Data Processing

Lau Chi Yung

3 Feb, 2018

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

- 1 bit can represent 2 elements
- 8 bits can represent elements

Integers

- 1 bit can represent 2 elements
- 8 bits can represent $2^8 = 256$ elements

Integers

- 1 bit can represent 2 elements
- 8 bits can represent $2^8 = 256$ elements
- 8-bit unsigned integer represents the 256 integers in $[0, 255]$
- 8-bit signed integer represents
  - 128 negative integers $[-128, -1]$ and
  - 128 non-negative integers $[0, 127]$

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

- Applying two's-complement flips the sign of a signed integer
- Two's-complement: flip all bits and add one

	01111110	(126)
$\Rightarrow$	10000001	
$\Rightarrow$	10000010	(-126)

Integers

- Beware of overflow

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

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

[1] different on 32/64-bit machines

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

Outline

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]?

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?

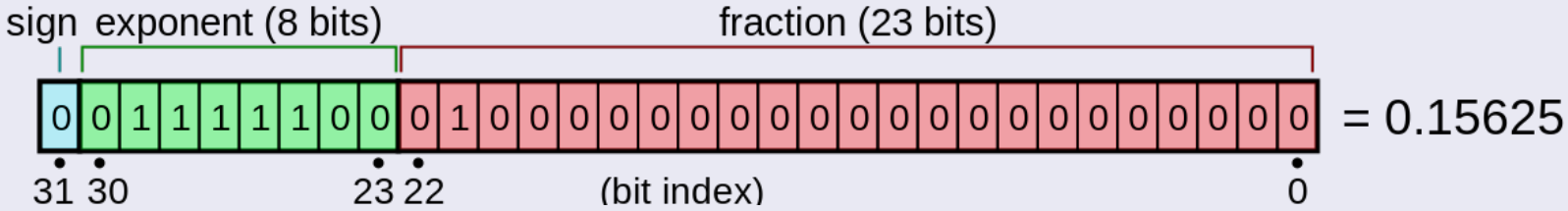
Floating points

- 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

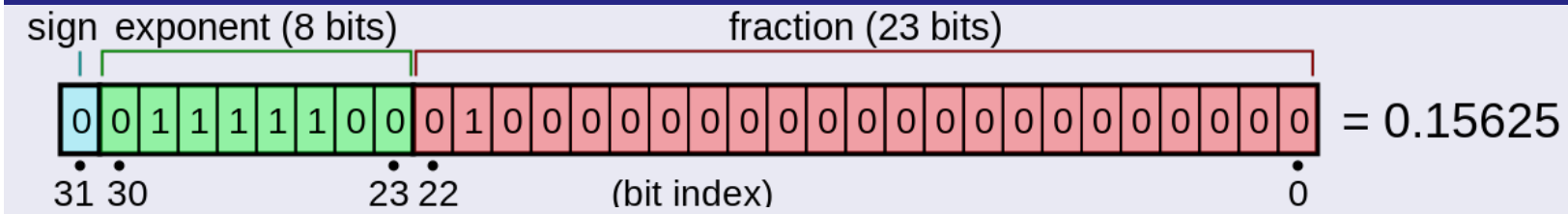


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits
- Faster than **double**

