Binary Representation

CSE 410
Lecture 03
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
First: Why Binary?

- Electronic implementation
  - Easy to store
  - Reliably transmitted on noisy and inaccurate wires

- Other bases possible:
  - Distinguish more voltage levels
  - DNA data storage (base 4: A, C, G, T)
Bit

- A bit is a single binary value
- “Binary” means there are (only) two distinct values
  - in computers, high and low voltage
- We can map the two values to any other pair of values
  - Orange vs Apple; Up vs Down; 8 vs 10; 0 vs 1; true vs false
- Of these, the last two have many attractive properties
  - 0 and 1 → base-2 (binary) integers
  - true and false → Boolean circuits
Bit Operations

- **Unary operation**
  - not
    - \( \sim 1 == 0 \)
    - \( \sim 0 == 1 \)

- **Binary operations**
  - and
    - \( 0 \& 0 == 0 \)
    - \( 0 \& 1 == 0 \)
    - \( 1 \& 0 == 0 \)
    - \( 1 \& 1 == 1 \)

*Operators are written as in C (and many other languages)*

*Note that operator \& is different from operator &&*
Bit Operations

- Binary Operations
  - or
    - $0 | 0 = 0$
    - $0 | 1 = 1$
    - $1 | 0 = 1$
    - $1 | 1 = 1$
  - xor ("exclusive or")
    - $0 ^ 0 = 0$
    - $0 ^ 1 = 1$
    - $1 ^ 0 = 1$
    - $1 ^ 1 = 0$
Bit Strings

- A concatenation of bits
  - Example: 01010111

- The bit operators can be applied to bit strings
  - \[ \begin{array}{c}
  01010111 \\
  \& 11000110 \\
  \end{array} \]
  
  \[\begin{array}{c}
  01000110 \\
  \end{array}\]

  - Similarly for |, ^, and ~
Bit String Shift

- **Left shift:** `<<`
  
  - `0 1 0 1 0 1 0 1 << 1 == 1 0 1 0 1 0 1 0`
  - `0 1 0 1 0 1 0 1 << 3 == 1 0 1 0 1 0 0 0`

- **Right shift arithmetic:** `>>`
  
  - `0 1 0 1 0 1 0 1 >> 1 == 0 0 1 0 1 0 1 0`
  - `0 1 0 1 0 1 0 1 >> 3 == 0 0 0 0 1 0 1 0`

- **Note:** right shift has two form, arithmetic and logical
  
  - Arithmetic propagates the high order bit
    
    - `0 1 0 1 0 1 0 1 >> 3 == 0 0 0 0 1 0 1 0`
    - `1 0 1 0 1 0 1 0 >> 3 == 1 1 1 1 0 1 0 1`
  
  - Logical shifts in zeros from the left
Bit Masks: and

❖ “and masks” turn off bits wherever the mask has a 0
  ▪ Example mask: 0 0 0 0 0 0 1
    • And’ed with another 8 bit string, it copies the low order bit of the other string and sets everything else to zero
      1 1 1 1 1 1 1
      & 0 0 0 0 0 0 1
      ------------------
      0 0 0 0 0 0 1
    • Other masks:
      − 0 0 0 0 0 1 1 => copy two low order bits
      − 0 0 0 0 1 1 0 0 => copy bits 2 and 3
      − etc.
Bit masks: or

- or masks turn on bits (wherever the mask has a 1)
  - Example mask: \[ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \]
    \[ \begin{array}{cccccccc}
    1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
    0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
    \end{array} \]
  
  \[ \begin{array}{cccccccc}
    1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
    \end{array} \]
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Integers and Integer Representations

- What is 7061?
  - It’s not an integer, it’s a “place value” representation of an integer
    - We could equally write $7 \times 10^3 + 0 \times 10^2 + 6 \times 10^1 + 1 \times 10^0$ but that’s a lot less convenient
    - We could write $353 \times 20 + 1$, but that’s even dumber (why?)

