Google’s AlphaGo AI secretively won more than 50 straight games against the world’s top Go players

Ke, the reigning top-ranked Go player, has acknowledged that human beings are no match for robots in the complex board game, after he lost three games to an AI that mysteriously popped up online in recent days.

The AI turned out to be AlphaGo in disguise.

Ke [wrote] “humans have evolved in games in thousands of years—but computers now tell us humans are all wrong. From now on, we Go players will need to work with the computers to enter a new area and a new level.”

Administrivia

- Homework 1 due today
  - Reminder: you get an unlimited number of tries!
- Lab 0 due Friday (1/13)
  - Credit/no credit assignment
  - You should be able to answer these questions much more thoroughly by the end of this class!
- Optional section tomorrow @ 4:30pm in MGH 241
  - C and pointers
- Justin’s Friday office hours in a locked lab (CSE 003)
  - I’ll set up near the door, so just knock!
Review Questions

1) If the word size of a machine is 64-bits, which of the following is usually true? (pick all that apply)
   a) 64 bits is the size of a pointer $\text{T}$
   b) 64 bits is the size of an integer $\text{F}$
   c) 64 bits is the width of a register $\text{T}$

2) (True/False) By looking at the bits stored in memory, I can tell if a particular 4-bytes is being used to represent an integer, floating point number, or instruction.

3) If the size of a pointer on a machine is 6 bits, the address space is how many bytes? $64 \text{ bytes}$
Memory, Data, and Addressing

- Representing information as bits and bytes
- Organizing and addressing data in memory
- **Manipulating data in memory using C**
- Boolean algebra and bit-level manipulations
Addresses and Pointers in C

- **&** = “address of” operator
- *** = “value at address” or “dereference” operator

```
int* ptr;
int x = 5;
int y = 2;
ptr = &x;
y = 1 + *ptr;
```

* is also used with variable declarations

What is *(&y)?

returns value of y
Assignment in C

- A variable is represented by a memory location
- Declaration ≠ initialization (initially holds “garbage”)
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18
### Assignment in C

- A variable is represented by a memory location
- Declaration ≠ initialization (initially holds “garbage”)
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18

#### 32-bit example (pointers are 32-bits wide)

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>00 01 29 F3</td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>01 00 00 00</td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td></td>
</tr>
</tbody>
</table>
```

- Little-endian
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a memory location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- int x, y;
- x = 0;
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a memory location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- int x, y;
- x = 0;
- y = 0x3CD02700;

32-bit example (pointers are 32-bits wide)

& = “address of”
* = “dereference”

little endian!
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a memory location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

```
int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
```

- Get value at `y`, add 3, store in `x`
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a memory *location*
  - RHS must evaluate to a *value* (could be an address)
  - Store RHS value at LHS location

```
int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
```

- Get value at `y`, add 3, store in `x`

```
int* z;
```
- `z` is at address 0x20

32-bit example (pointers are 32-bits wide)

\& = “address of”
* = “dereference”

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>00</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>DE</td>
<td>AD</td>
<td>BE</td>
<td>EF</td>
</tr>
<tr>
<td>initial value is whatever bits were already there! (“garbage”)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a memory location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - Get value at y, add 3, store in x
- int* z = &y + 3;
  - Get address of y, “add 3”, store in z

32-bit example (pointers are 32-bits wide)

& = “address of”
* = “dereference”

# Pointer arithmetic
Pointer Arithmetic

- Pointer arithmetic is scaled by the size of target type
  - In this example, \texttt{sizeof(int)} = 4

- \texttt{int* z = \&y + 3;}
  - Get address of \texttt{y}, add 3*\texttt{sizeof(int)}, store in \texttt{z}
  - \texttt{\&y = 0x18 = 1*16^1 + 8*16^0 = 24}
  - 24 + 3*(4) = 36 = 2*16^1 + 4*16^0 = 0x24

- Pointer arithmetic can be dangerous!
  - Can easily lead to bad memory accesses
  - Be careful with data types and \textit{casting}
Assignment in C

- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - Get value at y, add 3, store in x
- int* z = &y + 3;
  - Get address of y, add 12, store in z
- *z = y;
  - What does this do?

32-bit example (pointers are 32-bits wide)

& = “address of”
* = “dereference”
Assignment in C

- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - Get value at y, add 3, store in x
- int* z = &y + 3;
  - Get address of y, add 12, store in z
- *z = y;
  - Get value of y, put in address stored in z

32-bit example (pointers are 32-bits wide)

\& = “address of”
* = “dereference”
Arrays in C

Declaration: `int a[6];`

- **Element type**: `int`
- **Name**: `a`
- **Number of elements**: 6

Arrays are adjacent locations in memory storing the same type of data object.

- `a` is a name for the array’s address.

64-bit example (pointers are 64-bits wide)

Arrays in C

Declaration: `int a[6];`

Indexing: 
- `a[0] = 0x015f;`
- `a[5] = a[0];`

Arrays are adjacent locations in memory storing the same type of data object.

- `a` is a name for the array’s address.
- The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes.
Arrays in C

Declaration: `int a[6];`

Indexing:  
- `a[0] = 0x015f;`
- `a[5] = a[0];`

No bounds  
- `a[6] = 0xBAD;`

checking:  
- `a[-1] = 0xBAD;`

Arrays are adjacent locations in memory storing the same type of data object

- `a` is a name for the array’s address
- The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes

