Functions, For loops, Binary Arithmetic
EOP Internships

- VISITOR ANNOUNCEMENT
HW9: Creativity in Processing

- In a couple of weeks when you make a web page and put your creativity programs on there, some people will want to show what they did to the class?

- Another chance to be creative: Last two weeks of term. Get a pair programming partner again!
Announcements

- It is your responsibility to understand how Ecommons works. “My dog ate my homework”
- Midterm: People did well. Get them back next week.
- Makeup for those with a doctor’s note: Tuesday Feb 21st in my office after class E2 267.
- Today: Functions in Processing. Binary Arithmetic
- Next Tuesday: Guest Lecture on Human Computer Interaction by Prof. Sri Kurniawan
**Sudoku GUI**

The Sudoku board is a 9×9 array of cells. Alternating 3×3 cells are colored gray and strong lines separate all of the 3×3 cells, as follows. (More information at [http://www.websudoku.com/](http://www.websudoku.com/).)
Instructions Formed of Simpler Instructions

- Check out this screen shot of the Lightbot
- It is partway through an instruction ... its beacon is lit, but not the tile
- To a programmer the instruction is monolithic (one thing)
- To an agent each instruction is a series of steps

An Instruction abstracts those steps
Abstraction

- The word “abstraction” is used a lot in computing
- Remember: it was one of the 7 big ideas
- Abstraction is a way to understand and solve problems
- As a general definition, abstraction eliminates details to focus on essential properties
- The instruction example just given illustrates functional abstraction meaning that we have given a name to a series of operations that perform a coherent (and to us meaningful) activity; the name is the instruction, the series of operations are the bot’s actions to implement it.
Abstracting

- Collecting the operations together and giving them a name is *functional abstraction*
  - The group of operations perform some function but we ignore all of the details
  - Giving it a name is *functional* abstraction
  - This is AMAZINGLY powerful
  - What makes it powerful, is we can forget about the operations and think only about the function they do; more about this later

- Let’s do some functional abstraction
Example: Abstraction in Everyday life

Get Dressed

Dress Bottom Half

Put on Sox

Put on Pants

Dress Top Half

Put on Shirt
Functions abstract by packaging Computation

- $F_1(\ )$ packages actions: E.G. “process a riser”
The Function Becomes A Concept

- Because $F_1(\cdot)$ “processes a riser,” I think of the programming task as

$$
\begin{array}{c}
\text{Process a riser} & F_1 \\
\text{Move to next riser} \\
\text{Process a riser} & F_1 \\
\text{Move to next riser} \\
\text{Process a riser} & F_1 \\
\end{array}
$$

- With $F_1(\cdot)$ as a concept, I simplify the programming to just 5 steps rather than 21

- It also suggests another concept:
  - Move_to_next_riser( )

2/16/12 © 2010 Larry Snyder, CSE
Homework 10: Functions

- void cell(int x, int y, int s, color tinto)
- void triple(int x, int y, int s, color tinto)
- Also functions for
  - block()
  - row()
  - cellarray()
Important Processing Concepts (some review)
Processing Review

- The basic features of Processing
- Also Check Chapter 4 in the Reas & Fry book.
- Remember the Processing Reference Page: http://www.processing.org/reference/
Review: List of Items

- Variables and Declarations
- Assignments
- Expressions
- Repetition (looping) or For-statements
- Tests or If-statements
- Writing programs
Variables

- Facts about variables are ...
  - Variables “contain” their values, and they can be changed using assignment
  - Variables have a data type such as `int`, `float`, `color`, etc. which is the kind of data they contain

- Rules about variables are ...
  - Variables can be any string of letters, numbers or underscores (`_`) starting with a letter; case-sensitive
  - Variables must be declared; declarations at the top of the program or at the start of a function
  - Variables can be initialized in a declaration
Variables, the Picture

- **Facts**
  - “Contain their value”: grade_point: 3.8
  - “Assign to change: grade_point = 3.9;”
  - “Variables have data type”:

    ```
    color colorBlinky = color(255,0,0);
    ```

- **Rules**
  - “Any string”,
    - should mean something: ColorBlinky
    - theyRuseful_4_U_despite_their_length
  - “Declare vars”: int score; float gpa; color purple;
  - “Initializing is OK”: int score=0; float gpa=4.0; color purple=color(128, 0, 128);
Assignments