- What about 70000000000000000000061?
  - It might be handier to write $7 \times 10^{22} + 61$

- There is no “right representation” there are just ones that are more convenient than others
Place value representation

- We write $n$ consecutive digits, numbering them 0 to $n-1$ starting from the right. Place $j$ has value $b^j$ for some base $b$.
- We write in each place a *digit*. There are $b$ digits, representing the numbers 0, 1, 2, ..., $b-1$.

$$\begin{align*}
\frac{d_3}{b^3} & \quad \frac{d_2}{b^2} & \quad \frac{d_1}{b^1} & \quad \frac{d_0}{b^0}
\end{align*}$$

- The place value string represents the integer $d_{n-1}b^{n-1} + d_{n-2}b^{n-2} + ... + d_0b^0$
Example: 1024

- **b=10 (decimal)**
  - Digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - 1024 means $1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$

- **b=2 (binary)**
  - Digits are 0, 1
  - 10000000000 means $1 \times 2^{10}$ (plus a lot of “zero times x” terms)
    - Which is 1024 in base 10
Simplifying representations

- Which is bigger, 231237943432586732275839 or 23123794343584332235839? (Both are base 10)
- We (humans) prefer representations with fewer digits
- We can reduce the number of digits a factor of k by raising the base by a power of k.
  - E.g., instead of base 10, use base 1000
    - Of course, we now need a 1000 different symbols for digits
Simplifying binary

- Start with binary: 00000001001000110100010101100111

- Octal: Raise the base by a power of 3 (so, base 8)
  00 000 001 001 000 110 100 010 101 100 111
  0 0 1 1 0 6 4 2 5 4 7

- Hexadecimal: Raise the base by a power of 4 (base 16)
  0000 0001 0010 0011 0100 0101 0110 0111
  0 1 2 3 4 5 6 7
Hexadecimal

- Grouping by four bits is handy
  - Memories are always a multiple of 8 bits in length
    - We’ll see later why
- Hex digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Correspond to values in base 10 of 0, 1, ..., 9, 10, 11, 12, 13, 14, 15
  - Case insensitive
## Hex ↔ Binary

<table>
<thead>
<tr>
<th>Hex Digit</th>
<th>Binary String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>

**What is FFFF in binary?**

**Is 237E even or odd?**

We should specific what base we’re using when writing integers. In C:
- 123 is a decimal constant
- 0123 is an octal constant
- 0X0123 is a hex constant
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Addition with Place Value Representations

- Addition is **easy** with the standard algorithm (carry ripple)

  1
  0 0 1 0 2
  0 1 1 1 7
  1 0 0 1 9

- One problem: what about addition of negative numbers?

  24
  +(−7)

- Another problem: Hey, what about negative numbers at all?

- Last problem: Overflow
Overflow

- A fixed amount of space is allocated for each value on a computer
  - For integers, usually 1, 2, 4, or 8 bytes (8, 16, 32, or 64 bits)
- Q: What if the result is too big to fit in that much space?
- A: Too bad. The highest order digit is thrown away.
- That’s called overflow
Two’s Complement Binary Integers

- Two’s complement is a representation for positive and negative integers
  - Addition is always addition, even if one or both values are negative
  - About half the bit strings are negative and half are positive

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>-4</th>
<th>-3</th>
<th>-2</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>

Verify that x + -x == 0
Properties

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>unsigned</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- If you count up from 0 by 1, you wrap from the largest positive integer to the smallest negative integer.
- If the high order bit is 0, the number is non-negative. If it’s 1, the number is negative.
- You get the same bit string result adding bit strings as signed values as you do adding them as unsigned.
Overflow

- Overflow occurs when the result doesn’t fit in the limited number of bits you have
  - $0001 + 0111 \Rightarrow 1001$
    
    $1 + 7 = -8$

- You can overflow when subtracting or multiplying as well

- Unsigned integers also overflow
  - $0001 + 0111 = 1000$
    
    $1 + 7 = 8$ [no overflow]
  - $0001 + 1111 = 0000$
    
    $1 + 15 = 0$
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Floating Point Overview

- We have only 32 bits, so we have only $2^{32}$ different values we can represent

- We’re going to do the binary version of scientific notation: $2.357 \times 10^{14}$

- Different representations have different:
  - range – roughly, how big the exponent can be
  - precision – basically the number of significant digits in the fraction
Floating Point: Limited Digits

- Let’s start in decimal to see the issues
- Suppose you have enough space in a word to hold 8 decimal digits
  - How many digits should you use to represent the exponent and how many the fraction
- At the extremes
  - $1 \times 10^{2345678}$
  - $1.234567 \times 10^8$
- So
  - Want some balance in range vs precision
  - Also need a way to represent negative values
  - Also need a way to represent negative exponents
32-bit Binary Floats

