yes
- Can **float** represent 0.3 exactly? no

[2] only need 23 bits storage

Floating points

IEEE-754 Double-precision floating-point format



- Only approximate the first 53 significant bits of a real number[3]
- 15.95 significant decimal digits
- Can represent integers in $\pm$9007199254740991 exactly

[3] only need 52 bits storage

Floating points

Print the underlying representation of a **double**

```
double x = 0.1;
cout << hex << *(long long*)&x << endl;
```

```
uses sysutils;
type int64Ptr = ^int64;
var x: double = 0.1;
begin
  writeln(inttohex(int64Ptr(@x)^, 16))
end;
```

Precision error

- 0.3 + 0.3 = 0.6? true
- 0.2 + 0.4 = 0.6? false!

Precision error

- 0.3 + 0.3 = 0.6? true
- 0.2 + 0.4 = 0.6? false!
- $0.6 = 1.00110011\ldots00110011 \times 2^{-1}$
- $0.3 + 0.3 = 1.00110011\ldots00110011 \times 2^{-1}$
- $0.2 + 0.4 = 1.00110011\ldots00110100 \times 2^{-1}$

Precision error

- Solution: allow a small *epsilon* offset when comparing equality

```
#include <cmath>
bool equal(double a, double b) {
  return fabs(a - b) < 1e-9;
}
```

```
function equal(a, b: double): boolean;
begin
  if abs(a - b) < 1e-9 then
    equal := true
  else
    equal := false
end;
```

Precision error

- We want
  - 0.2 + 0.4 = 0.6 true
  - 0.2 + 0.4 > 0.6 false
  - 0.2 + 0.4 < 0.6 false
- let ϵ = 0.000000001

	Which one should we use?	
a = b	abs(a - b) $<\epsilon$	abs(a - b) $>\epsilon$

Precision error

- We want
  - 0.2 + 0.4 = 0.6 true
  - 0.2 + 0.4 > 0.6 false
  - 0.2 + 0.4 < 0.6 false
- let ϵ = 0.000000001

	Which one should we use?	
a = b	abs(a - b) $< \epsilon$	abs(a - b) $> \epsilon$

Precision error

- We want
  - 0.2 + 0.4 = 0.6 true
  - 0.2 + 0.4 > 0.6 false
  - 0.2 + 0.4 < 0.6 false
- let ϵ = 0.000000001

	Which one should we use?	
a = b	abs(a - b) $< \epsilon$	abs(a - b) $> \epsilon$
a $>$ b	a $>$ b + ϵ	a + $\epsilon >$ b

Precision error

- We want
  - 0.2 + 0.4 = 0.6 true
  - 0.2 + 0.4 > 0.6 false
  - 0.2 + 0.4 < 0.6 false
- let ϵ = 0.000000001

	Which one should we use?	
a = b	abs(a - b) $< \epsilon$	abs(a - b) $> \epsilon$
a $>$ b	a $>$ b + ϵ	a + ϵ $>$ b

Precision error

- We want
  - 0.2 + 0.4 = 0.6 true
  - 0.2 + 0.4 > 0.6 false
  - 0.2 + 0.4 < 0.6 false
- let ϵ = 0.000000001

	Which one should we use?	
a = b	abs(a - b) $< \epsilon$	abs(a - b) $> \epsilon$
a $>$ b	a $>$ b + ϵ	a + $\epsilon >$ b
a $<$ b	a $<$ b + ϵ	a + $\epsilon <$ b

Precision error

- We want
  - 0.2 + 0.4 = 0.6 true
  - 0.2 + 0.4 > 0.6 false
  - 0.2 + 0.4 < 0.6 false
- let ϵ = 0.000000001

	Which one should we use?	
a = b	abs(a - b) $< \epsilon$	abs(a - b) $> \epsilon$
a $>$ b	a $>$ b + ϵ	a + $\epsilon >$ b
a $<$ b	a $<$ b + ϵ	a + $\epsilon <$ b

Precision error

- We want
  - 0.2 + 0.4 = 0.6 true
  - 0.2 + 0.4 > 0.6 false
  - 0.2 + 0.4 < 0.6 false
- let ϵ = 0.000000001

	Which one should we use?	
a = b	abs(a - b) $< \epsilon$	abs(a - b) $> \epsilon$
a $>$ b	a $>$ b + ϵ	a + $\epsilon >$ b
a $<$ b	a $<$ b + ϵ	a + $\epsilon <$ b

- The above depends on the task context
- Usually, the task will accept a floating point answer within an error range

