Quiz 0 Review!

Part 0
Logistics
Bits ‘n’ Bytes

• A bit = 0 or 1
• A byte = 8 bits
• 0001101101111100
  – How many bits?
  – How many bytes?
  – Convert it to hexadecimal!
C and Compilers

• The reason we don’t have to write code using 0’s and 1’s!
• We use the GNU Compilers Collection (GCC)
• gcc vs. make
  – make hello
  – gcc -ggdb -std=c99 -Wall -Werror hello.c -lcs50 -lm -o hello
• Where did we specify the options listed above?
Data Types

• int
• char
• float
• double
• long
• long long
• string (same as char *)
Casting

- float age = 6.75;
  float new_age = (int) age + 1.5;
  printf("%.2f", new_age);
Casting

• float age = 6.75;
  float new_age = (int) age + 1.5;
  printf("%.2f", new_age);

• Output: 7.50
Arithmetic operators

- +, -, *, /, %

- $x = x + 1$ is the same as $x += 1$ is the same as $x++$

- Integer division: why does $(n/10) * 10$ not always equal $n$?

- What does the mod operator do?
### Ye Olde ASCII Table

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Chr</th>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Chr</th>
<th>Dec</th>
<th>Hx</th>
<th>Oct</th>
<th>Chr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>MUL (null)</td>
<td></td>
<td>32</td>
<td>020</td>
<td>040</td>
<td>&amp; #32;</td>
<td>Space</td>
<td>64</td>
<td>040</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>SOH (start of heading)</td>
<td></td>
<td>33</td>
<td>021</td>
<td>041</td>
<td>&amp; #33;</td>
<td>!</td>
<td>65</td>
<td>041</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>002</td>
<td>STX (start of text)</td>
<td></td>
<td>34</td>
<td>022</td>
<td>042</td>
<td>&amp; #34;</td>
<td>“</td>
<td>66</td>
<td>042</td>
<td>102</td>
</tr>
<tr>
<td>3</td>
<td>003</td>
<td>ETX (end of text)</td>
<td></td>
<td>35</td>
<td>023</td>
<td>043</td>
<td>&amp; #35;</td>
<td>#</td>
<td>67</td>
<td>043</td>
<td>103</td>
</tr>
<tr>
<td>4</td>
<td>004</td>
<td>EOT (end of transmission)</td>
<td></td>
<td>36</td>
<td>024</td>
<td>044</td>
<td>&amp; #36;</td>
<td>$</td>
<td>68</td>
<td>044</td>
<td>104</td>
</tr>
<tr>
<td>5</td>
<td>005</td>
<td>ENQ (enquiry)</td>
<td></td>
<td>37</td>
<td>025</td>
<td>045</td>
<td>&amp; #37;</td>
<td>%</td>
<td>69</td>
<td>045</td>
<td>105</td>
</tr>
<tr>
<td>6</td>
<td>006</td>
<td>ACK (acknowledge)</td>
<td></td>
<td>38</td>
<td>026</td>
<td>046</td>
<td>&amp; #38;</td>
<td>&amp;</td>
<td>70</td>
<td>046</td>
<td>106</td>
</tr>
<tr>
<td>7</td>
<td>007</td>
<td>BEL (bell)</td>
<td></td>
<td>39</td>
<td>027</td>
<td>047</td>
<td>&amp; #39;</td>
<td>'</td>
<td>71</td>
<td>047</td>
<td>107</td>
</tr>
<tr>
<td>8</td>
<td>010</td>
<td>BS (backspace)</td>
<td></td>
<td>40</td>
<td>028</td>
<td>050</td>
<td>&amp; #40;</td>
<td>(</td>
<td>72</td>
<td>048</td>
<td>110</td>
</tr>
<tr>
<td>9</td>
<td>011</td>
<td>TAB (horizontal tab)</td>
<td></td>
<td>41</td>
<td>029</td>
<td>051</td>
<td>&amp; #41;</td>
<td>)</td>
<td>73</td>
<td>049</td>
<td>111</td>
</tr>
<tr>
<td>10</td>
<td>012</td>
<td>LF (NL line feed, new line)</td>
<td></td>
<td>42</td>
<td>032</td>
<td>052</td>
<td>&amp; #42;</td>
<td>*</td>
<td>74</td>
<td>04A</td>
<td>112</td>
</tr>
<tr>
<td>11</td>
<td>013</td>
<td>VT (vertical tab)</td>
<td></td>
<td>43</td>
<td>033</td>
<td>053</td>
<td>&amp; #43;</td>
<td>+</td>
<td>75</td>
<td>04B</td>
<td>113</td>
</tr>
<tr>
<td>12</td>
<td>014</td>
<td>FF (NP form feed, new page)</td>
<td></td>
<td>44</td>
<td>034</td>
<td>054</td>
<td>&amp; #44;</td>
<td>,</td>
<td>76</td>
<td>04C</td>
<td>114</td>