- Facts about assignment:
  - Form is always `<variable> = <expression>`
  - Information moves from right to left
  - The `<expression>` is computed first, then the variable is changed, so `x=x+1` is sensible
  - To exchange values in two variables takes 3 stmts

- Rules about assignment:
  - All assignment statements end with a semicolon
Assignments, The Picture

- **Facts**
  - “Form”: grade_point=3.9; yellow=color(255,255,0);
  - \(3.9 = \text{grade\_point}\) is ILLEGAL
  - “Info moves right to left”: \(x = 4.0\);
  - “Compute <expression> first”: \(x = x + 1\);
  - “Exchanging values of \(x, y\) takes 3 statements”. Need to hold onto the value of \(x\) somewhere.

\[
\begin{align*}
\text{temp} &= x; \\
x &= y; \\
y &= \text{temp};
\end{align*}
\]
Expressions

- Facts about expressions
  - Expressions are formulas using OPERATORS:
    - `+ - * / % || ! && == < <= >= > !=`
  - Plus, minus, times, divide, mod, or, not, and, equals,
  - Operators can only be used with certain data types and their result is a certain data type
  - Can be smart to put in extra parentheses is OK
  - Self documenting code (variable names, parentheses, comments)

- Rules about expressions
  - Expressions can usually go anywhere that a variable can go
Expressions, the Picture

- **Facts**
  - Expressions are formulas: \( a+b \) points*\( wgt \)
    (year\%4 == 0) \( 7 != 4 \) \( (age>12) \) \&\&(age<20)
  - **Data types:**
    - \(+\, -\, *\, /\, \%\, <\, <=\, =>\, >\) want **numbers**;
    - \&\& \! \|| want **logical** (Boolean) values
    - \(==\) and \(!=\) want arguments to be **the same type**
  - “Parentheses are good”: \((a \ast b) + c\) is the same as \(a\ast b+c\), but easier to read

- **Rules:** Expressions replace vars”: \( \text{rect}(x, y, x+4, y+4)\);
For loops (Repetition)

- Repeating commands is powerful:
  - Lightbot 2.0 used recursion, a function calling itself
  - Symbolic Lightbot prefixed a number, 2:Step
- Processing uses a for loop:

```java
void setup() {
  size(500,200);
  background(0);
  noStroke();
  smooth();
  fill(255);
  for (int i=0; i < 16; i++) {
    ellipse(100+25*i, 100, 15, 15);
  }
}
```
A **for loop** has several parts, all required ...

```plaintext
for (int i=0; i < 16; i++) {
// <stuff to be repeated>
ellipse(100+25*i, 100, 15, 15);
}
```

The result of this statement is 16 copies of the stuff to be repeated. 16 Pacman pills
Or how about a bullseye?

Note the *loop variable* must be declared ... but could do it in loop itself like we did for pacman pills:

```java
for (int i = 0; ...`
If statements

- The instructions of a program are executed sequentially, one after another … sometimes we want to skip some.
- We’ve used **If in several HWs.**
- **If** also has a required form
  ```
  if (year%4 == 0) {
      <stuff to do if condition true>;
  }
  ```

  ```
  if (chosen_tint != red) {
      fill(chosen_tint);
  }
  ```
If Statements: the Picture

- An **If**-statement has a standard form

```java
if (bmi > 18.5 && bmi <= 24.9) {
    fill(0, 255, 0);
}
```

The result is that if bmi is in range (more than 18.5 and less than or equal to 24.9) **then** the fill color is green (indicating OK)
Else Statement

- What happens if we want to do something else if the condition is false? What else? **else**!

- The `else` statement must follow an `if` ...

```java
if (year % 4 == 0) {
    <stuff to do if condition true>; //Then Clause
} else {
    <stuff to do if condition false>; //Else Clause
}
```
Else, the Picture

- The standard form my now be obvious
  
  if (year%4 == 0) {
      feb_days = 29;
  } else {
      feb_days = 28;
  }

  Else must follow if because it does the test

  open brace, immediately after “else”

  finally, close brace

  The result is sets the number of days in February based on leap year
If/Else, if_else_red_blue

- Remember this early example?