Type conversion


```c
int i;
double d;

i = 1000;
d = i;

d = 1000.0;
i = d;
```

```pascal
var
  i: longint;
  d: double;
begin
  i := 1000;
  d := i;

  d := 1000.0;
  i := trunc(d)
end.
```

- When assigning to a different type, precision may lose

Literals

		C/C++	Pascal
Integer	Decimal	102	102
	Octal	0146	&146
	Hexadecimal	0x66	$66
	Binary notation	0b1100110[4]	%1100110
Floating point		1.2345	1.2345
		12345e-4	12345e-4
		0.012345e2	0.012345e2
Character		'a'	'a'
String		"apple"	'apple'

- for C/C++ **long long**, append "LL"
- e.g. "`long long x = 9223372036854775807LL;`"

[4] GCC extension

The most useful numeric data type

If you are allowed to use only one numeric data type, which one will you choose?

A char/byte
B int/longint
C long long/int64
D double/double

The most useful numeric data type

If you are allowed to use only one numeric data type, which one will you choose?

A char/byte

B int/longint

C long long/int64

D double/double

In Javascript, the only numeric data type is double.

Summary

Size	C/C++	Pascal	Range
32 bits	int	longint	$\pm 10^9$
64 bits	long long	int64	$\pm 10^{18}$
64 bits	double	double	real number: $\pm 1.78 \times 10^{308}$ integer: $\pm 10^{15}$

Outline

Character

- In C/C++, **char** is just an 8-bit integer (signed or unsigned depending on machine)
- In Pascal, **char** is also 8-bit but not an integer

```
char c = 'c';
char d = 99;
int i = c * 10;
```

```
var
  c, d: char;
  i: longint;
begin
  c := 'c';
  d := chr(99);
  i := ord(c) * 10
end.
```

- The value of i is 990

String

- In Pascal, there are two types of strings
  - **string**: stores at most 255 characters (don't use this)
  - **ansistring**: no limit
- **ansistring** is null-terminated

```
string s = "hello";
cout << s.length() << endl;
cout << s[0] << endl;
cout << s << endl;

//5
//h
//hello
```

```
var s: ansistring;
begin
  s := 'hello';
  writeln(length(s));
  writeln(s[1]);
  writeln(s)
end.
```

Number to string


```
//#include <cstdio>
char s[10];
sprintf(s, "%d", 123);
```

```
var s: ansistring;
begin
  str(123, s)
end.
```

String to number


```cpp
//#include <cstdlib>
int x = atoi("123");
```

```pascal
var x: longint;
begin
  val('123', x)
end.
```

Outline

Header file for C++

- In C++, add

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

One character


```
int c = cin.get();
cout << c;
```

```
var c: char;
begin
  read(c);
  write(c)
end.
```

- note that `cin.get()` returns **int**
- `cin >> c` behaves differently as it would skip whitespaces

One line (naive)


```
string s;
while (true) {
  int c = cin.get();
  if (c == '\n')
    break;
  s += c;
}
cout << s << endl;
```

```
var
  c: char;
  s: ansistring;
begin
  while true do
  begin
    read(c);
    if ord(c) = 10 then
      break;
    s := s + c
  end;
  writeln(s)
end.
```

- didn't handle EOF correctly
- didn't handle EOL correctly

One line (smart)


```
string s;
getline(cin, s);
cout << s << endl;
```

```
var s: ansistring;
begin
  readln(s);
  writeln(s)
end.
```

- On Windows, EOL (end-of-line) is the two characters "\r\n" ('#13#10' in Pascal)
- On Linux, EOL is '\n' ('#10' in Pascal)
- `getline()` and `readln()` read and discard EOL, so `s` does not contain EOL

N lines


```cpp
string s;
int i, n = 10;
for (i = 0; i < n; i++) {
  getline(cin, s);
  cout << s << endl;
}
```

```pascal
var
  s: ansistring;
  i: longint;
  n: longint = 10;
begin
  for i := 1 to n do
  begin
    readln(s);
    writeln(s)
  end
end.
```

All lines until EOF


```
string s;
while (getline(cin, s))
  cout << s << endl;
```

```
var s: ansistring;
begin
  while not eof do
  begin
    readln(s);
    writeln(s)
  end
end.
```

- **getline** returns **cin**, which evaluates to false when nothing was read
- **eof** is a function called with no arguments

Space-separated words on a line

Excerpt from S184 Bogo Translate

Solutions from contestants:

- Keep reading words, until the next character is a newline
- Keep reading words, until the word contains uppercase letters
- Read entire line, scan for spaces to break it into words