</tr>
<tr>
<td>13</td>
<td>015</td>
<td>CR (carriage return)</td>
<td></td>
<td>45</td>
<td>035</td>
<td>055</td>
<td>&amp; #45;</td>
<td>-</td>
<td>77</td>
<td>04D</td>
<td>115</td>
</tr>
<tr>
<td>14</td>
<td>016</td>
<td>SO (shift out)</td>
<td></td>
<td>46</td>
<td>036</td>
<td>056</td>
<td>&amp; #46;</td>
<td>.</td>
<td>78</td>
<td>04E</td>
<td>116</td>
</tr>
<tr>
<td>15</td>
<td>017</td>
<td>SI (shift in)</td>
<td></td>
<td>47</td>
<td>037</td>
<td>057</td>
<td>&amp; #47;</td>
<td>/</td>
<td>79</td>
<td>04F</td>
<td>117</td>
</tr>
<tr>
<td>16</td>
<td>020</td>
<td>DLE (data link escape)</td>
<td></td>
<td>48</td>
<td>040</td>
<td>060</td>
<td>&amp; #48;</td>
<td>0</td>
<td>80</td>
<td>050</td>
<td>120</td>
</tr>
<tr>
<td>17</td>
<td>021</td>
<td>DC1 (device control 1)</td>
<td></td>
<td>49</td>
<td>041</td>
<td>061</td>
<td>&amp; #49;</td>
<td>1</td>
<td>81</td>
<td>051</td>
<td>121</td>
</tr>
<tr>
<td>18</td>
<td>022</td>
<td>DC2 (device control 2)</td>
<td></td>
<td>50</td>
<td>042</td>
<td>062</td>
<td>&amp; #50;</td>
<td>2</td>
<td>82</td>
<td>052</td>
<td>122</td>
</tr>
<tr>
<td>19</td>
<td>023</td>
<td>DC3 (device control 3)</td>
<td></td>
<td>51</td>
<td>043</td>
<td>063</td>
<td>&amp; #51;</td>
<td>3</td>
<td>83</td>
<td>053</td>
<td>123</td>
</tr>
<tr>
<td>20</td>
<td>024</td>
<td>DC4 (device control 4)</td>
<td></td>
<td>52</td>
<td>044</td>
<td>064</td>
<td>&amp; #52;</td>
<td>4</td>
<td>84</td>
<td>054</td>
<td>124</td>
</tr>
<tr>
<td>21</td>
<td>025</td>
<td>NAK (negative acknowledge)</td>
<td></td>
<td>53</td>
<td>045</td>
<td>065</td>
<td>&amp; #53;</td>
<td>5</td>
<td>85</td>
<td>055</td>
<td>125</td>
</tr>
<tr>
<td>22</td>
<td>026</td>
<td>SYN (synchronous idle)</td>
<td></td>
<td>54</td>
<td>046</td>
<td>066</td>
<td>&amp; #54;</td>
<td>6</td>
<td>86</td>
<td>056</td>
<td>126</td>
</tr>
<tr>
<td>23</td>
<td>027</td>
<td>ETB (end of trans. block)</td>
<td></td>
<td>55</td>
<td>047</td>
<td>067</td>
<td>&amp; #55;</td>
<td>7</td>
<td>87</td>
<td>057</td>
<td>127</td>
</tr>
<tr>
<td>24</td>
<td>030</td>
<td>CAN (cancel)</td>
<td></td>
<td>56</td>
<td>048</td>
<td>070</td>
<td>&amp; #56;</td>
<td>8</td>
<td>88</td>
<td>058</td>
<td>130</td>
</tr>
<tr>
<td>25</td>
<td>031</td>
<td>EM (end of medium)</td>
<td></td>
<td>57</td>
<td>049</td>
<td>071</td>
<td>&amp; #57;</td>
<td>9</td>
<td>89</td>
<td>059</td>
<td>131</td>
</tr>
<tr>
<td>26</td>
<td>032</td>
<td>SUB (substitute)</td>
<td></td>
<td>58</td>
<td>050</td>
<td>072</td>
<td>&amp; #58;</td>
<td>:</td>
<td>90</td>
<td>05A</td>
<td>132</td>
</tr>
<tr>
<td>27</td>
<td>033</td>
<td>ESC (escape)</td>
<td></td>
<td>59</td>
<td>051</td>
<td>073</td>
<td>&amp; #59;</td>
<td>;</td>
<td>91</td>
<td>05B</td>
<td>133</td>
</tr>
<tr>
<td>28</td>
<td>034</td>
<td>FS (file separator)</td>
<td></td>
<td>60</td>
<td>052</td>
<td>074</td>
<td>&amp; #60;</td>
<td>&lt;</td>
<td>92</td>
<td>05C</td>
<td>134</td>
</tr>
<tr>
<td>29</td>
<td>035</td>
<td>GS (group separator)</td>
<td></td>
<td>61</td>
<td>053</td>
<td>075</td>
<td>&amp; #61;</td>
<td>=</td>
<td>93</td>
<td>05D</td>
<td>135</td>
</tr>
<tr>
<td>30</td>
<td>036</td>
<td>RS (record separator)</td>
<td></td>
<td>62</td>
<td>054</td>
<td>076</td>
<td>&amp; #62;</td>
<td>&gt;</td>
<td>94</td>
<td>05E</td>
<td>136</td>
</tr>
<tr>
<td>31</td>
<td>037</td>
<td>US (unit separator)</td>
<td></td>
<td>63</td>
<td>055</td>
<td>077</td>
<td>&amp; #63;</td>
<td>?</td>
<td>95</td>
<td>05F</td>
<td>137</td>
</tr>
</tbody>
</table>