```
| 0x0 | 0x08 | 0x1 | 0x9 | 0x2 | 0xA | 0x3 | 0xB | 0x4 | 0xC | 0x5 | 0xD | 0x6 | 0xE | 0x7 | 0xF |
|-----|------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| AD  | OB   | 00  | 00  | 5F  | 01   | 00  | 00  |     |     |     |     |     |     |     |     |
| AD  | OB   | 00  | 00  |     |     |     |     | 5F  | 01   | 00  | 00  |     |     |     |     |
|     |      |     |     | AD  | OB   | 00  | 00  |     |     |     |     |     |     |     |     |
|     |      |     |     |     |      |     |     |     |     |     |     |     |     |     |     |
```

0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48
Arrays in C

Declaration: `int a[6];`

Indexing: `a[0] = 0x015f;`
`a[5] = a[0];`

No bounds `a[6] = 0xBAD;`
checking: `a[-1] = 0xBAD;`

Pointers:
`int* p;`
`p = a;`
`p = &a[0];`
`*p = 0xA;`

Arrays are adjacent locations in memory storing the same type of data object
`a` is a name for the array’s address
The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes
Arrays in C

Declaration: \( \text{int } a[6]; \)

Indexing: \( a[0] = 0x015f; \)
\( a[5] = a[0]; \)

No bounds \( a[6] = 0xBAD; \)
checking: \( a[-1] = 0xBAD; \)

Pointers:
\[ \text{int}\ast p; \]
\[ p = a; \]
\[ p = \&a[0]; \]
\[ *p = 0xA; \]

The address of \( a[i] \) is the address of \( a[0] \) plus \( i \) times the element size in bytes

Arrays are adjacent locations in memory storing the same type of data object
\( a \) is a name for the array’s address

Array indexing = address arithmetic (both scaled by the size of the type)

Equivalent:
\[ p[1] = 0xB; \]
\[ *(p+1) = 0xB; \]

Pointer arithmetic:
\[ 0x10 + 1 \rightarrow 0x14 \]
\[ p = p + 2; \]
\[ 0x10 + 2 \rightarrow 0x18 \]
Arrays in C

Declaration:  int a[6];

Indexing:     a[0] = 0x015f;
               a[5] = a[0];

No bounds    a[6] = 0xBAD;
checking:     a[-1] = 0xBAD;

Pointers:     int* p;
               p = a;
               p = &a[0];
               *p = 0xA;

array indexing = address arithmetic
(both scaled by the size of the type)

equivalent
p[1] = 0xB;
*(p+1) = 0xB;

store at addr 0x18
p = a[1] + 1;

The address of a[i] is the address of a[0] plus i times the element size in bytes

Arrays are adjacent locations in memory storing the same type of data object

a is a name for the array’s address
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5?

- Vote at http://PollEv.com/justinh