[2] only need 23 bits storage

Floating points

IEEE-754 Single-precision floating-point format

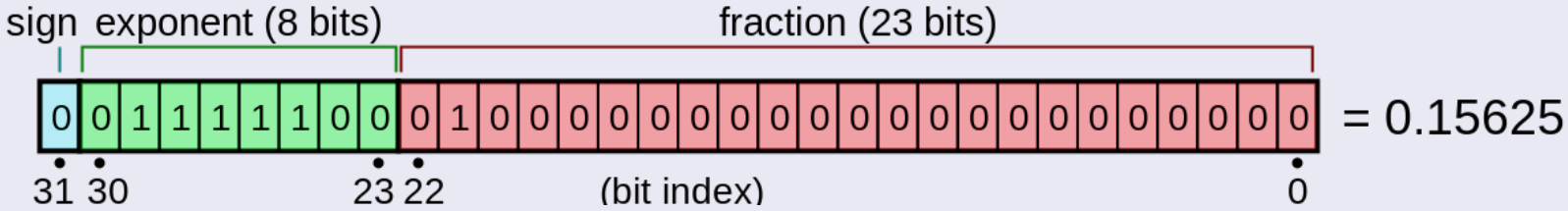


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits
- Faster than **double**
- Can **float** represent integers in ± 16777215 exactly?
- Can **float** represent 0.5 exactly?
- Can **float** represent 0.3 exactly?

[2] only need 23 bits storage

Floating points

IEEE-754 Single-precision floating-point format

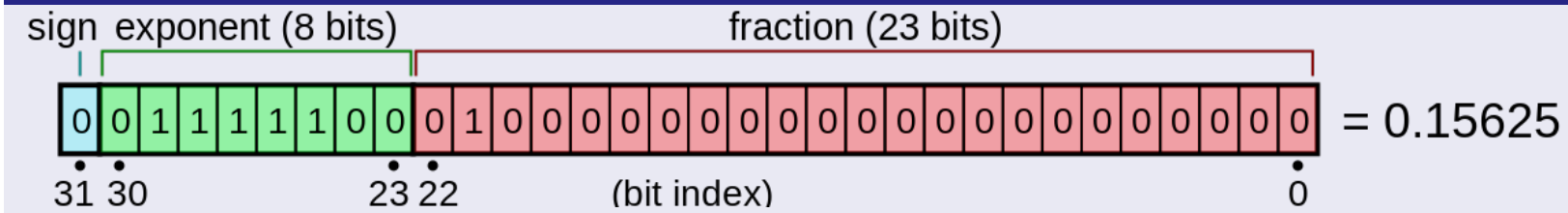


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits
- Faster than **double**
- Can **float** represent integers in ± 16777215 exactly? yes
- Can **float** represent 0.5 exactly?
- Can **float** represent 0.3 exactly?

[2] only need 23 bits storage

Floating points

IEEE-754 Single-precision floating-point format

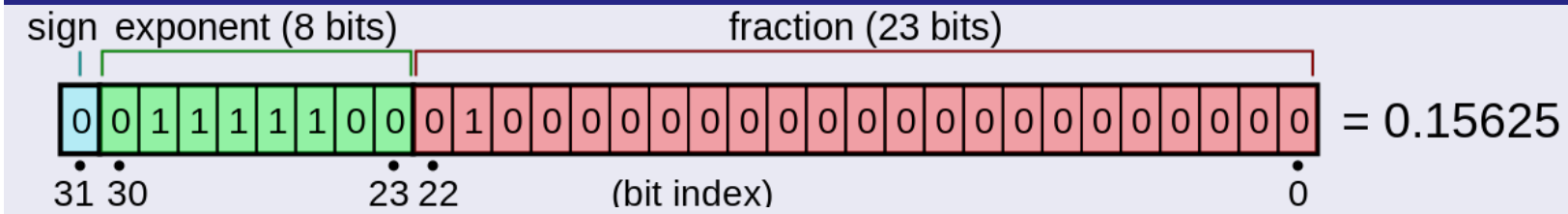


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits
- Faster than **double**
- Can **float** represent integers in ± 16777215 exactly? yes
- Can **float** represent 0.5 exactly? yes
- Can **float** represent 0.3 exactly?