Source: [www.LookupTables.com](http://www.LookupTables.com)
ASCII Math

• ‘P’ + 1?

• ‘5’ ≠ 5
  – How would we transform one to the other?
ASCII Math

- ‘P’ + 1?
  - ‘Q’

- ‘5’ ≠ 5
  - How would we transform one to the other?
    - ‘5’ – ‘0’ = 5
    - 5 + ‘0’ = ‘5’
A simple C program...

```c
#include <stdio.h>
#include <cs50.h>

#define LIMIT 100

int main(int argc, char* argv[]) {
    int x = GetInt();
    if (x >= LIMIT) {
        printf("That number is too big!\n");
        return 1;
    } else {
        int y = x*x;
        printf("The square of %d is %d\n", x, y);
    }
    return 0;
}
```
...with lots of elements!

```c
#include <stdio.h>
#include <cs50.h>

#define LIMIT 100

int main(int argc, char* argv[]) {
    int x;
    x = GetInt();
    if (x >= LIMIT) {
        printf("That number is too big!\n");
        return 1;
    }
    printf("The square of %d is %d\n", x, x*x);
    return 0;
}
```
Loops

When would we use each of the following?

• for?

• while?

• do while?
Loops

When would we use each of the following?

• **for?**
  – We already know how many times we want to iterate through our loop (could also use while)

• **while?**
  – We’re not sure how many times we want our loop to run, but there is some condition that needs to be true for our loop to keep running

• **do while?**
  – Similar to while, but we want the code in our loop to run **at least once**
Loops

Each loop needs an initialization, a condition, and an update.

