An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 5

Internal Representation of Data Types

Bits and Bytes

Based on material developed by Prof. Abhiram G. Ranade

So far

- Control flow: sequential, `if-else` conditions, loops
- Variables, types (`int`, `char`, `bool`, `...`), operators, expressions
- Assignment, incrementing/decrementing

Today

- Bit-level representation of data
  - `bool`, `char`, `int`, `float`, `double`, ...
- Conversions across types
- Bit-level operations

Suggested reading:
Chapter 3 in the textbook

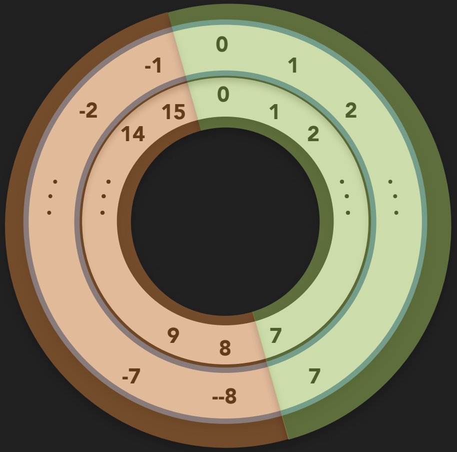
Reminder: Quiz 1 on Aug 21
Covers Lectures 1 through 5

bool

- Simplest data type: Only two values. Internally a *byte* (8 bits)
- Other types can be converted to bool, implicitly or by explicitly casting
  - Expression *expr* can be explicitly cast to `bool` by writing `bool(`*expr*`)`
  - Implicitly: By using it where a bool expression is expected:
    E.g., `bool x =` *expr*, or `if(`*expr*`)`
- Zero is converted to `false`, and non-zero values to `true`
  - E.g., `while(1)` or `while(2)` becomes `while(true)`. `if(n)` becomes `if(n!=0)`.
- Conversely, `bool` values can be converted to `char`, `int`, etc.
  - `false` converted to `0` and `true` to 1
  - Arithmetic on `bool` first promotes it to an integer type
- `int x; bool y; x = y = 2;  // what will x have?`

Binary Representation

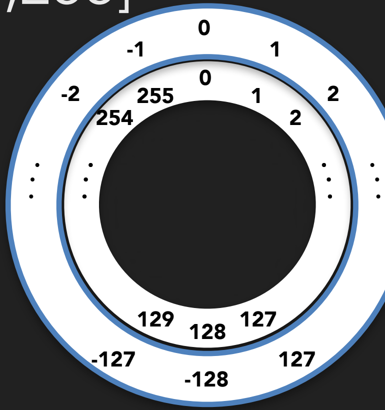
- Using n bits, can represent 2^n different numbers
  - e.g., [0,7] using 3 bits, and [0,15] using 4 bits
- We can use them to represent negative numbers too
- A standard format (called "two's complement"):
  - n bits to represent integers in the range $[-2^{n-1}, 2^{n-1}-1]$



0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
-8	8	1000
-7	9	1001
-6	10	1010
-5	11	1011
-4	12	1100
-3	13	1101
-2	14	1110
-1	15	1111

Characters

- In C++, `char` data type corresponds to a single byte, i.e., 8 bits
- `unsigned char` works like an integer in the range [0,255]
  - `00000000` is 0, and `11111111` is 255
- `signed char` works like an integer ∊ [-128,127]
  - `00000000` is 0, and `01111111` is 127.
    `10000000` is -128, and `11111111` is -1.



- Operations like + , - , * , / work like for integers, and <u>the result is an integer</u> (can be converted back to `char`, if desired)
- Converting (implicitly or by casting) integers to `char` done mod 256
- Input/Output (keyboard/printing) of `char` uses ASCII code

Characters

128-character, 7-bit ASCII code [designed in 1963, with a few revisions later]

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

`null :` 00000000 (0) `space:` 00100000 (32) `delete:` 01111111 (127)

`'0' :` 00110000 (48) `'A':` 01000001 (65) `'a':` 01100001 (97)

As `char` or `unsigned char`, the first bit is 0 for all valid ASCII characters

Bitwise Operations

- In C++, can carry out bit-level manipulations
- Recall boolean operations `AND`, `OR`, `XOR`, `NOT`
- Can be interpreted as operations on bits, with 0 and 1 interpreted as false and true, respectively
- E.g., `1 OR 0 → 1.  1 AND 0 → 0.  1 XOR 0 → 1. 1 XOR 1 → 0. NOT 0 → 1.`
- One can apply such an operation bit-wise on bytes
- E.g., `00001111 AND 10101010 → 00001010`
- E.g., `00001111 OR  10101010 → 10101111`
- E.g., `00001111 XOR 10101010 → 10100101`
- E.g., `NOT 00001111 → 11110000`

In C++

```
a & b
a | b
a ^ b
~a
```

Example: Operations on `char`

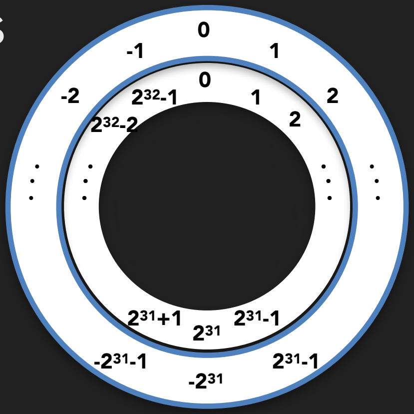
Demo

```
// toggle between uppercase and lowercase (non-alpha unchanged)
cin >> noskipws;
char c;

for (cin >> c; c != '\n' ; cin >> c) {
    if (c >= 'a' && c <= 'z')          c = c - 'a' + 'A';
    else if (c >= 'A' && c <= 'Z')  c = c - 'A' + 'a';
    cout << c;
}

// alternately, exploiting more ASCII specifics
const char casebit = 32; // casebit=00100000 switches the case
for (cin >> c; c != '\n' ; cin >> c) {
    if ((c|casebit) >= 'a' && (c|casebit) <= 'z')
        c ^= casebit ; // toggle the case bit
    cout << c;
}
```