```plaintext
int next=1;

void setup() {
  size(100,100);
  fill(255, 0,0);
}

void draw() {
  background(0);
  rect(mouseX, mouseY, 25, 25);
}

void mousePressed() {
  if (next == 1) {
    fill(0, 0, 255);  // go to blue
  } else {
    fill(255,0,0);    // go to red
  }
  next=1-next;
}
```
Writing Programs

- Programs are given sequentially, the declarations at the top
- Braces {} are statement groupers ... they make a sequence of statements into just one thing, like the “true clause of an If-statement”
- All statements must end with a semicolon EXCEPT the grouping braces ... they don’t end with a semicolon
- Name your variables something meaningful
- Generally white space doesn’t matter; comment your code
Layering: Building Functions out of Functions
Functions In Processing

- Form of function definition in Processing

  `<return type> <name> ( <param list> ) {`

  `<body>`

  `}`

  as in

  ```
  void draw_a_box (int x_pos, int y_pos) {
    rect(x_pos, y_pos, 20, 20);
  }
  color pink ( ) {
  or
  return color(255, 200, 200);
  }
  ```
Functions that do something, but do not return a value, have **void** as their *return type*

Functions that return a value must say its type

```java
void draw_a_box (int x_pos, int y_pos) {
    rect(x_pos, y_pos, 20, 20);
}

color pink () {
    return color(255, 200, 200);
}
```
Functions In Processing: Params

- Parameters are the values used as input to the function; parameters are not required, but the parentheses are.
- The type of each parameter must be given.

```java
void draw_a_box (int x_pos, int y_pos) {
    rect(x_pos, y_pos, 20, 20);
}

color pink () {
    return color(255, 200, 200);
}
```
Functions In Processing: Return

- A function returns its value with the `return` statement ... the stuff following return is the result
- The function is done when it reaches return

```cpp
void draw_a_box (int x_pos, int y_pos) {
  rect(x_pos, y_pos, 20, 20);
}

color pink () {
  return color(255, 200, 200);
}
```
Writing Functions

- Processing function definitions are typically listed after the standard blocks: setup(), draw(), mousePressed(), etc.

```java
void setup() {
  size(100, 100);
  background(0);
  noStroke();
}

void draw() {
  fill(255);
  hexa(20, 20);
}

void hexa(float xbase, float ybase) {
  rect(xbase, ybase+10, 20, 40);
  triangle(xbase, ybase+10, xbase+20, ybase+10, xbase+10, ybase);
  triangle(xbase, ybase+50, xbase+20, ybase+50, xbase+10, ybase+60);
}
```
Using Functions

- Once defined, functions can be called repeatedly ... it’s the point of writing them!

```c
void setup() {
  size(110, 100);
  background(0);
  noStroke();
}

void draw() {
  fill(255);
  hexa(20, 20);
  hexa(50, 20);
  hexa(80, 20);
}

void hexa(float xbase, float ybase) {
  rect(xbase, ybase+10, 20, 40);
  triangle(xbase, ybase+10, xbase+20, ybase+10, xbase+10, ybase);
  triangle(xbase, ybase+50, xbase+20, ybase+50, xbase+10, ybase+60);
}
```
Arguments Become Parameters

- Notice that if the DEFINITION has \( n \) parameters, the CALL needs \( n \) arguments
- The parameters and arguments correspond

```c
void draw( ) {
    fill(255);
    hexa(20, 40);
    hexa(50, 40);
    hexa(80, 40);
}

void hexa(float xbase, float ybase) {
    rect(xbase, ybase+10, 20, 40);
    triangle(xbase, ybase+10, xbase+20, ybase+10, xbase+10, ybase);
    triangle(xbase, ybase+50, xbase+20, ybase+50, xbase+10, ybase+60);
}
```

Inside of the function, the parameter, e.g. \( xbase \), is declared and initialized to the corresponding argument, e.g. 80. Then, the definition uses it, e.g. \( \text{rect} \,(80, \, 40+10, \, 20, \, 40) \)
Parameters

- Parameters are automatically declared (and initialized) on a call, and remain in existence as long as the function remains unfinished.

- When the function ends, the parameters vanish, only to be recreated on the next call.

- It is wise to choose parameter names that help you remember exactly what they mean.
  - colorFlag (Chao’s code),
  - dir (for direction)

- I chose xbase as the orientation point of the figure in the x direction, I use that name a lot and I know what it means.
Example: Clock Timer. In sample codes.

- Draw digital timer elements
- Assemble elements into digits
- Light digit segments to create numbers
- Select number based on a digit
Define hexa() and rexa()

- Patter: Parameterize the functions by a consistent position – upper left corner is good