• for?
  – for (initialization; condition; update)
    {
      // do this
    }

• while?
  – initialization
    while (condition)
    {
      // do this
      // update
    }

• do while?
  – initialization
    do
    {
      // do this
      // update
    }
    while (condition);
Look, a function!

```c
void
double_array(int nums[], int length)
{
    for (int i = 0; i < length; i++)
        nums[i] *= 2;
    printf("Your array has been doubled!\n");
}
```
Today

- scope
- arrays
- command-line arguments
- searching
- sorting
- asymptotic notation
- recursion
Variable scope

- Global variables: accessible by all functions
  - defined outside of `main`
- Local variables: accessible by a single block
  - defined within a block, only accessible in that block
Variable scope

```cpp
int x = 5;
int f() {
    int y = 6;
    x++;
}
int g() {
    int y = 8;
    x--;
}
```
Arrays

- list of elements of the same type
- elements accessed by their **index** (aka position)
  - index starts at 0!
- int array[3] = {1, 2, 3};
- array[1] = 4;
Multi-dimensional Arrays

- can also have arrays of arrays!
- multi-dimensional array creates a grid instead of a list
- needs multiple indices: int grid[3][5];
  - 3 rows, 5 columns
int grid[2][3] = {{1, 2, 3}, {4, 5, 6}};
grid[1][2] = 6;

<table>
<thead>
<tr>
<th>Rows</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Passing Arrays to Functions

- ints, chars, floats, etc. are passed by **value**
  - contents CANNOT be changed by the function they’re passed to (unless we use pointers!)
- arrays (of any type) are passed by **reference**
  - contents CAN be changed by the function they’re passed to
main

- main is a **function** that can take 2 arguments
  - `argc`: number of arguments given
  - `argv[]`: array of arguments
Arguments

- ./this is cs 50
  - argc == 4
  - argv[0] == "./this"
  - argv[1] == "is"
  - argv[2] == "cs"
  - argv[3] == "50"

- "50" != 50;
  - atoi("50") == 50;
Big $O$

- $O$: worst-case running time
  - given the worst possible scenario, how fast can we solve a problem?
    - e.g. array is in descending order, we want it in ascending order
  - upper bound on runtime
Omega

- $\Omega$: best-case running time
  - given the best possible scenario, how fast can we solve a problem?
    - e.g. array is already sorted
  - lower bound on runtime
Common Running Times

- in ascending order:
  - $O(1)$: constant
  - $O(\log n)$: logarithmic
  - $O(n)$: linear
  - $O(n \log n)$: linearithmic
  - $O(n^c)$: polynomial
  - $O(c^n)$: exponential
  - $O(n!)$: factorial
Comparing Running Times

- $O(n)$, $O(2n)$, and $O(5n + 3)$ are all asymptotically equivalent: $O(n)$
  - constants drop out, because $n$ dominates
- similarly, $O(n^3 + 2n^2) = O(n^3)$
  - $n^3$ dominates $n^2$
- however, $O(n^3) > O(n^2)$
  - 2 and 3 are not constants here, they’re exponents
Linear Search

- implementation: iterate through each element of the list, looking for it
- runtime: $O(n), \Omega(1)$
- does not require list to be sorted
Binary Search

- implementation: keep looking at middle elements
  - start at middle of list
  - if too high, forget right half and look at middle of left half
  - if too low, forget left half and look at middle of right half

- runtime: $O(\log n)$, $\Omega(1)$

- requires list to be sorted
while length of list > 0
    look at middle of list
    if number found, return true
    else if number is too high, only consider left half of list
    else if number is too low, only consider right half of list
return false
# Binary Search

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>61</td>
<td>121</td>
<td>124</td>
<td>143</td>
<td>161</td>
<td>164</td>
<td>171</td>
<td>175</td>
</tr>
</tbody>
</table>
Binary Search

164  171  175  182
Binary Search
Binary Search
Bubble Sort

- implementation: if adjacent elements are out of place, switch them
  - repeat until no swaps are made
- runtime: $O(n^2)$, $\Omega(n)$
do
    swapped = false
    for i = 0 to n - 2
        if array[i] > array[i + 1]
            swap array[i] and array[i + 1]
            swapped = true
    while elements have been swapped
Bubble Sort
Bubble Sort
Bubble Sort
Bubble Sort
Bubble Sort
Bubble Sort

0 1 5 4 6
Bubble Sort

0 1 5 4 6
Bubble Sort
Bubble Sort
Selection Sort

- implementation: start at beginning of list, find smallest element
  - swap first element with smallest element
  - go to second element, treat that as the new first element, continue
    - because everything to the left is already sorted
- runtime: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$
Selection Sort

```
for i = 0 to n - 1
    min = i
    for j = i + 1 to n
        if array[j] < array[min]
            min = j
    if array[min] != array[i]
        swap array[min] and array[i]
```
Selection Sort
Selection Sort