- Called “single precision” floats
- Value is \([+/-] \text{[fraction]} \times 2^{\text{exponent}}\)

The 32 bits are used as:
- High order bit is the sign of the value: 1 for negative, 0 for non-negative
- The next 8 bits are the signed (two’s complement) value for the exponent: 127 to -128
- The remaining 23 bits are the fraction

Range: approximately \(2.0 \times 10^{38}\) to \(2.0 \times 10^{-38}\)

Numbers can overflow: exponent gets too big
Numbers can underflow: exponent gets too small
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Character Representation

- We simply agree on a mapping from bit strings to characters
  - “Everyone” knows what the mapping is
  - The compiler inserts the agreed bit string when you write ‘A’
  - The output system writes A when it sees that bit string

- There is more than one agreed representation

  - ASCII
    - Historically the agreed mapping
    - Fixed, 8-bit long strings

  - Unicode
    - Variable length encoding: 8, 16, or 32 bits per character
    - Many, many more bit strings, so many, many more characters
## ASCII

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th></th>
<th>32 &lt;NUL&gt;</th>
<th>64</th>
<th>@</th>
<th>96</th>
<th>128 Å</th>
<th>160 †</th>
<th>192 Ʝ</th>
<th>224 ±</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;SOH&gt;</td>
<td>33</td>
<td>!</td>
<td>65</td>
<td>A</td>
<td>97</td>
<td>129 Å</td>
<td>161 ◦</td>
<td>193 i</td>
<td>225</td>
</tr>
<tr>
<td>2</td>
<td>&lt;STX&gt;</td>
<td>34</td>
<td>&quot;</td>
<td>66</td>
<td>B</td>
<td>98</td>
<td>130 Ç</td>
<td>162 ç</td>
<td>194 ÷</td>
<td>226</td>
</tr>
<tr>
<td>3</td>
<td>&lt;ETX&gt;</td>
<td>35</td>
<td>#</td>
<td>67</td>
<td>C</td>
<td>99</td>
<td>131 É</td>
<td>163 É</td>
<td>195 √</td>
<td>227</td>
</tr>
<tr>
<td>4</td>
<td>&lt;EOT&gt;</td>
<td>36</td>
<td>$</td>
<td>68</td>
<td>D</td>
<td>100</td>
<td>132 Ń</td>
<td>164 §</td>
<td>196 f</td>
<td>228</td>
</tr>
<tr>
<td>5</td>
<td>&lt;ENQ&gt;</td>
<td>37</td>
<td>%</td>
<td>69</td>
<td>E</td>
<td>101</td>
<td>133 Õ</td>
<td>165 ⋅</td>
<td>197 ≈</td>
<td>229</td>
</tr>
<tr>
<td>6</td>
<td>&lt;ACK&gt;</td>
<td>38</td>
<td>&amp;</td>
<td>70</td>
<td>F</td>
<td>102</td>
<td>134 Ü</td>
<td>166 ¶</td>
<td>198 Δ</td>
<td>230</td>
</tr>
<tr>
<td>7</td>
<td>&lt;BEL&gt;</td>
<td>39</td>
<td>'</td>
<td>71</td>
<td>G</td>
<td>103</td>
<td>135 ß</td>
<td>167 β</td>
<td>199 «</td>
<td>231</td>
</tr>
<tr>
<td>8</td>
<td>&lt;BS&gt;</td>
<td>40</td>
<td>(</td>
<td>72</td>
<td>H</td>
<td>104</td>
<td>136 à</td>
<td>168 ®</td>
<td>200 »</td>
<td>232</td>
</tr>
<tr>
<td>9</td>
<td>&lt;TAB&gt;</td>
<td>41</td>
<td>)</td>
<td>73</td>
<td>I</td>
<td>105</td>
<td>137 å</td>
<td>169 ©</td>
<td>201 ...</td>
<td>233</td>
</tr>
<tr>
<td>10</td>
<td>&lt;LF&gt;</td>
<td>42</td>
<td>*</td>
<td>74</td>
<td>J</td>
<td>106</td>
<td>138 ä</td>
<td>170 ™</td>
<td>202</td>
<td>234</td>
</tr>
<tr>
<td>11</td>
<td>&lt;VT&gt;</td>
<td>43</td>
<td>+</td>
<td>75</td>
<td>K</td>
<td>107</td>
<td>139 ā</td>
<td>171  ’</td>
<td>203  À</td>
<td>235</td>
</tr>
<tr>
<td>12</td>
<td>&lt;FF&gt;</td>
<td>44</td>
<td>,</td>
<td>76</td>
<td>L</td>
<td>108</td>