```
1 void main() {
2    int a[] = {5,10};
3    int *p = a;
4    p = p + 1;
5    *p = *p + 1;
6 }
```

<table>
<thead>
<tr>
<th>Address (decimal)</th>
<th>Data (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>5</td>
</tr>
<tr>
<td>a[1]</td>
<td>10</td>
</tr>
<tr>
<td>p</td>
<td>100</td>
</tr>
</tbody>
</table>

- (A) | 101 | 10 | 5 | 10 | then | 101 | 11 | 5 | 11
- (B) | 104 | 10 | 5 | 10 | then | 104 | 11 | 5 | 11
- (C) | 100 | 6  | 6 | 10 | then | 101 | 6  | 6 | 10
- (D) | 100 | 6  | 6 | 10 | then | 104 | 6  | 6 | 10
Representing strings

- C-style string stored as an array of bytes (char *)
  - Elements are one-byte ASCII codes for each character
  - No “String” keyword, unlike Java

ASCII: American Standard Code for Information Interchange
Null-Terminated Strings

- **Example:** “Donald Trump” stored as a 13-byte array

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>0x44</td>
<td>Donald</td>
</tr>
<tr>
<td>111</td>
<td>0x6F</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>0x6E</td>
<td>l</td>
</tr>
<tr>
<td>97</td>
<td>0x61</td>
<td>d</td>
</tr>
<tr>
<td>108</td>
<td>0x6C</td>
<td>a</td>
</tr>
<tr>
<td>100</td>
<td>0x64</td>
<td>l</td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>0x54</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>0x72</td>
<td>u</td>
</tr>
<tr>
<td>117</td>
<td>0x75</td>
<td>m</td>
</tr>
<tr>
<td>109</td>
<td>0x6D</td>
<td>p</td>
</tr>
<tr>
<td>112</td>
<td>0x70</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x00</td>
<td></td>
</tr>
</tbody>
</table>

- Last character followed by a 0 byte (‘\0’) (a.k.a. “null terminator”)
  - Must take into account when allocating space in memory
  - Note that ‘0’ ≠ ‘\0’ (i.e. character 0 has non-zero value)

- How do we compute the length of a string?
  - Traverse array until null terminator encountered
Endianness and Strings

char s[6] = "12345";

- Byte ordering (endianness) is not an issue for 1-byte values
  - The whole array does not constitute a single value
  - Individual elements are values; chars are single bytes

C (char = 1 byte)

IA32, x86-64 (little-endian)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>31</td>
<td>31</td>
<td>0x00</td>
<td>'1'</td>
</tr>
<tr>
<td>0x01</td>
<td>32</td>
<td>32</td>
<td>0x01</td>
<td>'2'</td>
</tr>
<tr>
<td>0x02</td>
<td>33</td>
<td>33</td>
<td>0x02</td>
<td>'3'</td>
</tr>
<tr>
<td>0x03</td>
<td>34</td>
<td>34</td>
<td>0x03</td>
<td>'4'</td>
</tr>
<tr>
<td>0x04</td>
<td>35</td>
<td>35</td>
<td>0x04</td>
<td>'5'</td>
</tr>
<tr>
<td>0x05</td>
<td>00</td>
<td>00</td>
<td>0x05</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

SPARC (big-endian)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>31</td>
<td>0x00</td>
<td>'1'</td>
<td></td>
</tr>
<tr>
<td>0x01</td>
<td>32</td>
<td>0x01</td>
<td>'2'</td>
<td></td>
</tr>
<tr>
<td>0x02</td>
<td>33</td>
<td>0x02</td>
<td>'3'</td>
<td></td>
</tr>
<tr>
<td>0x03</td>
<td>34</td>
<td>0x03</td>
<td>'4'</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>35</td>
<td>0x04</td>
<td>'5'</td>
<td></td>
</tr>
<tr>
<td>0x05</td>
<td>00</td>
<td>0x05</td>
<td>'\0'</td>
<td></td>
</tr>
</tbody>
</table>
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a *byte array* by **casting** it to `char`
  - C has **unchecked casts** !! DANGER !!

```c
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

**printf directives:**
- `%p` Print pointer
- `	` Tab
- `%x` Print value as hex
- `
` New line

26
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a *byte array* by *casting* it to *char*
  - C has **unchecked casts**  (!! DANGER !!)

```c
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

```c
void show_int(int x) {
    show_bytes((char*) &x, sizeof(int));
}
```
**show_bytes Execution Example**

```c
int a = 12345; // 0x00003039
printf("int a = 12345;\n");
show_int(a); // show_bytes((char *) &a, sizeof(int));
```

- **Result (Linux x86-64):**
  - **Note:** The addresses will change on each run (try it!), but fall in same general range

```c
int a = 12345;
0x7fffb7f71dbc 0x39
0x7fffb7f71dbd 0x30
0x7fffb7f71dbe 0x00
0x7fffb7f71dbf 0x00
```

- Little endian
Summary

- Assignment in C results in value being put in memory location
- Pointer is a C representation of a data address
  - `& = “address of” operator`
  - `* = “value at address” or “dereference” operator`
- Pointer arithmetic scales by size of target type
  - Convenient when accessing array-like structures in memory
  - Be careful when using – particularly when casting variables
- Arrays are adjacent locations in memory storing the same type of data object
  - Strings are null-terminated arrays of characters (ASCII)