0 5 1 6 4
Selection Sort

0  1  5  6  4
Selection Sort
Selection Sort

0 1 4 5 6
Recursion

- base case: when function should stop calling itself
  - without a base case, function would call itself forever!
- recursive case: function calls itself, probably using different arguments
Recursion

```c
int factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}
```
Recursion and the Stack

factorial(4)
main
Recursion and the Stack

<table>
<thead>
<tr>
<th>factorial(3)</th>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(4)</td>
<td></td>
</tr>
</tbody>
</table>
Recursion and the Stack

<table>
<thead>
<tr>
<th>factorial(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>factorial(3)</td>
</tr>
<tr>
<td>factorial(4)</td>
</tr>
<tr>
<td>main</td>
</tr>
</tbody>
</table>
Recursion and the Stack

```
factorial(1)
factorial(2)
factorial(3)
factorial(4)
main
```
This is CS50. (Quiz 0 Review)  o hai!

Joseph Ong
Merge Sort

mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;

| 50 | 3  | 42 | 1337 | 15 |
Merge Sort

mSort (list of n numbers)
  if n < 2
    return;
  else
    mSort left half;
    mSort right half;
    merge sorted halves;

50  3  42  1337  15
Merge Sort

mSort (list of n numbers)
if n < 2
  return;
else
  mSort left half;
  mSort right half;
  merge sorted halves;
Merge Sort

\[ \text{mSort (list of n numbers)} \]

if \( n < 2 \)

\[ \text{return;} \]

else

\[ \text{mSort left half;} \]
\[ \text{mSort right half;} \]
\[ \text{merge sorted halves;} \]
Merge Sort

mSort (list of n numbers)
  if n < 2
      return;
  else
      mSort left half;
      mSort right half;
      merge sorted halves;
Merge Sort

\[ mSort \text{ (list of } n \text{ numbers)} \]

\[ \text{if } n < 2 \]
\[ \quad \text{return;} \]
\[ \text{else} \]
\[ \quad mSort \text{ left half;} \]
\[ \quad mSort \text{ right half;} \]
\[ \quad \text{merge sorted halves;} \]
Merge Sort

mSort (list of n numbers)
  if n < 2
    return;
  else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
# Merge Sort

mSort (list of n numbers)

\[
\begin{align*}
\text{if } n &< 2 \\
& \quad \text{return;}
\end{align*}
\]

\[
\begin{align*}
\text{else} & \\
& \quad \text{mSort left half;}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{mSort right half;}
& \quad \text{merge sorted halves;}
\end{align*}
\]

<table>
<thead>
<tr>
<th>50</th>
<th>3</th>
<th>42</th>
<th>1337</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>3</td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
   return;
else
   mSort left half;
   mSort right half;
   merge sorted halves;
Merge Sort

mSort (list of n numbers)
  if n < 2
      return;
  else
      mSort left half;
      mSort right half;
      merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
→ merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
  return;
else
  mSort left half;
  mSort right half;
  merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
  return;
else
  mSort left half;
  mSort right half;
  merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
   return;
else
   mSort left half;
   mSort right half;
   merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
   return;
else
   mSort left half;
   mSort right half;
   merge sorted halves;
Merge Sort

mSort (list of n numbers)
  if n < 2
    return;
  else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
merge sorted halves;
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
Merge Sort

mSort (list of \( n \) numbers)
if \( n < 2 \)
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;
**Merge Sort**

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
merge sorted halves;

```
50  3  42  1337  15
```

```
50  3  42
```

```
1337  15
```

```
50  3
```

```
1337  15
```

```
15  1337
```

```
3  50  42
```

```
3  42  50
```

```
3  15  42  50
```
Merge Sort

mSort (list of n numbers)
if n < 2
    return;
else
    mSort left half;
    mSort right half;
    merge sorted halves;

MERGE ALL THE HALVES
Merge Sort

mSort (list of n numbers)
  if n < 2
    return;
  else
    mSort left half;
    mSort right half;
    merge sorted halves;