<td>140 å</td>
<td>172 ”</td>
<td>204  À</td>
<td>236</td>
</tr>
<tr>
<td>13</td>
<td>&lt;CR&gt;</td>
<td>45</td>
<td>-</td>
<td>77</td>
<td>M</td>
<td>109</td>
<td>141 ç</td>
<td>173 ≠</td>
<td>205  Ō</td>
<td>237</td>
</tr>
<tr>
<td>14</td>
<td>&lt;SO&gt;</td>
<td>46</td>
<td>.</td>
<td>78</td>
<td>N</td>
<td>110</td>
<td>142 é</td>
<td>174 Æ</td>
<td>206  Æ</td>
<td>238</td>
</tr>
<tr>
<td>15</td>
<td>&lt;SI&gt;</td>
<td>47</td>
<td>/</td>
<td>79</td>
<td>O</td>
<td>111</td>
<td>143 è</td>
<td>175 Ø</td>
<td>207  ø</td>
<td>239</td>
</tr>
<tr>
<td>16</td>
<td>&lt;DLE&gt;</td>
<td>48</td>
<td>0</td>
<td>80</td>
<td>P</td>
<td>112</td>
<td>144 ê</td>
<td>176 ∞</td>
<td>208</td>
<td>240</td>
</tr>
<tr>
<td>17</td>
<td>&lt;DC1&gt;</td>
<td>49</td>
<td>1</td>
<td>81</td>
<td>Q</td>
<td>113</td>
<td>145 ë</td>
<td>177 ±</td>
<td>209</td>
<td>241</td>
</tr>
<tr>
<td>18</td>
<td>&lt;DC2&gt;</td>
<td>50</td>
<td>2</td>
<td>82</td>
<td>R</td>
<td>114</td>
<td>146 i</td>
<td>178 ≤</td>
<td>210</td>
<td>242</td>
</tr>
<tr>
<td>19</td>
<td>&lt;DC3&gt;</td>
<td>51</td>
<td>3</td>
<td>83</td>
<td>S</td>
<td>115</td>
<td>147 ī</td>
<td>179 ≥</td>
<td>211</td>
<td>243</td>
</tr>
<tr>
<td>20</td>
<td>&lt;DC4&gt;</td>
<td>52</td>
<td>4</td>
<td>84</td>
<td>T</td>
<td>116</td>
<td>148 ī</td>
<td>180 ¥</td>
<td>212</td>
<td>244</td>
</tr>
<tr>
<td>21</td>
<td>&lt;NAK&gt;</td>
<td>53</td>
<td>5</td>
<td>85</td>
<td>U</td>
<td>117</td>
<td>149 ĩ</td>
<td>181 µ</td>
<td>213</td>
<td>245</td>
</tr>
<tr>
<td>22</td>
<td>&lt;SYN&gt;</td>
<td>54</td>
<td>6</td>
<td>86</td>
<td>V</td>
<td>118</td>
<td>150 ŉ</td>
<td>182 Ø</td>
<td>214</td>
<td>246</td>
</tr>
<tr>
<td>23</td>
<td>&lt;ETB&gt;</td>
<td>55</td>
<td>7</td>
<td>87</td>
<td>W</td>
<td>119</td>
<td>151 ô</td>
<td>183 Ï</td>
<td>215</td>
<td>247</td>
</tr>
<tr>
<td>24</td>
<td>&lt;CAN&gt;</td>
<td>56</td>
<td>8</td>
<td>88</td>
<td>X</td>
<td>120</td>
<td>152 ò</td>
<td>184 Ï</td>
<td>216</td>
<td>248</td>
</tr>
<tr>
<td>25</td>
<td>&lt;EM&gt;</td>
<td>57</td>
<td>9</td>
<td>89</td>
<td>Y</td>
<td>121</td>
<td>153 ô</td>
<td>185 Ï</td>
<td>217</td>
<td>249</td>
</tr>
<tr>
<td>26</td>
<td>&lt;SUB&gt;</td>
<td>58</td>
<td>:</td>
<td>90</td>
<td>Z</td>
<td>122</td>
<td>154 Ũ</td>
<td>186 Ï</td>
<td>218</td>
<td>250</td>
</tr>
<tr>
<td>27</td>
<td>&lt;ESC&gt;</td>
<td>59</td>
<td>;</td>
<td>91</td>
<td>[</td>
<td>123  {</td>
<td>155 Ō</td>
<td>187 Ï</td>
<td>219</td>
<td>251</td>
</tr>
<tr>
<td>28</td>
<td>&lt;FS&gt;</td>
<td>60</td>
<td>&lt;</td>
<td>92</td>
<td>\</td>
<td>124</td>
<td>156 Ū</td>
<td>188 Ï</td>
<td>220</td>
<td>252</td>
</tr>
<tr>
<td>29</td>
<td>&lt;GS&gt;</td>
<td>61</td>
<td>=</td>
<td>93</td>
<td>]</td>
<td>125  }</td>
<td>157 Ū</td>
<td>189 Ï</td>
<td>221</td>
<td>253</td>
</tr>
<tr>
<td>30</td>
<td>&lt;RS&gt;</td>
<td>62</td>
<td>&gt;</td>
<td>94</td>
<td>^</td>
<td>126  ~</td>
<td>158 Ū</td>
<td>190 æ</td>
<td>222</td>
<td>254</td>
</tr>
<tr>
<td>31</td>
<td>&lt;US&gt;</td>
<td>63</td>
<td>?</td>
<td>95</td>
<td>_</td>
<td>127  &lt;DEL&gt;</td>
<td>159 Ū</td>
<td>191 ø</td>
<td>223</td>
<td>255</td>
</tr>
</tbody>
</table>
Lecture Outline