What does the following expression give?
`c & (~casebit)`

`int` and Friends

- `int` and `unsigned int` correspond to 32 bits (4 bytes)
  (except in some very old implementations on Windows, where it was 16 bits)
- `unsigned int` takes values in the range $[0, 2^{32} - 1]$ ($\approx$ [0, 4 billion])
- `int` takes values in the range $[-2^{31}, 2^{31}-1]$ ($\approx$ [-2 billion, 2 billion])
- `short` and `unsigned short` correspond to 16 bits
  - Ranges [-32K,+32K-1] and [0,64K-1], resp.
    K here stands for Kilo: 2^{10} = 1024 (roughly 1000)



- `long long` and `unsigned long long` correspond to 64 bits
- There is also a type (`unsigned`) `long`, but in many implementations it is the same as (`unsigned`) `int`

`int` and Friends: Literal Formats

- Integer *literal* : an integer constant as written in a program
    - E.g., int a = `32`, b = `+21`, c = `-1` ;
- Typically integer literals are in decimal. But literals can be binary, octal (base 8) or hexadecimal (base 16) : Start with `0b`, `0` or `0x` resp.
    - E.g., `0b11010` == 26, `032` == 26, `0x1a` == 26
    - In hexadecimal, can write a byte as 2 digits: `00011010` is `0x1a`
- Suffix U and/or LL at the end to indicate unsigned and/or long long
- Note: `cin` reads decimal integers (e.g., leading 0 is ignored)

`float` and Friends

- `float` stands for floating point number
  - E.g. in decimal: $7.9225 \times 10^2 = 792.25$ has 5 digits of ***precision***, and its ***scale*** (given by the exponent 2) is such that is is between 100 and 999
  - E.g. in binary: $1.10001100001 \times 2^9 = 1100011000.01$ has 12 bits of precision and its scale is such that it is between 512 and 1023
  - E.g. in decimal: $1.875 \times 10^{-1} = .1875$ in binary: $1.1 \times 2^{-3} = .0011$

- By changing the exponent, the "point" floats left or right

- While representing a real number as a floating point number, will use some bits for precision, and some for scale (both signed)

  - Only finitely many real numbers have an exact representation

`float` and Friends

- `float` uses 32 bits
  - 1 bit for sign. Precision of 24 bits (23 bits stored, a leading 1 is implicit). Scale stored using 8-bits: 2^{-126} to 2^{127} (two values of the exponent are used for indicating special values).
  - Special values:
    - 0 (actually, ±0). Since implicit leading 1 won't allow representing 0.
    - Subnormal numbers (no implicit leading 1, with exponent 2^{-126})
    - ± infinity (e.g., result of dividing a non-0 number by 0)
    - "Not a Number" (NaN)
- `double` (for double precision floating point number) uses 64 bits
  - 1 bit for sign, 53 bits for precision (one implicit), 11 bits for scale.
- `long double` : may be 64, 80 or 128 bits (platform specific)

`float` and Friends: Literal Formats

- Format for floating point literals (numbers appearing in the programs) and also as used by cin/cout
  - We write <u>*num*</u> E <u>*exp*</u> (with no spaces) to mean *num* × 10^{exp}, where <u>*num*</u> can optionally have a decimal point.
  - Note: Exponent is for 10. Also, the number is in decimal.
    (There is a format allowing numbers to be specified in hexadecimal.)
  - Examples: `314E-2`, `-.01`, `1.` (E part is optional if `.` present) , `6.02214076e23` (can use E or e), `+1E+1` (+ signs are optional)
- By default, the literal is taken as a `double`. Suffix F to force `float`.

Example: Precision Issues

Demo

- Floating point arithmetic has a lot of subtleties

```
// Order of operations matters
float f = 2e7; // 20 million > 2^24
cout << 1 + f - f << endl;  // gives 0 instead of 1
cout << f - f + 1 << endl;  // gives 1 as expected

// for fractions, internal representation being binary matters
cout << 1 + 0.01F - 1 << endl;  // not equal to 0.01!
cout << 1 + 0.0078125F - 1 << endl;  // is equal to 0.0078125 !
```

Working with Real Numbers

- For the sake of better precision, use `double` instead of `float`
  - Using `double` can be a little less efficient in large applications: more memory needed, and (hence) slower
- When comparing, allow a "tolerance" (and be prepared for false positives)
  - E.g., instead of `a == b`, use `abs(a-b) <= epsilon`
  - E.g., instead of `a >= b`, use `(a-b) >= -epsilon`
  - The choice of the tolerance value will be application dependent!
    - Further, epsilon could be a function of `a,b`:
      e.g., `epsilon = max(a,b)*delta` (where delta is application dependent)

Exercise

- Inspect the sample program explain.cpp accompanying this lecture.

  For each part, try to find out why the program behaves the way it does.

- Simulate a projectile's trajectory, given initial x and y velocity. A sample solution is provided. Modify it to add a second projectile, and detect near collisions. (Report the same collision event only once.)