$O(n \log n)$

$\Omega(n \log n)$
CS50: Quiz 0

Memory, Stack, Heap
### Memory Layout

<table>
<thead>
<tr>
<th>Memory Area</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The program itself</td>
</tr>
<tr>
<td>initialized global data</td>
<td>initialized globals</td>
</tr>
<tr>
<td>uninitialized global data</td>
<td>declared, but not initialized, globals</td>
</tr>
<tr>
<td>heap</td>
<td>Memory allocated using malloc()</td>
</tr>
<tr>
<td>stack</td>
<td>local variables and parameters of functions</td>
</tr>
<tr>
<td>environment variables</td>
<td>special variables, like the username of person running program</td>
</tr>
</tbody>
</table>

- **fon()**
- **bar()**
- **baz()**
- **main()**

**Special Variables:**
- `username`
What are pointers?

They are data types that refer to another location in memory, where other data is stored.

In this case, ptr “references” 50.

Just fyi, on 32-bit systems, pointers take up 32 bits, or 4 bytes, of space, just like an int does.
Dynamic Memory Allocation

Recall, local variables are allocated on the stack, and we can’t access them outside the scope of the functions or loops they belong to.

So, what dynamic memory allocation lets us do is hold on to data for the entire duration of the program.

This is done by:

1) Allocating that data in a permanent space on the heap.
2) Giving us a pointer to that location in memory.
malloc()

```c
int main(void)
{
    int x = 5;
    int *ptr = giveMeThreeInts();
    ptr[0] = 1;
    ptr[1] = 2;
    ptr[2] = 3;
}

int *giveMeThreeInts(void)
{
    int *temp = malloc(sizeof(int) * 3);
    return temp;
}
```
int main(void)
{
    int x = 5;
    int *ptr = giveMeThreeInts();
    ptr[0] = 1;
    ptr[1] = 2;
    ptr[2] = 3;
}

int *giveMeThreeInts(void)
{
    int *temp = malloc(sizeof(int) * 3);
    return temp;
}
```c
int main(void)
{
    int x = 5;
    int *ptr = giveMeThreeInts();
    ptr[0] = 1;
    ptr[1] = 2;
    ptr[2] = 3;
}

int *giveMeThreeInts(void)
{
    int *temp = malloc(sizeof(int) * 3);
    return temp;
}
```
main()

```c
int main(void)
{
    int x = 5;
    int *ptr = giveMeThreeInts();
    ptr[0] = 1;
    ptr[1] = 2;
    ptr[2] = 3;
}
```

```c
int *
giveMeThreeInts(void)
{
    int *temp = malloc(sizeof(int) * 3);
    return temp;
}
```
```c
int main(void)
{
    int x = 5;
    int *ptr = giveMeThreeInts();

    ptr[0] = 1;
    ptr[1] = 2;
    ptr[2] = 3;
}

int *
giveMeThreeInts(void)
{
    int *temp = malloc(sizeof(int) * 3);
    return temp;
}
```
CS50: Quiz 0

Pointers
Recall Binky

```c
int
main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));
    if (ptr == NULL)
        return 1;
    *ptr = 1;
    free(ptr);
    int x = 5;
    ptr = &x;
    return 0;
}
```
int main(void) {
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));
    if (ptr == NULL)
        return 1;
    *ptr = 1;
    free(ptr);
    int x = 5;
    ptr = &x;
    return 0;
}
int main(void)
{
    int *ptr;
    ptr = malloc(sizeof(int));
    if (ptr == NULL)    // did malloc work?
        return 1;
    *ptr = 1;
    free(ptr);

    int x = 5;
    ptr = &x;

    return 0;
}
Recall Binky

```c
int main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));
    if (ptr == NULL)
        return 1;

    // sets contents of memory to 1
    *ptr = 1;
    free(ptr);

    int x = 5;
    ptr = &x;

    return 0;
}
```
Recall Binky

```c
int main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));
    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);      // frees up memory from heap

    int x = 5;
    ptr = &x;

    return 0;
}
```
Recall Binky

```c
int main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));
    if (ptr == NULL)
        return 1;
    *ptr = 1;
    free(ptr);

    int x = 5;
    ptr = &x;
    return 0;
}
```

```plaintext
int* ptr
```
main(void) {
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));
    if (ptr == NULL)
        return 1;
    *ptr = 1;
    free(ptr);
    int x = 5;
    int* ptr = &x; // gets address of x, points to it.
    return 0;
}
```c
int main(void) {
    int *ptr = malloc(sizeof(int) * 3);
    *ptr = 1;
    *(ptr + 1) = 2;  // one int down from ptr
    *(ptr + 2) = 3;

    printf("%d", *(ptr + 1));

    ptr++;
    printf("%d", *(ptr + 1));

    ptr--;
    free(ptr);
}
```
int main(void) {
    int *ptr = malloc(sizeof(int) * 3);