- **Binary**
- **Decimal, Binary, and Hexadecimal Integers**
- **Why Place Value Representation**
  - And why not
- **Floating Point Representation**
- **Character Representation**
- **Pointer Representation**
- **Array Representation**
- **Structure Representation**
C Language Pointers

- `int x;` // x names a 32-bit string we’ll use as an int

- `int *p;` // p names a bit string that can hold a
  // memory address

- `p = &x;` // set p to the address of x

- `*p = 4;` // sets the word of memory pointed at by
  // to 4 (i.e., x = 4)
C Language Pointers

```c
int x;
int *p;
p = &x;
*p = 4;
```

```
.text
addi x2, x0, x  # x2 = &x
sw x2, p        # p = &x
addi x3, x0, 4   # 4
sw x3, 0(x2)    # *p = 4

.text

x: .word 0

p: .word 0
```
Lecture Outline

- **Binary**
- **Decimal, Binary, and Hexadecimal Integers**
- **Why Place Value Representation**
  - And why not
- **Floating Point Representation**
- **Character Representation**
- **Pointer Representation**
- **Array Representation**
- **Structure Representation**
Arrays

- Arrays are just consecutive words of memory
  - The CPU doesn’t know anything about “arrays”
- The array name is the “base address” of the array
- The index is the offset from that base address

int A[5]
Arrays

- int A[10];

.text
addi x1, x0, 4    # 4
addi x2, x0, A    # base address of A
sw x1, 3(x2)     # store at A[3]
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
Structure Representation

- `struct person {
  int id;
  int department;
}

struct person *p

- ... 

  p->department = 10

  addi x1, x0, 10
  lw x2, p
  sw x1, 1(x2)  # “department” is an offset

This defines a type. It doesn’t allocate memory.
“id” and “department” are offsets from the base of a struct person.
They have values 0 and 1 respectively.

It’s a similar idea for objects. They’re hunks of consecutive memory. Field names are offsets into those hunks.
Lecture Outline

- Binary
- Decimal, Binary, and Hexadecimal Integers
- Why Place Value Representation
  - And why not
- Floating Point Representation
- Character Representation
- Pointer Representation
- Array Representation
- Structure Representation
- Strings
String Representation

- RISC-V hardware doesn’t know anything about “string”
- Strings are inventions of the programming language
- In C, strings are zero terminated arrays of characters

```c
char str[17];
str[0] = 'T';
str[1] = 'h';
str[2] = 'i';
str[3] = 's';
str[4] = ' ';
...```

The C standard library provides functions that make this easier:

```c
strcpy( str, “This is a string”);```