```c
void draw() {
    fill(255);
    hexa(20, 40);
    rexa(30, 20);
}

void hexa(float xbase, float ybase) {
    rect(xbase, ybase+10, 20, 40);
    triangle(xbase, ybase+10, xbase+20, ybase+10, xbase+10, ybase);
    triangle(xbase, ybase+50, xbase+20, ybase+50, xbase+10, ybase+60);
}

void rexa(float xbase, float ybase) {
    triangle(xbase, ybase+10, xbase+10, ybase, xbase+10, ybase+20);
    rect(xbase+10, ybase, 40, 20);
    triangle(xbase+50, ybase, xbase+50, ybase+20, xbase+60, ybase+10);
}
```
void draw() {
    fill(255);
    digit(50, 20);
    digit(140, 20);
}

void hexa(float xbase, float ybase) {
    rect(xbase, ybase+10, 20, 40);
    triangle(xbase, ybase+10, xbase+20, ybase+10, xbase+10, ybase);
    triangle(xbase, ybase+50, xbase+20, ybase+50, xbase+10, ybase+60);
}

void rexa(float xbase, float ybase) {
    triangle(xbase, ybase+10, xbase+10, ybase, xbase+10, ybase+20);
    rect(xbase+10, ybase, 40, 20);
    triangle(xbase+50, ybase, xbase+50, ybase+20, xbase+60, ybase+10);
}

void digit(float xbase, float ybase) {
    hexa(xbase, ybase+10);       // left upper
    hexa(xbase, ybase+70);       // left lower
    rexa(xbase+10, ybase);       // mid horizontal
    rexa(xbase+10, ybase+60);    // top horizontal
    rexa(xbase+10, ybase+120);   // bot horizontal
    hexa(xbase+60, ybase+10);    // right upper
    hexa(xbase+60, ybase+70);    // right lower
}
Let There Be Light (and Dark)

- Define the illumination of the digit
  - Must declare two color variables, initialize to proper colors, use them in fill, and then check them

```cpp
color dark, lite;

void setup( ) {  
  size(250, 180);  
  background(0);  
  stroke(0);  
}

void draw( ) {  
  lite = color(255, 185, 0);  
  dark = color(64, 48, 0);  
  fill(dark);  
  digit(50, 20);  
  fill(lite);  
  digit(140, 20);  
}
```
Count In Lights: a function for each number

- Light up the digit for each number:

```c
void digit(float xbase, float ybase) {
    hexa(xbase, ybase+10); //left upper
    hexa(xbase, ybase+70); //left lower
    rexa(xbase+10, ybase); //top horizontal
    rexa(xbase+10, ybase+60); //mid horizontal
    rexa(xbase+10, ybase+120); //bot horizontal
    hexa(xbase+60, ybase+10); //right upper
    hexa(xbase+60, ybase+70); //right lower
}

void one (float xbase, float ybase) {
    hexa(xbase+60, ybase+10); //right upper
    hexa(xbase+60, ybase+70); //right lower
}

void two (float xbase, float ybase) {
    rexa(xbase+10, ybase); //top horizontal
    rexa(xbase+10, ybase+60); //mid horizontal
    rexa(xbase+10, ybase+120); //bot horizontal
    hexa(xbase+60, ybase+10); //right upper
    hexa(xbase, ybase+70); //left lower
}
```
Select A Number To Display

- Given an integer, display it in lights

```c
void sel(int n, float xbase, float ybase) {
    fill(lite);
    if (n == 0) {
        zero(xbase, ybase);
    }
    if (n==1) {
        one(xbase, ybase);
    }
    if (n==2) {
        two(xbase, ybase);
    }
    if (n==3) {
        three(xbase, ybase);
    }
    if (n==4) {
        four(xbase, ybase);
    }
    if (n==5) {
        five(xbase, ybase);
    }
    if (n==6) {
        six(xbase, ybase);
    }
    ...
}
```
Create a 3 Digit Display