    *ptr = 1;
    *(ptr + 1) = 2;
    *(ptr + 2) = 3;

    printf("%d", *(ptr + 1));    // prints out 2
    ptr++;
    printf("%d", *(ptr + 1));
    ptr--;
    free(ptr);
}
`int *ptr = malloc(sizeof(int) * 3);`
`*ptr = 1; *(ptr + 1) = 2; *(ptr + 2) = 3;`
`printf("%d", *(ptr + 1));
ptr++; // changes ptr permanently.
printf("%d", *(ptr + 1)); // now prints out 3
ptr--; // move back to original ptr location before freeing
free(ptr);`
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6; i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
int main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6; i++)
    {
        *(ptr + i) = 'z'; // i chars down from ptr
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```c
int main(void) {
    char *ptr = malloc(sizeof(char) * 7);
    for (int i = 0; i < 6; i++) {
        *(ptr + i) = 'z';
    }
    ptr[6] = '\0'; // shorthand for *(ptr + 6)
    while (*ptr != '\0') {
        printf("%c", *ptr);
        ptr++;
    }
    ptr -= 6;
    free(ptr);
}
```
int main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6; i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6; i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("\%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```c
int main(void)
{
    char *ptr = malloc(sizeof(char) * 7);
    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = 'z';
    }
    ptr[6] = '\0';
    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }
    ptr -= 6;
    free(ptr);
}
```
int main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6; i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```c
int main(void) {
    char *ptr = malloc(sizeof(char) * 7);
    for (int i = 0; i < 6; i++) {
        *(ptr + i) = 'z';
    }
    ptr[6] = '\0';
    while (*ptr != '\0') {
        printf("%c", *ptr);
        ptr++;
    }
    ptr -= 6;
    free(ptr);
}
```
```c
int main(void) {
    char *ptr = malloc(sizeof(char) * 7);
    for (int i = 0; i < 6; i++) {
        *(ptr + i) = 'z';
    }
    ptr[6] = '\0';
    while (*ptr != '\0') {
        printf("%c", *ptr);
        ptr++;
    }
    ptr -= 6;
    free(ptr);
}
```
int main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6; i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')  // !!!
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6; // move back to original memory location before freeing
    free(ptr);
}
```c
int main(void) {
    char *ptr = malloc(sizeof(char) * 7);
    for (int i = 0; i < 6; i++) {
        *(ptr + i) = 'z';
    }
    ptr[6] = '\0';
    printf("%s", ptr);  // no *
    free(ptr);
}
```
int main(void) {
    // oops, pretty sure we don’t have that much memory
    // malloc will fail, returning a NULL pointer
    int *ptr = malloc(sizeof(int) * 2147483647);

    // oops, we forgot to check if it was NULL
    *ptr = 1;

    return 0;
}
int main(void) {
    // oops, pretty sure we don't have that much memory
    // malloc will fail, returning a NULL pointer
    int *ptr = malloc(sizeof(int) * 2147483647);

    // oops, we forgot to check if it was NULL
    *ptr = 1;  // oops, we just died x.x, aka "dereferencing a null pointer"
    return 0;
}
int main(void) {
    // oops, pretty sure we don’t have that much memory
    // malloc will fail, returning a NULL pointer
    int *ptr = malloc(sizeof(int) * 2147483647);

    // solution, check if null, and exit the program
    if (ptr == NULL)
        return 1;

    *ptr = 1;  // no longer dereferenced if ptr is NULL

    return 0;
}
Okay, human.

Huh?

Before you hit 'compile', listen up.

You know when you're falling asleep, and you imagine yourself walking or something,

And suddenly you misstep, stumble, and jolt awake?

Yeah!

Well, that's what a segfault feels like.

Double-check your damn pointers, okay?
```c
#define cs50isAwesome 1

int main(void) {
    while (cs50isAwesome) {
        int *ptr = malloc(sizeof(int));

        if (ptr == NULL)
            return 1;

