Data Types

- **char** (character)

  8 bits (stored in one byte in memory)

  unsigned:

  $0 \leq \text{char} \leq 2^8 - 1$

  $00000000 \leq \text{char} \leq 11111111$

  Overflow at 255 (255 + 1 = 0)

  Underflow at 0 (0 – 1 = 255)

  signed (if supported in the implementation):

  $-2^7 \leq \text{char} \leq 2^7 - 1$

  $10000000 \leq \text{char} \leq 01111111$

  Overflow at 127 (127 + 1 = -128)

  Underflow at –128 (-128 – 1 = 127)

Data Types

- **int** (integer <u>on our machines</u>)

  32 bits (stored in four sequential bytes in memory)

  unsigned:

  $0 \le$ char $\le 2^{32} - 1$

  0x00000000 $\le$ char $\le$ 0xffffffff

  Overflow at 4294967295 (4294967295 + 1 = 0)

  Underflow at 0 (0 – 1 = 4294967295)

  signed:

  $-2^{31} \le$ char $\le 2^{31}-1$

  0x80000000 $\le$ char $\le$ 0x7fffffff

  Overflow at 2147483647 (2147483647 + 1 = –2147483648)

  Underflow at –2147483648 (-2147483648 – 1 = 2147483647)

Data Types

- **short int** (short integer <u>on our machines</u>)

  16 bits (stored in two sequential bytes in memory)

  unsigned:

  $0 \leq \text{char} \leq 2^{16} - 1$

  $0x0000 \leq \text{char} \leq 0xffff$

Overflow at 65535	(65535 + 1 = 0)
Underflow at 0	(0 – 1 = 65535)

  signed:

  $-2^{15} \leq \text{char} \leq 2^{15}-1$

  $0x8000 \leq \text{char} \leq 0x7fff$

Overflow at 32767	(32767 + 1 = –32768)
Underflow at –32768	(-32768 – 1 = 32767)

Data Types

- **long int** (long integer <u>on our machines</u>, same as int)

  32 bits (stored in four sequential bytes in memory)

  unsigned:

  $0 \leq char \leq 2^{32} - 1$

  0x0000 0000 ≤ char ≤ 0xffff ffff

  Overflow at 4294967295 (4294967295 + 1 = 0)

  Underflow at 0 (0 – 1 = 4294967295)

  signed:

  $-2^{31} \leq char \leq 2^{31}-1$

  0x8000 0000 ≤ char ≤ 0x7fff ffff

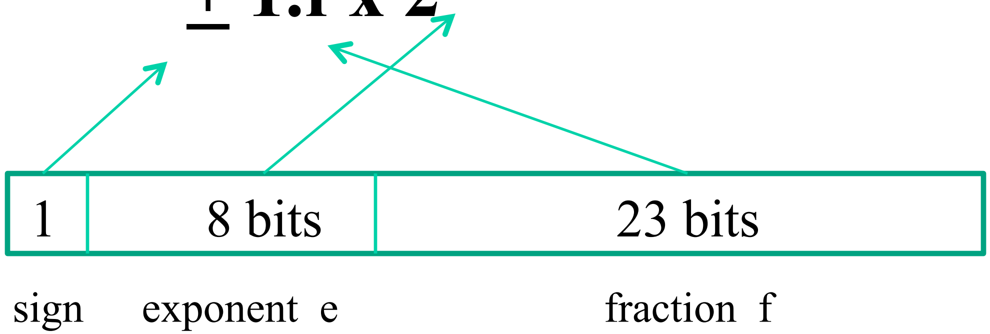
  Overflow at 2147483647 (2147483647 + 1 = –2147483648)

  Underflow at –2147483648 (-2147483648 – 1 = 2147483647)

Data Types

- **float**
  - 32-bits (stored in four sequential bytes in memory)
  - based on the IEEE 754 floating point standard

$$\pm\ \mathbf{1.f \times 2^{e}}$$



Data Types

- **float**, **double**
  - "double" is a "float" with more precision
  - Implementation is machine dependent:
    - for our machine: float is 4 bytes, double 8 bytes
- Without a co-processor to do floating point computation, it is computationally expensive in software.
  - Not often used in real time, embedded systems.
  - A cost versus performance tradeoff!