```c
void three_digit(int n, float xbase, float ybase) {
    fill(dark);
    digit(50, 90);
    digit(140, 90);
    digit(260, 90);
    fill(lite);
    rect(xbase+185, ybase+125, 15, 15);
    sel(((n/100)%10, xbase, ybase);
    sel(((n/10)%10, xbase+90, ybase);
    sel(n%10, xbase+210, ybase);
}
```

Here’s The Action
Count up At The Frame Rate

color dark, lite;
int i;

void setup( ) {
    size(400, 300);
    background(0);
    noStroke();
    frameRate(10);
}

void draw( ) {
    lite = color(255,185,0);
dark = color(64, 48, 0);
i = i + 1;
    three_digit(i, 50, 90);
}
Functional Abstraction: Layers of Functions

- Review What We Did

- The computation is ONLY drawing triangles and rectangles, but we don’t think of it that way ... to us, it’s a timer
Homework 10: Functions

- `void cell(int x, int y, int s, color tinto)`
- `void triple(int x, int y, int s, color tinto)`
- Also functions for:
  - `block()`
  - `row()`
  - `cellarray()`

Sudoku GUI
The Sudoku board is a 9×9 array of cells. Alternating 3×3 cells are colored gray and strong lines separate all of the 3×3 cells, as follows. (More information at http://www.websudoku.com/.)
Binary Arithmetic; We’ve seen lots of bits and bytes already
### ASCII Table

<table>
<thead>
<tr>
<th>ASCII</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>:</th>
<th>;</th>
<th>&lt;</th>
<th>=</th>
<th>&gt;</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### ASCII Characters

- `!":#$%&'()*/+,-./` are represented by `0010` to `0011`.
- `0100 0111` is a valid ASCII code for `@`.
- `0100 0001` is a valid ASCII code for `A`.
- `0101 0100` is a valid ASCII code for `C`.
With 8 places how many different letters?
Logic to Gates and Back again

Draw the logic diagrams for the following logical statements.
(Note: You do not need to include the switches shown in the logic diagram above. You may represent your inputs with simple lines as shown in lecture.)

(a) NOT (P OR Q)

(b) (A OR B) AND (NOT C)

2.2

Create a truth table and write the logical statement for the following logic diagram. Consider the inputs A and B to have truth values like that of the switch in the picture; each is either on (true or 1) or off (false or 0).
## Binary combinations, True/False possibilities

<table>
<thead>
<tr>
<th>One bit</th>
<th>Three bits</th>
<th>Four bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td>010</td>
<td>0010</td>
</tr>
<tr>
<td>Two Bits</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>00</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>01</td>
<td>101</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>110</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>1011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1111</td>
</tr>
</tbody>
</table>
White, Gray, Black

- You know that gray is just different degrees of white as the “light is turned down” till we get to black.

  Black = [0, 0, 0] 0000 0000 0000 0000 0000 0000
  Gray = [128,128,128] 1000 0000 1000 0000 1000 0000
  White = [255,255,255] 1111 1111 1111 1111 1111 1111

White-gray-black all have same values for RGB.
Step 2: Replace Bits In Host

- Put guest bits into right 2 bits of host

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101

1111 0100 1101 0011 1011 1101
BUT…. Arithmetic?
Colors

Colors use different combinations of RGB

**Purple**
Red=160
Green=76
Blue=230
Positional Notation: More after the midterm

- The RGB intensities are binary numbers
- Binary numbers, like decimal numbers, use *place notation*

\[
1101 = 1 \times 1000 + 1 \times 100 + 0 \times 10 + 1 \times 1
= 1 \times 10^3 + 1 \times 10^2 + 0 \times 10^1 + 1 \times 10^0
\]

except that the base is 2 not 10

\[
1101 = 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1
= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0
\]

1101 in binary is 13 in decimal
