bases
storing numbers
base conversion
how different types of data are stored in the computer
hexadecimal and octal constants

storing numbers
when your code says:
```
int x = 19;
```
and we draw the computer’s memory to look like this:
```
x ← 0
```
what is really stored looks like this:
```
000000000000000000000000000010011
```
where each 0 or 1 is a switch that is either off (0) or on (1)
the set of switches can be interpreted as a binary or base 2 number!
```
19_{10} = 10011_{2}
```

remember bases?

base 10:
```
362 = (2 * 1) + (6 * 10) + (3 * 100)
= (2 * 10^0) + (6 * 10^1) + (3 * 10^2)
```
base 2:
```
1 = 2^0 = 1
10 = 2^1 = 2
100 = 2^2 = 4
1000 = 2^3 = 8
10000 = 2^4 = 16
```
so
```
10011_2 = (1 * 2^2) + (1 * 2^1) + (0 * 2^2) + (0 * 2^1) + (1 * 2^0)
= (1 * 1) + (1 * 2) + (0 * 4) + (0 * 8) + (1 * 16)
= 19_{10}
```

base conversion: 2 to 10
```
1010100_2 =
= (0 * 2^5) + (0 * 2^4) + (1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (0 * 2^0)
= (0 * 32) + (0 * 16) + (1 * 8) + (0 * 4) + (1 * 2) + (0 * 1)
= 84_{10}
```
```
base conversion: 10 to 2

\[
\begin{align*}
84_{10} &= 84 / 2 = 42 \text{ rem } 0 \\
42 / 2 &= 21 \text{ rem } 0 \\
21 / 2 &= 10 \text{ rem } 1 \\
10 / 2 &= 5 \text{ rem } 0 \\
5 / 2 &= 2 \text{ rem } 1 \\
2 / 2 &= 1 \text{ rem } 0 \\
1 / 2 &= 0 \text{ rem } 1 \\
\Rightarrow 1010100_2
\end{align*}
\]

two tricks

<table>
<thead>
<tr>
<th>base 8 (octal):</th>
<th>base 16 (hexadecimal, &quot;hex&quot;):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0</td>
<td>0000 0 1000 8</td>
</tr>
<tr>
<td>0001 1</td>
<td>0001 1 1001 9</td>
</tr>
<tr>
<td>0100 2</td>
<td>0010 2 1010 A (10)</td>
</tr>
<tr>
<td>0111 3</td>
<td>0011 3 1011 B (11)</td>
</tr>
<tr>
<td>1000 4</td>
<td>0100 4 1100 C (12)</td>
</tr>
<tr>
<td>1011 5</td>
<td>0101 5 1101 D (13)</td>
</tr>
<tr>
<td>1110 6</td>
<td>0110 6 1110 E (14)</td>
</tr>
<tr>
<td>1111 7</td>
<td>0111 7 1111 F (15)</td>
</tr>
</tbody>
</table>

- replace each octal (or hex) digit with the 3 (or 4) digit binary
- replace every 3 (or 4) binary digits with one octal (or hex) digit

back to storage and how different types of data are stored

\[ x \rightarrow [19] \]

is really stored like this:

```
31 30 ... 7 6 5 4 3 2 1 0
0 0 ... 0 0 0 1 0 1 1
```

- bits are numbered, from right to left, starting with 0
- the highest (rightmost, "most significant") bit (i.e., bit number 31) is the sign bit
- if the sign bit is 0, then the number is positive;
  if the sign bit is 1, then the number is negative
- negative numbers are encoding using a method called two’s complement
- integers can also be unsigned
  which means that the sign bit is interpreted as another binary digit
- the largest signed integer value is: \( 2^{31} - 1 = 2,147,483,648 \)
- the largest unsigned integer value is: \( 2^{32} - 1 = 4,294,967,295 \)

storing characters: ASCII

- ASCII = American Standard Code for Information Interchange
- characters are stored as numbers
- standard table defines 128 characters
- for example, when you define:
  ```
  char c = 'A';
  ```
  the data is stored as a number:
  `'A' = 65_{10} = 01000001_2` like this:
  ```
  c = [7 6 5 4 3 2 1 0]
  0 1 0 0 0 0 1 1
  ```
- sometimes it is handy to convert between integers and characters explicitly
  ```
  char c = 'A';
  int i;
  i = (int)c;
  ```
  in which case, the value of 1 will be 65.
### The Sizes of Primitive Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>1 bit</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits</td>
<td>(-128 = -2^{7})</td>
<td>(127 = 2^{7} - 1)</td>
</tr>
<tr>
<td>char</td>
<td>8 bits</td>
<td>(-128 = -2^{7})</td>
<td>(127 = 2^{7} - 1)</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>(-32,768 = -2^{15})</td>
<td>(32,767 = 2^{15} - 1)</td>
</tr>
<tr>
<td>int</td>
<td>32 (or 16) bits</td>
<td>(-2^{31} (2^{15}))</td>
<td>(2^{31} - 1 (2^{15} - 1))</td>
</tr>
<tr>
<td>long</td>
<td>32 bits</td>
<td>(-2^{31})</td>
<td>(2^{31} - 1)</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>(-3.4E+38, 7) sig. dig.</td>
<td>(3.4E+38, 7) sig. dig.</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>(-1.7E+308, 15) sig. dig.</td>
<td>(1.7E+308, 15) sig. dig.</td>
</tr>
</tbody>
</table>

"sig. dig." = significant digits

(Note that the minimum and maximum values given above are based on using signed numbers.)

### Finding the Size of Things in a Program

- C++ has a function called `sizeof()` which returns the size of its argument, in bytes.
- For example, `sizeof(a)`, where `verb+a+` is defined as an `int` variable, returns the value 4.
- Since there are 8 bits in a byte, then an `int` takes up \(4 \times 8 = 32\) bits.
- For example:
  ```cpp
  #include <iostream>
  using namespace std;
  int main() {
    int i;
    cout << "the number of bytes in an int is: " << sizeof(i) << endl;
    cout << "the number of bits in an int is: " << sizeof(i) * 8 << endl;
  } // end of main()
  ```

### Hexadecimal and Octal Constants

- Sometimes it is handy to use the hexadecimal (base 16) or octal (base 8) representation of a number in a program.
- In C++, octal numbers are represented by using a leading zero in the number, which indicates that it is octal.
- Hexadecimal numbers are represented using `0x` before the value.
- For example:
  ```cpp
  #include <iostream>
  using namespace std;
  int main() {
    int i = 10; // set value using decimal notation (base 10)
    int o = 010; // set value using octal (base 8)
    int h = 0x10; // set value using hexadecimal (base 16)
    cout << "i = " << i << endl;
    cout << "o = " << o << endl;
    cout << "h = " << h << endl;
  } // end of main()
  ```

### Want More?

- Things to look up or research on your own:
  - How to print out numbers in octal and hexadecimal (see textbook page 251).
  - What is "two’s complement"?
- Note that there are appendices in the text book that list:
  - Sizes of all the primitive data types.
  - ASCII table.
- For fun, have a look at [http://www.asciiation.co.nz/]...