[2] only need 23 bits storage

Floating points

IEEE-754 Single-precision floating-point format

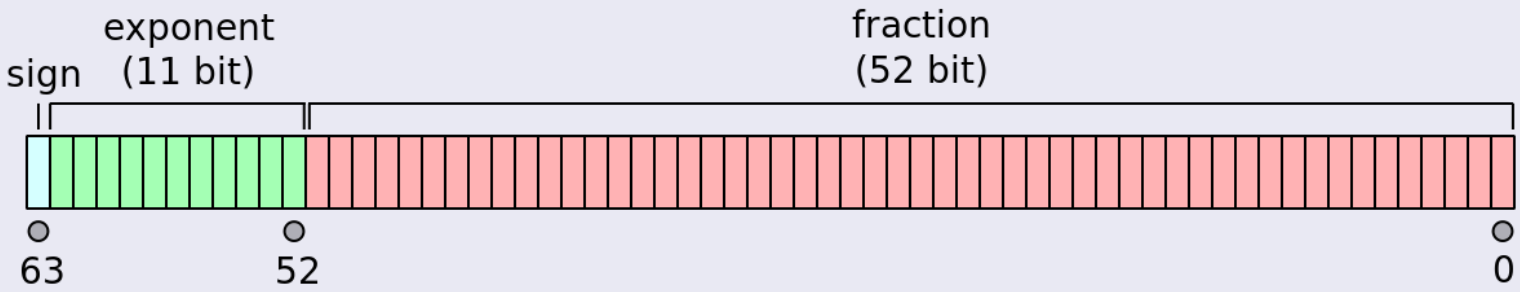


- Only approximate the first 24 significant bits of a real number[2]
- 7.22 significant decimal digits
- Faster than **double**
- Can **float** represent integers in ± 16777215 exactly? yes
- Can **float** represent 0.5 exactly? yes
- Can **float** represent 0.3 exactly? no

[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 ± 9007199254740991 exactly

[3] only need 52 bits storage

Floating points

Precision error

- $3.0 / 3.0 = 1.0$?

Floating points

Precision error

- $3.0 / 3.0 = 1.0$? true
- $0.3 / 3.0 = 0.1$?

Floating points

Precision error

- $3.0 / 3.0 = 1.0$? true
- $0.3 / 3.0 = 0.1$? false!

Floating points

Precision error

- 3.0 / 3.0 = 1.0? true
- 0.3 / 3.0 = 0.1? false!
- 0.3 / 3.0 = $1.10011001\ldots10011001 \times 2^{-3}$
- 0.1 = $1.10011001\ldots10011010 \times 2^{-3}$

Floating points

Precision error

- 3.0 / 3.0 = 1.0? true
- 0.3 / 3.0 = 0.1? false!
- 0.3 / 3.0 = $1.10011001\ldots10011001 \times 2^{-3}$
- 0.1 = $1.10011001\ldots10011010 \times 2^{-3}$

How did I get the underlying representation of a **double**?

Floating points

Precision error

- 3.0 / 3.0 = 1.0? true
- 0.3 / 3.0 = 0.1? false!
- 0.3 / 3.0 = $1.10011001\ldots10011001 \times 2^{-3}$
- 0.1 = $1.10011001\ldots10011010 \times 2^{-3}$

How did I get the underlying representation of a **double**?

```
double x = 0.1;
printf("%llx\n", *(long long*)&x);
```