Positional Notation Logic: Base 10 vs. Base 2 (binary)

Recall that the place represents a power of the base value

\[
\begin{align*}
\text{Base 10:} & \quad d_7 \times 10^7 \\
& \quad d_6 \times 10^6 \\
& \quad d_5 \times 10^5 \\
& \quad d_4 \times 10^4 \\
& \quad d_3 \times 10^3 \\
& \quad d_2 \times 10^2 \\
& \quad d_1 \times 10^1 \\
& \quad d_0 \times 10^0
\end{align*}
\]

\[
\begin{align*}
\text{Base 2:} & \quad d_7 \times 2^7 \\
& \quad d_6 \times 2^6 \\
& \quad d_5 \times 2^5 \\
& \quad d_4 \times 2^4 \\
& \quad d_3 \times 2^3 \\
& \quad d_2 \times 2^2 \\
& \quad d_1 \times 2^1 \\
& \quad d_0 \times 2^0
\end{align*}
\]
A Conjecture: Knuth states the first known appearance of pure binary notation was about 1605 with Thomas Harriot, but he suggests another intriguing idea. He lists the common units of liquid measure used by English Wine Merchants as late as the 13th century.

This is effectively a binary counting system using a 14 bit word. Each measure, twice as large as the last, has its own name.

In Ohio we buy peaches by the PECK or BUSHEL

2 gills = 1 chopin
2 chopins = 1 pint
2 pints = 1 quart
2 quarts = 1 pottle
2 pottles = 1 gallon
2 gallons = 1 peck
2 pecks = 1 demibushel
2 demibushels = 1 bushel (or firkin)
2 firkins = 1 kilderkin
2 kilderkins = 1 barrel
2 barrels = 1 hogshead
2 hogsheads = 1 pipe
2 pipes = 1 tun
If you represent each named unit as a single bit and combine them all, you get a binary number.

Say I have

- a hogshead,
- a firkin,
- a gallon, and
- a pint of wine,

I could write the measured amount of wine as 00100100100100 gills, where "gills" is the LSB (least significant bit) and "tuns" is the MSB (most significant bit).

\[
\begin{align*}
2 \text{ gills} &= 1 \text{ chopin} \\
2 \text{ chopins} &= 1 \text{ pint} \\
2 \text{ pints} &= 1 \text{ quart} \\
2 \text{ quarts} &= 1 \text{ pottle} \\
2 \text{ pottles} &= 1 \text{ gallon} \\
2 \text{ gallons} &= 1 \text{ peck} \\
2 \text{ pecks} &= 1 \text{ demibushel} \\
2 \text{ demibushels} &= 1 \text{ bushel (or firkin)} \\
2 \text{ firkins} &= 1 \text{ kilderkin} \\
2 \text{ kilderkins} &= 1 \text{ barrel} \\
2 \text{ barrels} &= 1 \text{ hogshead} \\
2 \text{ hogsheads} &= 1 \text{ pipe} \\
2 \text{ pipes} &= 1 \text{ tun} \\
1 \text{ TUN} &= \text{ the MSB}
\end{align*}
\]
The Red of \( P \) As A Binary Number

**Purple:** Red=160, Green=76, Blue=230

Given a binary number, add up the powers of 2 corresponding to 1s

\[
\begin{align*}
1 \times 2^7 &= 1 \times 128 &= 128 \\
0 \times 2^6 &= 0 \times 64 &= 0 \\
1 \times 2^5 &= 1 \times 32 &= 32 \\
0 \times 2^4 &= 0 \times 16 &= 0 \\
0 \times 2^3 &= 0 \times 8 = 0 \\
0 \times 2^2 &= 0 \times 4 = 0 \\
0 \times 2^1 &= 0 \times 2 = 0 \\
0 \times 2^0 &= 0 \times 1 = 0 \\
\end{align*}
\]

\[= 160\]
Green of P As A Binary Number

Purple: Red=160, Green=76, Blue=230

Given a binary number, add up the powers of 2 corresponding to 1s

\[
\begin{align*}
0 \times 2^7 & = 1 \times 128 \quad = 0 \\
1 \times 2^6 & = 0 \times 64 \quad = 64 \\
0 \times 2^5 & = 1 \times 32 \quad = 0 \\
0 \times 2^4 & = 0 \times 16 \quad = 0 \\
1 \times 2^3 & = 0 \times 8 = 8 \\
1 \times 2^2 & = 0 \times 4 = 4 \\
0 \times 2^1 & = 0 \times 2 = 0 \\
0 \times 2^0 & = 0 \times 1 = 0 \\
\end{align*}
\]

=76
Is It Really the Purple we want?

- So Purple is \((160,76,230)\) which is
  
  \[
  \begin{align*}
  &10100000 \\
  &01001100 \\
  &11100110
  \end{align*}
  
  \[
  \begin{align*}
  &160 \\
  &76 \\
  &230
  \end{align*}
  
  Suppose you decide it’s not “red” enough

  - Increase the red by \(16 = 10000\)
    
    \[
    \begin{align*}
    &10100000 \\
    + &10000 \\
    \hline
    &\begin{array}{c}
    10110000
    \end{array}
    \end{align*}
    
    Adding in binary is pretty much like adding in decimal
How do we make a Redder Purple

ADD 16 more