Numbering Systems

- Binary
- Octal (Octal Constant is written 0dd…)

OCTAL	BINARY	OCTAL	BINARY
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

Note: Can’t write a decimal value with a leading 0 digit – will be interpreted as octal

Numbering Systems

- Hex (Hex Constant is written 0xdd…)

HEX	BIN.	**HEX**	BIN.	**HEX**	BIN.	**HEX**	BIN.
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

- DO NOT convert between binary and Hex or Octal by converting to decimal and back!
  - Group binary digits by 3 (octal) or 4 (hex) to convert

Examples of the Number System

Decimal Octal Hex

31 ---------> 037 ------------> 0x1f

128 --------->0200 -----------> 0x80

Numbering Systems

- char Data Type Constants

‘a’	int value in ASCII code for letter ‘a’
‘0’	int value in ASCII code for number 0
‘\b’	int value in ASCII code for backspace
‘\ooo’	octal value 000-377 (0-255 decimal)
‘\xhh’	hex value 0x00-0xff (0-255 decimal)

- Examples

‘a’ = 0x61
‘0’ = 0x30
‘\127’ = 0x57
‘\x2b’ = 0x2b

Numbering Systems

- Other Data Type Constants

1234	int
1234L	long int
1234UL	unsigned long int
1234.	double (because of decimal point)
1234.4F	float (because of the suffix)
1234e-2	double (because of exponent)

Converting Decimal to Binary

- Divide the decimal number successively by 2 and write down the remainder in binary form:

e.g. 117_{10}

	117		1	LSB	(after divide by 2, remainder =1)
	58	odd	0		(after divide by 2, remainder =0)
even	29		1		(after divide by 2, remainder =1)
	14		0		(after divide by 2, remainder =0)
	7		1		(after divide by 2, remainder =1)
	3		1		(after divide by 2, remainder =1)
	1		1		(after divide by 2, remainder =1)
	0		0	MSB	

- Read UP and add any leading 0's: 01110101

Converting Decimal to Hex

- **Method 1**: Convert decimal to binary and group the binary digits in groups of 4

  e.g. $117_{10} \rightarrow \underbrace{0111}\,\underbrace{0101}_2 \rightarrow 75_{16}$

- **Method 2**: Divide the decimal number successively by 16 and write down the remainder in hex form:

117	5	LSB (after divide by 16, remainder =5)
7	7	MSB (after divide by 16, remainder =7)

  – Read UP and add any leading 0's: 0x75

Converting Binary to Decimal

- Treat each bit position n that contains a one as adding 2^n to the value. Ignore 0's.

Bit 0	LSB	1	1	(= 2^0)
Bit 1		0	0	
Bit 2		1	4	(= 2^2)
Bit 3		1	8	(= 2^3)
Bit 4		0	0	
Bit 5		1	32	(= 2^5)
Bit 6		0	0	
Bit 7	MSB	0	0	
Total			45	

Converting Hex to Decimal

- Treat each digit n as adding 16^n to the value.

Digit 0	LSB	0	0	
Digit 1		2	32	(= 2 * 16^1)
Digit 2		b	2816	(= 11 * 16^2)
Digit 3		0	0	
Digit 4		0	0	
Digit 5		1	1048576	(= 1 * 16^5)
Digit 6		0	0	
Digit 7	MSB	0	0	
Total			1051424	

Base for **Integer** Constants

- Designating the base for an integer constant
- If constant begins with either:

0x	It is Hex with a-f as Hex digits
0X	It is Hex with A-F as Hex digits

- Otherwise, if constant begins with

0	It is Octal

- Otherwise, it is decimal

Base for **Character** Constants

- If constant begins with either:

  ‘\x   It is Hex with 0-9 and a-f as Hex digits

  ‘\X   It is Hex with 0-9 and A-F as Hex digits

- Otherwise, if constant begins with

  ‘\0   It is Octal

- Otherwise, it is the ASCII code for a character

  ‘a’

Signed (Default) Behavior

- By default, int and char variables are **signed**
- Careful mixing modes when initializing variables!

int i;	(signed behavior is default)
char c;	(signed behavior is default)
i = 0xaa;	(== 000000aa) as intended
i = '\xaa';	(== ffff ffaa) sign extends!
c = '\xaa';	(== aa) as intended
i = c;	(==ffff ffaa) sign extends!

Unsigned Behavior

unsigned int i; (must specify unsigned if wanted)

unsigned char c; (must specify unsigned if wanted)

i = 0xaa; (== 000000aa) as intended

i = '\xaa'; (== ffffffaa) char sign extends!

c = '\xaa'; (== aa) as intended

i = c; (==0000 00aa) char sign not extended!

Example to illustrate signed/ unsigned types

```
void copy_characters(void){
  char ch;
  while ((ch =getchar()) != EOF)
    putchar(ch);
}
```

If getchar() == EOF, is this true?

What happens if ch is defined as unsigned char? Is this true?

No

Yes, because "int getchar(void)"

Changing the declaration of ch into int ch; makes it work.

1's Complement

- Flip the value of each bit

  All zeroes become one, All ones become zero

  ~ '\xaa' == '\x55'

  ~10101010 == 01010101

- Number **anded** with its 1's complement = 0

```
  10101010
& 01010101
  --------
  00000000
```

2's Complement

- Flip the value of each bit and add 1
- It creates the negative of the data value
  - '\x55' == '\xab'
  - 01010101 == 10101011
- Number **added** to its 2's complement = 0

$$
\begin{array}{rr}
 & 01010101 \\
+ & \underline{10101011} \\
(1) & 00000000
\end{array}
$$

(carry out of MSB is dropped)

Two Special Case Values

- char 0 (or zero of any length)

$$\begin{array}{lr} -00000000 & = 11111111 \\ & \underline{+\qquad\qquad 1} \\ & = 00000000 \end{array}$$

- char -2^7 (or -2^{n-1} for any length = n)

$$\begin{array}{lr} -10000000 & = 01111111 \\ & \underline{+\qquad\qquad 1} \\ & = 10000000 \end{array}$$

Bit Manipulation

- Bitwise Operators:
  - `~` one's complement (unary not)
  - `&` and
  - `|` or
  - `^` xor (exclusive or)
  - `<<` left shift
  - `>>` right shift

Binary Logic Tables

NOT	
0	1
1	0

AND	0	1
0	0	0
1	0	1

OR	0	1
0	0	1
1	1	1

XOR	0	1
0	0	1
1	1	0

ADD	0	1
0	0	1
1	1	0 Carry 1

Operands

Results

Bit Manipulation

```
unsigned char n = '\xa6';
   n          10100110

   ~n         01011001  (1s complement: flip bits)

   n | '\x65'  10100110  turn on bit in result if
            |  01100101  on in either operand
               --------
               11100111
```

Bit Manipulations

```
n & '\x65'   10100110   turn on bit in result if
          |  01100101   on in both operands
             --------
             00100100
n ^ '\x65'   10100110   turn on bit in result if
          |  01100101   on in exactly 1 operand
             --------
             11000011
```

Bit Manipulations

n = ‘\x18’; 00011000 (Remember n is **unsigned**)

n << 1 00110000 shift 1 to left (like times 2)

n << 2 01100000 shift 2 to left (like times 4)

n << 4 10000000 shift 4 to left

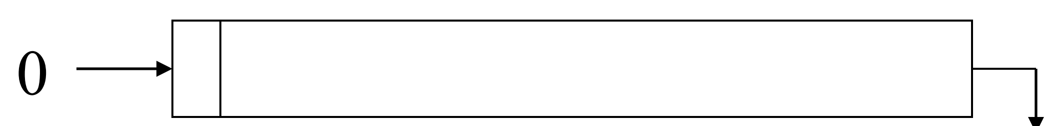
(bits disappear off left end)

n >> 2 00000110 shift 2 to right (like / 4)

n >> 4 00000001 shift 4 to right

(bits disappear off right end)