Floating points

- Solution: use a small *epsilon* 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;
```

Floating points

- We want:
  - 0.3 / 3.0 = 0.1 true
  - 0.3 / 3.0 > 0.1 false
  - 0.3 / 3.0 < 0.1 false
- let $\epsilon = 0.000000001$

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

Floating points

- We want:
  - 0.3 / 3.0 = 0.1 true
  - 0.3 / 3.0 > 0.1 false
  - 0.3 / 3.0 < 0.1 false
- let $\epsilon = 0.000000001$

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

Floating points

- We want:
  - 0.3 / 3.0 = 0.1 true
  - 0.3 / 3.0 > 0.1 false
  - 0.3 / 3.0 < 0.1 false
- let $\epsilon = 0.000000001$

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

Floating points

- We want:
  - 0.3 / 3.0 = 0.1 true
  - 0.3 / 3.0 > 0.1 false
  - 0.3 / 3.0 < 0.1 false
- let $\epsilon = 0.000000001$

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

Floating points

- We want:
  - 0.3 / 3.0 = 0.1 true
  - 0.3 / 3.0 > 0.1 false
  - 0.3 / 3.0 < 0.1 false
- let $\epsilon = 0.000000001$

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

Floating points

- We want:
  - 0.3 / 3.0 = 0.1 true
  - 0.3 / 3.0 > 0.1 false
  - 0.3 / 3.0 < 0.1 false
- let $\epsilon = 0.000000001$

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

Floating points

- We want:
  - 0.3 / 3.0 = 0.1 true
  - 0.3 / 3.0 > 0.1 false
  - 0.3 / 3.0 < 0.1 false
- let $\epsilon = 0.000000001$

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

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

The most useful numeric data type

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

A char

B int

C long long

D 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

B int

C long long

D double

In Javascript, the only numeric data type is double.

Summary

Size	C/C++	Pascal	Range
32 bits	int	longint	-2147483648 2147483647
64 bits	long long	int64	-9223372036854775808 9223372036854775807
64 bits	double	double	real number: $\pm 1.78 \times 10^{308}$ integer: ± 9007199254740991

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

Outline

Character

- In 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

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

```pascal
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 C, string is just an array of **char** with a null character at the end
- In Pascal, there are two types of strings
  - **string**: stores at most 255 characters (don't use this)
  - **ansistring**: no limit
- Theoretically you can use an array of **char** in Pascal too, but it will be troublesome
- **ansistring** is null-terminated

```
char s[] = "abc";
char t[4] = "abc";
char u[4] =
  {'a', 'b', 'c', '\0'};
```

```
var s : ansistring =
  'hello';
```

Outline

Header files for C/C++

- In C, add "#include<stdio.h>"
- In C++, add "#include<cstdio>"

One character


```c
char c;
scanf("%c", &c);
printf("%c", c);

int d = getchar();
putchar(d);
```

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

- note that `getchar()` returns **int**

One line


```c
char s[1001];
gets(s);
puts(s);
```

```pascal
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)
- `gets()` and `readln()` reads and discards EOL, so `s` does not contain EOL
- `puts()` appends an EOL
- In C, remember to allocate one more cell for the null character
- In Pascal, **ansistring** is dynamically allocated

N lines


```c
char s[1001];
int i, n = 10;
for (i = 0; i < n; i++) {
    gets(s);
    puts(s);
}
```

```
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


```c
char s[1001];
while (gets(s) != NULL)
    puts(s);
```

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

- gets returns NULL upon EOF
- **eof** is a function with no arguments

Numbers

Type	C/C++	Pascal
int	scanf("%d", &x) printf("%d", x)	read(x) write(x)
double	scanf("%lf", &x) printf("%f", x)	read(x) write(x)
long long (Linux)	scanf("%lld", &x) printf("%lld", x)	read(x) write(x)
long long (Windows)	scanf("%I64d", &x) printf("%I64d", x)	read(x) write(x)

- Notice the use of %lf in scanf and %f in printf

Space separated numbers

Input

```
3    5  7
```

```c
int x;
while (scanf("%d", &x) == 1)
    printf("%d\n", x);
```

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

- `scanf("%d")` skips all whitespaces and EOLs
- `scanf` returns the number of variables successfully read
- `read` skips only whitespaces

Formatted output


```c
char s[] = "abc";
int d = 3;
double f = -3.1875;
printf("%s\n", s);
printf("%10s\n", s);
printf("%d\n", d);
printf("%10d\n", d);
printf("%10.2f\n", f);
```

```
var
  s : ansistring = 'abc';
  d : longint = 3;
  f : double = -3.1875;
begin
  writeln(s);
  writeln(s:10);
  writeln(d);
  writeln(d:10);
  writeln(f:10:2)
end.
```

Output

```
abc
       abc
3
         3
     -3.19
```

Advanced C formatted output


```
char s[] = "abc";
printf("%s$\n", s);      //normal
printf("%1s$\n", s);     //at least 1 char
printf("%5s$\n", s);     //at least 5 char
printf("%.2s$\n", s);    //at most 2 char from s
printf("%5.2s$\n", s);   //5 char, 2 char from s
printf("%-5.2s$\n", s); //left justified
```