```
00110 000
1011 0000
+ 1 0000
1100 0000
```

Carries

| Original | +16 | +16 |

The rule: When the "place sum" equals the radix (base) or more, subtract radix (base 2) & carry

Check it out online: searching hits 19M times, and all of the p.1 hits are good explanations

binary addition
Find Binary From Decimal

What is 230 (the Blue of P)? Fill in the Table:

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>230</th>
<th>230</th>
<th>102</th>
<th>38</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Value</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Subtract. Remainder</td>
<td>102</td>
<td>38</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Binary Num</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

2/16/12 © 2010 Larry Snyder, CSE
Find Binary From Decimal

Place number to be converted into the table; fill place value row with decimal powers of 2

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>Place Value</th>
<th>Binary Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>256 128 64 32 16 8 4 2 1</td>
<td></td>
</tr>
</tbody>
</table>

Subtract
Find Binary From Decimal

Rule: Subtract PV from the number; a positive result gives new number and “1”; otherwise, “0”

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>230</th>
<th>230</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Value</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>Subtract</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>Binary Num</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

© 2010 Larry Snyder, CSE
Find Binary From Decimal

Rule: Subtract PV from the number; a positive result gives new number and “1”; otherwise, “0”

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>Place Value</th>
<th>Subtract</th>
<th>Binary Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>256</td>
<td>102</td>
<td>0</td>
</tr>
<tr>
<td>230</td>
<td>128</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Find Binary From Decimal

Rule: Subtract PV from the number; a positive result gives new number and “1”; otherwise, “0”

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>230</th>
<th>230</th>
<th>102</th>
<th>38</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Value</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>Subtract</td>
<td>102</td>
<td>38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Num</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

© 2010 Larry Snyder, CSE
Find Binary From Decimal

Rule: Subtract PV from the number; a positive result gives new number and “1”; otherwise, “0”

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>230</th>
<th>230</th>
<th>102</th>
<th>38</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Value</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>Subtract</td>
<td>102</td>
<td>38</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Num</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

© 2010 Larry Snyder, CSE
Find Binary From Decimal

Rule: Subtract PV from the number; a positive result gives new number and “1”; otherwise, “0”

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>230</th>
<th>230</th>
<th>102</th>
<th>38</th>
<th>6</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Value</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Subtract</td>
<td>102</td>
<td>38</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Num</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Find Binary From Decimal

Rule: Subtract PV from the number; a positive result gives new number and “1”; otherwise, “0”

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>Place Value</th>
<th>Subtract</th>
<th>Binary Num</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>256</td>
<td>102</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>230</td>
<td>64</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>32</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>16</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Find Binary From Decimal

Rule: Subtract PV from the number; a positive result gives new number and “1”; otherwise, “0”

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>230</th>
<th>230</th>
<th>102</th>
<th>38</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Value</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Subtract</td>
<td>102</td>
<td>38</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Binary Num</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

© 2010 Larry Snyder, CSE
## Find Binary From Decimal

Rule: Subtract PV from the number; a positive result gives new number and "1"; otherwise, "0"

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>230</th>
<th>230</th>
<th>102</th>
<th>38</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Value</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Subtract</td>
<td>102</td>
<td>38</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Num</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Find Binary From Decimal

Rule: Subtract PV from the number; a positive result gives new number and “1”; otherwise, “0”

<table>
<thead>
<tr>
<th>Num Being Converted</th>
<th>230</th>
<th>230</th>
<th>102</th>
<th>38</th>
<th>6</th>
<th>6</th>
<th>6</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Value</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Subtract</td>
<td>102</td>
<td>38</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Num</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Read off the result: 0 1110 0110

2/16/12

© 2010 Larry Snyder, CSE