Input

```
3
charlieli
G
G
i am charlieli
XYZ
XZY
i like coding
SVO
SOV
```

Space-separated words on a line


```
//#include <sstream>
string line, word[100];
getline(cin, line);
stringstream ss(line);
int i = 0;
while (ss >> word[i])
  i++;
```

One number


```cpp
int a;
//double a;
//long long a;
cin >> a;
cout << a << endl;
```

```
var
  a: longint;
  {a: double;}
  {a: int64;}
begin
  read(a);
  writeln(a);
end.
```

- `cin` skips all whitespaces and EOLs
- `read` (for numbers) skps all whitespaces and EOLs

All space-separated numbers until EOF

Input

```
3    5  7
  9
```

```cpp
int x;
while (cin >> x)
  cout << x << endl;
```

```
var x: longint;
begin
  while not eof do
  begin
    read(x);
    writeln(x)
  end
end.
```

- All whitespaces and EOLs are skipped

Formatted output


```
//#include<iomanip>
cout << "abc" << endl;
cout << setw(8) << "abc" << endl;
cout << 13 << endl;
cout << setw(8) << 13 << endl;
cout << -13.875 << endl;
cout << setw(8) << setprecision(2)
     << fixed << -13.875 << endl;
```

```
begin
  writeln('abc');
  writeln('abc':8);
  writeln(13);
  writeln(13:8);
  writeln(-13.875);
  writeln(-13.875:8:2)
end.
```

Output

```
abc
     abc
13
      13
-13.875
  -13.88
```

Output

```
abc
     abc
13
      13
-1.387500000E+01
  -13.88
```

Caveats

- Lead zeroes (both input and output)
- Number of decimals
- Zero
- Spaces
- EOLs

Outline

File I/O

- Read from a file, write to a file
- Required in some contests, e.g. NOIP
- Do not use file I/O in HKOI

File I/O


```cpp
//#include <fstream>
fstream fin(
  "in.txt",
  fstream::in
);
fstream fout(
  "out.txt",
  fstream::out
);
int x;
fin >> x;
fout << x << endl;
fin.close();
fout.close();
```

```
var
  fin, fout: text;
  x: longint;
begin
  assign(fin, 'in.txt');
  reset(fin);
  assign(fout, 'out.txt');
  rewrite(fout);
  read(fin, x);
  write(fout, x);
  close(fin);
  close(fout)
end.
```

- It is good to close the files

String functions

- shorten your code
- make it more readable and debuggable
- avoid making mistakes when implementing these functions by yourselves

String comparison


```
string abc = "abc";
string def = "def";
abc < def
abc == "abc"
```

```
'abc' < 'def'
'abc' = 'abc'
```

- Dictionary ordering

find / pos


```
string s = "abc";
int i = s.find("bc");
int j = s.find('c');
//i = 1, j = 2
```

```
var
  s: ansistring;
  i, j: longint;
begin
  s := 'abc';
  i := pos('bc', s);
  j := pos('c', s)
  {i = 2, j = 3}
end.
```

- `s.find(t)` returns
  - The position of the first occurrence of `t` in `s`
  - `string::npos` if not found
- `pos(t, s)` returns
  - The position of the first occurrence of `t` in `s`
  - 0 if not found
- Note that **ansistring** is 1-based

substr / copy


```
string s = "abc";
string t1 = s.substr(0, s.length());
string t2 = s.substr(1, 2);
//t1 = "abc"
//t2 = "bc"
```

```
var s, t1, t2: ansistring;
begin
  s := 'abc';
  t1 := copy(s, 1, length(s));
  t2 := copy(s, 2, 2)
end.
```

- Returns a copy

+ / concat


```
string s = "cd";
s = "ab" + s + "ef";
//s = "abcdef"
```

```
var s: ansistring = 'cd';
begin
  s := concat('ab', s, 'ef')
end.
```

- Concatenate
- `concat` accepts many arguments and creates a new string

insert


```
string s = "ad";
s.insert(1, "bc");
//s = "abcd"
```

```
var s: ansistring = 'ad';
begin
  insert('bc', s, 2)
end.
```

erase / delete


```
string s = "axxbc";
s.erase(1, 2);
//s = "abc"
```

```
var s: ansistring = 'axxbc';
begin
  delete(s, 2, 2)
end.
```

Practice

01060 Depreciation

01009 Words

01007 Packet Re-assembly

S184 Bogo Translate

01000 Append Insert Replace

01012 Allocating School Places

01001 TeX Processing

01013 Internet Usage Bills

10001 Data, Data, Everywhere

J021 Date Sorting