Unsigned Right Shift



Bit Manipulations

- “>>” result may be different if n is signed!
  - If value of n has sign bit = 0, works same as last slide
  - If value of n has sign bit = 1, works differently!!

char n =	‘\xa5’ ;	(default is signed)
n	10100101	(sign bit is set)
n >>2	11101001	(bring in 1’s from left)

Bit Manipulations

- For signed variable, negative value shifted right by 2 or 4 is still a negative value

  '\xa5' =10100101

  '\xa5' >> 2 = 11101001 = '\xe9'

- Same result as divide by 4 ($2^2 = 4$)
- But, this is not true on all machines

Bit Manipulations

- When dividing a negative value
  - Different rounding rules than for positive value
  - Remainder must be negative - not positive

- Note that (-1)/2 = -1

- Note that -(1/2) = 0

Forcing Groups of Bits Off

- Given char n, how to turn off all bits except the least significant 5 bits:

n = n & '\x1f'

n = '\xa5'		10100101	
n & '\x1f'		10100101	
	&	00011111	turn off all bits
		00000101	except bottom 5

Forcing Groups of Bits On

- Given n, how to turn on the MS two bits (if already on, leave on).

  n = n | '\xc0'

n = '\xa5'

n | '\xc0':   10100101

| 11000000 turn on MS 2 bits

11100101

String Constants

- String constant: “I am a string.”
  - An array (a pointer to a string) of char values somewhere ending with **NUL** = '\0’
  - "0" is not same as '0'. The value "0" can't be used in an expression - only in arguments to functions like printf().
- Also have a library with string functions

  #include <string.h>

  With these definitions, can use: len = strlen(msg);
  where msg is string in a string array

Enumeration Symbolic Constants

- **enum** boolean {FALSE, TRUE};
  - Enumerated names assigned values starting from 0
    - FALSE = 0
    - TRUE   = 1
- Now can declare a variable of type enum boolean:

  enum boolean x;

  x = FALSE;
- Just a shorthand for creating symbolic constants instead of with #define statements
- Storage requirement is the same as int

Enumeration Symbolic Constants

- If you define months as enum type

```
enum months {ERR, JAN, FEB, MAR,
             APR, MAY, JUN, JUL,
             AUG, SEP, OCT, NOV,
             DEC};
```

- Debugger might print value 2 as FEB

Code Example of the enum type

```
int main()
{
    enum month {ERR, JAN, FEB, MAR, APR, MAY, JUN,
                        JUL, AUG, SEP, OCT, NOV, DEC};
    enum month this_month;

     this_month = FEB;
     …
     …
}
```

const

- "const" declaration is like "final" in Java
  - warns compiler that value shouldn't change

    const char msg[ ] = "Warning: . . .";
  - Commonly used for function arguments

    int copy(char to[ ], **const** char from[ ]);

- If logic of the copy function attempts to modify the “from” string, compiler will give a warning

Operators

- Arithmetic Operators:

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo (Remainder after division)
	e.g. 5%7 = 5 7%7 = 0 10%7 = 3

- Logical Operators:

&&	logical and
||	logical or
!	Not

Relations / Comparisons

- We call a comparison between two arithmetic expressions a "relation"

  ae1 <= ae2 (Comparisons: <, <=, = =, !=, >=, >)

- A relation is evaluated as true or false (1 or 0) based on values of given arithmetic expressions

- if (i < lim-1 = = j < k)
  - What's it mean?

- Instead of c != EOF, could write !(c = = EOF)