        *ptr = 1;    // oops, we forgot to free memory, we’ll get a memory leak!
    }

    return 0;
}
```
#define cs50isAwesome 1

int main(void) {
    while (cs50isAwesome) {
        int *ptr = malloc(sizeof(int));

        if (ptr == NULL)
            return 1;

        *ptr = 1; // oops, we forgot to free memory, we’ll get a memory leak!
    }

    return 0;
}
Memory Leaks

```c
#define cs50isAwesome 1

int main(void)
{
    while (cs50isAwesome)
    {
        int *ptr = malloc(sizeof(int));

        if (ptr == NULL)
            return 1;

        *ptr = 1;
        free(ptr);     // fix’t!
    }

    return 0;
}
```

![Memory usage screenshot](image-url)
Freeing Twice (or n > 1 times)

```c
int main(void)
{
    int *ptr = malloc(sizeof(int));
    if (ptr == NULL)
        return 1;
    *ptr = 1;
    free(ptr);
    *ptr = 1;
    free(ptr);  // oops, we freed something we already freed earlier.
    return 0;
}
```
Freeing Twice (or n > 1 times)

```c
int main(void)
{
    int *ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    // fix't!

    return 0;
}
```
Failure to use sizeof()

```c
int main(void) {
    // wants to malloc 2 ints. 8 bytes? Right?
    int *ptr = malloc(8);

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    return 0;
}
```
int main(void)
{
    // actually, an int isn’t necessarily 4 bytes on all systems.
    // this is safer and is more compatible with different architectures.
    int *ptr = malloc(sizeof(int) * 2);

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    return 0;
}
Structs

A struct is a container that can hold and organize meaningfully related variables of different types.

For example, let’s say we want to make a collection of variables to represent a Sudoku board! (this is good for Pokemon too)

typedef struct
{
    int board[9][9];
    char *level;
    int x, y;
    int timeSpent;
    int totalMoves;
} sudokuBoard;

int main(void)
{
    sudokuBoard board;

    board.board = {{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, ...};
    board.level = "n00b";
    board.x = 0;
    board.y = 0;
    board.timeSpent = 0;
    int totalMoves = 0;

    // do stuff with board in rest of program
};
typedef struct {
  int board[9][9];
  char *level;
  int x, y;
  int timeSpent;
  int totalMoves;
} sBoard;

int main(void) {
  sBoard boardA = {{{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, ...},
                   "n00b", 0, 0, 0, 0};

  sBoard boardB = {{{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, ...},
                   "l33t", 0, 0, 0, 0};

  // do stuff with boards in rest of program
}
**Structs**

Finally, if we have a pointer to a struct, we can access its fields with the `->` operator.

```c
typedef struct
{
    int board[9][9];
    char *level;
    int x, y;
    int timeSpent;
    int totalMoves;
} sBoard;

int main(void)
{
    sBoard* board = malloc(sizeof(sBoard));

    (*board).x = 0;    // deref board, then change x field
    board->x = 0;      // does same thing as above
}
```
# GDB Commands

These are the ones you’ll need to know!

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run arg1, arg2, ...</td>
<td>runs program with command line arguments</td>
</tr>
<tr>
<td>print x</td>
<td>prints out the value of a variable named x in stack frame</td>
</tr>
<tr>
<td>break function_name</td>
<td>sets a breakpoint at a function called function_name</td>
</tr>
<tr>
<td>break line_number</td>
<td>sets a breakpoint at a line of your code</td>
</tr>
<tr>
<td>frame n</td>
<td>gives you information about the nth stack frame</td>
</tr>
<tr>
<td>backtrace n</td>
<td>tells you the last n stack frames prior to current point in program</td>
</tr>
<tr>
<td>next</td>
<td>moves forward one line in the current execution of code</td>
</tr>
<tr>
<td>step</td>
<td>moves forward one line, stepping into a function where applicable</td>
</tr>
<tr>
<td>continue</td>
<td>moves forward in the program until the next breakpoint</td>
</tr>
</tbody>
</table>
typedef struct {
    int pokedexNo;
    int level;
    char* owner;
    char* pokemonType;
    char* nickName;
    int stats[6];
    char* moveset[4];
    ...
} pokemon;

Questions?