Output

```
abc$
abc$
  abc$
ab$
   ab$
ab   $
```

Advanced C formatted output


```
printf("%5d$\n", 1);  //at least 5 char
printf("%-5d$\n", 1); //left justified
printf("%05d\n", 1);  //at least 5 char, 0 padded
printf("%.5d\n", 1);  //at least 5 digits
printf("%05d\n", -1); //at least 5 char, 0 padded
printf("%.5d\n", -1); //at least 5 digits
```

Output

```
    1$
1    $
00001
00001
-0001
-00001
```

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
Method 1


```c
FILE *fin, *fout;
int x;
fin = fopen("in.txt", "r");
fout = fopen("out.txt", "w");
fscanf(fin, "%d", &x);
fprintf(fout, "%d", x);
fclose(fin);
fclose(fout);
```

```pascal
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.
```

- Notice the different use of I/O functions
- Remember to close the files

File I/O
Method 2


```c
int x;
freopen("in.txt", "r",stdin);
freopen("out.txt", "w", stdout);
scanf("%d", &x);
printf("%d", x);
```

```
var
  x : longint;
begin
  assign(input, 'in.txt');
  reset(input);
  assign(output, 'out.txt');
  rewrite(output);
  read(x);
  write(x)
end.
```

- redirect standard I/O to the files and then use ordinary I/O functions

Outline

Bitwise operation

Operation	C/C++	Pascal
AND	&	and
OR	|	or
XOR	^	xor
NOT	~	not
Shift left	<<	<<
Shift right	>>	>>

AND

- Used to test if a bit is one
  - Test the third bit:

```
  10000100
AND 00000100
------------
  00000100
```

  - Test the fourth bit:

```
  10000100
AND 00001000
------------
  00000000
```

- Used to clear a bit
  - Clear the third bit:

```
  10000100
AND 11111011
------------
  10000000
```

OR

- Used to set a bit to one
- Set the third and fourth bit:

$$\begin{array}{rr} & 10000100 \\ \text{OR} & 00001100 \\ \hline & 10001100 \end{array}$$

XOR

- Used to flip a bit
- Flip the third and fourth bit:

$$\begin{array}{rr} & 10000100 \\ \text{XOR} & 00001100 \\ \hline & 10001000 \end{array}$$

NOT

- Used to flip all bits

$$\begin{array}{ll} \text{NOT} & 10000100 \\ \hline & 01111011 \end{array}$$

Shift left

- A quick way to multiply by a power of 2

$$\begin{array}{rr} & 12 \\ << & 1 \\ \hline & 24 \end{array}$$

Shift right

- A quick way to divide by a power of 2

$$\begin{array}{rr} & 12 \\ >> & 2 \\ \hline & 3 \end{array}$$

- Better not use this on negative numbers
  - -2 $>>$ 1 $=$ -1?
  - -2 $>>$ 1 $=$ 127?

Two's complement on signed integers

Still remember?

C/C++	Pascal
(~x) + 1 == -x	(not x) + 1 = -x

Outline

Type conversion


```
var
  l : longint;
  d : double;
  i : int64;
begin
  l := 100000;
  d := l;
  i := l;

  d := 9007199254740991;
  l := trunc(d); {l = -1}
  i := trunc(d);
end.
```

- When assigning to a different type, precision may lose

Type conversion


```
var
  l : longint;
  d : double;
  i : int64;
begin
  i := 9223372036854775807;
  l := i;
  d := i;
end.
```

What will be the value of d?

A 9007199254740991

B 9223372036854775800

C 9223372036854775807

D infinity

Type conversion


```
var
  l : longint;
  d : double;
  i : int64;
begin
  i := 9223372036854775807;
  l := i;
  d := i;
end.
```

What will be the value of d?

