13: C Review

C (Java, C++, Python) provide the following bitwise operations:

- `&` bitwise and
- `|` bitwise or
- `^` bitwise xor, also called exclusive or
- `<<` left shift
- `>>` right shift
- `~` one's complement

Notation: \((z)_i\) denotes the \(i\)-th bit in \(z\) (from right)
13: **Bitwise &**

- if \((x)_i == 1\) and \((y)_i == 1\) then \((x & y)_i == 1\)
- otherwise \((x & y)_i == 0\)

Bitwise *and* is often used to *clear* bits or bytes. For example:

- \(x \& 0xff\) clears all bytes but the low-order byte:

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(&amp;)</th>
<th>(0xff)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td></td>
<td>...0</td>
</tr>
<tr>
<td>(x &amp; )</td>
<td>0...0</td>
<td>0...0</td>
<td>1111111</td>
</tr>
<tr>
<td></td>
<td>(&amp;)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

low-order
13: **Bitwise** ^

- if $(x)_i == (y)_i$ then $(x ^ y)_i == 0$
- otherwise $(x ^ y)_i == 1$

- Bitwise xor clears those bits that are the same in both arguments, and sets the other bits.
- Can be used to test if two words are equal, for example $x ^ y$
- returns 0 if $x$ and $y$ are equal.
Properties of XOR (^)

- XOR (like negation) has the property that it is *invertible*, and it is its own inverse.

- i.e. $x \ XOR \ y \ XOR \ y = x$

- e.g. $x=1, y=1$: $1 \ xor \ 1 = 0$, $0 \ xor \ 1 = 1$

- For entire characters, this is a common simple encryption method.
  
  - Let $x \ ^\wedge \ y = z$, encrypted $x$ (z can be transmitted)
  
  - If the receiver knows $y$, then $z \ ^\wedge \ y = x$
13: Left Shift

- Left shift: \( i \ll j \)

- The resulting word will have all bits shifted to the left by \( j \) positions;

- for every bit that is shifted off the left end of the word, there is a zero bit added at the right end.

- \( x \ll 1 \) is equivalent to \( x *= 2 \)

- \( x \ll 2 \) is equivalent to \( x *= 4 \).
13: Examples of bitwise operations

- `getBit()` returns the $i$-th bit in $w$, using a bitwise `and` of its first argument and `MASK(j)`:  

```c
#define MASK(j)   (1 << j)

int getBit(int w, unsigned j) {
    return ((w & MASK(j)) == 0) ? 0 : 1;
}
```
make

a specific software tool

Reference book (not required):

*Managing Projects with make*
Andrew Oram and Steve Talbott
O'Reilly & Associates
Why make?

- When a project contains many source files, it can be very time consuming to compile all of the source files & error prone.
- We would like to re-compile only those files which have changed.
- make is a utility which allows us to specify dependencies, and to rebuild only the necessary files according to the dependencies and modification times.
Makefile

"in order to use make, we place all of our macro definitions, dependencies, commands, and targets into a file which must be called Makefile"

"we then run make with a target (default is all)"

make
make all
make clean
make install
Simple Targets

we can define several targets in a makefile. We simply list the target name, followed by any dependencies (if any):

```
all: foo.c bar.c

gcc -o foo foo.c bar.c -lm
extract_comments < foo.c > foo.doc

foo: foo.o

gcc -o foo foo.o

clean:

/bin/rm -f ${OBJS}
```
Macros

- macros are specified in make as follows:
  
  name = text string

- macro expansion:

  $(name) ${name}

- example:

  SRC=foo.c

  ${SRC}
Common Macros

SRCS=foo.c bar.c
CFLAGS=--Wall -ansi
LDFLAGS=--lm -lmylib
INCDIR=--I/home/ericb/include
LIBDIR=--L/home/ericb/lib

" example command in make:

gcc ${CFLAGS} ${INCDIR} -o foo ${SRCS}\${LIBDIR} ${LDFLAGS}
Targets

we can define several targets in a makefile. We simply list the target name, followed by any dependencies (if any):

all: ${OBJ}
    ${CC} -o foo ${OBJ} ${LIBDIR} ${LDFLAGS}

foo: ${OBJ}
    ${CC} -o foo ${OBJ} ${LIBDIR} ${LDFLAGS}

clean:
    /bin/rm -f ${OBJ}

install: foo
    /bin/cp -f foo /usr/local/bin
Macro String Substitution

"make" has a powerful string substitution operator for macros:

SRCS = defs.c redraw.c calc.c

OBJJS = ${SRCS:.c=.o}

same as:

OBJJS = defs.o redraw.o calc.o
Suffix Rules

suffix rules tell make how files are inter-dependent:

`.c.o:

`{CC} `{CFLAGS} `{INCDIR} -c `<

the above tells make how to create a ".o" file from a ".c" file.

`< is set to the current dependency

recall that `-c` to `gcc` means to compile only, not to link (i.e., to produce a `.o` file)
SRCS=foo.c bar.c  
OBJS=foo.o bar.o barbar.o  
CFLAGS=-Wall -ansi  
LDFLAGS=-lm -lmylib  
INCDIR=-I/home/ericb/include  
LIBDIR=-L/home/ericb/lib  

all: ${OBJJS}  
    ${CC} -o foo ${OBJJS} ${LIBDIR} ${LDFLAGS}  
foo: ${OBJJS}  
    ${CC} -o foo ${OBJJS} ${LIBDIR} ${LDFLAGS}  
clean:  
    /bin/rm -f ${OBJJS}  
install: foo  
    /bin/cp -f foo /usr/local/bin  

bar.c: math.h fooie  
    do magic stuff  

.c.o:  
    ${CC} ${CFLAGS} ${INCDIR} -c <$
makedepend

- For large projects, it is hard to generate dependencies on .h files and keep them up to date.
- When a .h file has changed, we would like all the .c files which included it to be recompiled.
- To automatically append dependency rules to the end of your makefile, run makedepend and tell it which .c files to inspect:

  makedepend *.c