A 9007199254740991

B 9223372036854775800

C 9223372036854775807

D infinity

Number to string


```c
char s[10];
sprintf(s, "%d", 123);
```

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

String to number


```c
int x;
sscanf("123", "%d", &x);
```

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

Outline

Header files for C/C++

- In C, add "`#include<string.h>`"
- In C++, add "`#include<cstring>`"
  - note: `cstring` is different from `string`

String functions

Learning these functions can help you

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

strlen / length


```c
int len = strlen("abc");
```

```
var
  len : longint;
begin
  len := length('abc')
end.
```

- `len` is 3

strlen / length

- Time complexity of `length()`: $O(1)$ on **ansistring**
- Time complexity of `strlen()`:

strlen / length

- Time complexity of `length()`: $O(1)$ on **ansistring**
- Time complexity of `strlen()`: $O(N)$
- Do not:

```
while (i < strlen(s)) {
    ...
}
```

strlen / length

- Time complexity of `length()`: $O(1)$ on **ansistring**
- Time complexity of `strlen()`: $O(N)$
- Do not:

```
while (i < strlen(s)) {
    ...
}
```

- Rather, save the length in a varible first
- Note that `strlen()` returns an unsigned integer, beware of overflow

strcmp / strncmp

`strcmp("abc", "def")` | `'abc' < 'def'`

- Dictionary ordering
- `strcmp(s, t)` < 0 when s $<$ t returns
- `strcmp(s, t)` $== 0$ when s $==$ t returns
- `strcmp(s, t)` > 0 when s $>$ t returns
- `strncmp(s, t, n)` compares at most n characters
- In Pascal, simply use $<$, $>$, $<>$, $=$ etc.

strstr / strchr / pos


```c
char target[] = "abc";
char *p =
  strstr(target, "bc");
char *q =
  strchr(target, 'c');
//p == &target[1]
//q == &target[2]
```

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

- `strstr(target, s)` returns
  - The pointer of the first occurrence of `s` in `target`
  - `NULL` if not found
- `pos(s, target)` returns
  - The index of the first occurrence of `s` in `target`
  - 0 if not found
- Note that **ansistring** is 1-based

strcpy / strncpy / copy


```c
char target1[10];
char target2[10];
char s[] = "abc";
strcpy(target1, s);
strncpy(target2, s + 1, 2);
```

```pascal
var
  target1 : ansistring;
  target2 : ansistring;
  s : ansistring = 'abc';
begin
  target1 := copy(s, 1,
                  length(s));
  target2 := copy(s, 2, 2)
end.
```

- `target1` = "abc"
- `target2` = "ab" (not null-terminated in C/C++)
- `target` and `s` should not overlap
- If `n` $\leq$ `strlen(s)`, `target` may not be null-terminated
- If `n` $>$ `strlen(s)`, `target` will be padded with zeroes
- `strcpy` and `strncpy` return `target`

strcat / concat


```c
char s[10] = "ab";
strcat(strcat(s, "cd"), "ef");
```

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

- Concatenate
- For `strcat`, arguments should not overlap
- `strcat` returns first argument
- `concat` accepts many arguments and creates a new string

insert


```c
char s[10] = "ad";
char t[] = "bc";
int tlen = strlen(t);
memmove(s + 1 + tlen,
        s + 1,
        strlen(s + 1));
memcpy(s + 1, t, tlen);
```

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

- s is "abcd"
- memmove allows memory overlap, while memcpy doesn't

delete


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

- s is now 'abc'

Space separated words on a line

Excerpt from S184 Bogo Translate

Solutions from contestants:

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

Input

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

Space separated words on a line

Solution using strtok:

```
char line[1000];
gets(line);
char *word[100];
char *p = strtok(line, " ");
int i = 0;
while (p != NULL) {
  word[i] = p;
  i++;
  p = strtok(NULL, " ");
}
```

Space separated words on a line

Solution using stringstream:

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

Practice

HKOJ01013 Internet Usage Bills
HKOJ01001 TeX Processing