Memory, Data, & Addressing II
CSE 351 Autumn 2022

Instructor:
Justin Hsia

Teaching Assistants:
Angela Xu
Arjun Narendra
Armin Magness
Assaf Vayner
Carrie Hu
Clare Edmonds
David Dai
Dominick Ta
Effie Zheng
James Froelich
Jenny Peng
Kristina Lansang
Paul Stevans
Renee Ruan
Vincent Xiao

http://xkcd.com/138/
Relevant Course Information

❖ Lab 0 due today @ 11:59 pm
  - *You will revisit the concepts from this program in future labs!*

❖ hw2 due Wednesday, hw3 due Friday
  - Autograded, unlimited tries, no late submissions

❖ Lab 1a released today, due next Monday (10/10)
  - Pointers in C
  - Last submission graded, can optionally work with a partner
    - One student submits, then add their partner to the submission
  - Short answer “synthesis questions” for after the lab
Late Days

- You are given 5 late day tokens for the whole quarter
  - Tokens can only apply to Labs
  - No benefit to having leftover tokens
- Count lateness in *days* (even if just by a second)
  - *Special:* weekends count as *one day*
  - No submissions accepted more than two days late
- Late penalty is 10% deduction of your score per day
  - Only late labs are eligible for penalties
  - Penalties applied at end of quarter to *maximize* your grade
- Use at own risk – don’t want to fall too far behind
  - Intended to allow for unexpected circumstances
Reading Review

❖ Terminology:
  - address-of operator (&), dereference operator (*), NULL
  - box-and-arrow memory diagrams
  - pointer arithmetic, arrays
  - C string, null character, string literal

❖ Questions from the Reading?
Review Questions

- int x = 351;
  char* p = &x;
  int ar[3];

- How much space does the variable p take up?
  A. 1 byte
  B. 2 bytes
  C. 4 bytes
  D. 8 bytes

- Which of the following expressions evaluate to an address?
  A. x + 10 → int
  B. p + 10 → char *
  C. &x + 10 → int *
  D. *(&p) → char *
  E. ar[1] → int
  F. &ar[2] → int *
Pointer Operators

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

❖ Operator confusion

▪ The pointer operators are unary (i.e., take 1 operand)
▪ These operators both have binary forms
  • x & y is bitwise AND (we’ll talk about this next lecture)
  • x * y is multiplication
▪ * is also used as part of the data type in pointer variable declarations – this is NOT an operator in this context!
Assignment in C

- A variable is represented by a location
- Declaration ≠ initialization (initially “mystery data”)
- \textbf{int} \ x, \ y;
  - \(x\) is at address 0x04, \(y\) is at 0x18

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

little-endian
Assignment in C

- A variable is represented by a location
- Declaration ≠ initialization (initially “mystery data”)

```c
int x, y;
```

- `x` is at address 0x04, `y` is at 0x18

![32-bit example diagram](little-endian)
Assignment in C

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

```c
int x, y;

x = 0;
```

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>0x00</td>
<td>0x04</td>
<td>0x08</td>
<td>0x0C</td>
</tr>
<tr>
<td>0x10</td>
<td>0x14</td>
<td>0x18</td>
<td>0x1C</td>
</tr>
<tr>
<td>0x20</td>
<td>0x24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assignment in C

- **left-hand side = right-hand side;**
  - LHS must evaluate to a *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 *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

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

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

- left-hand side = right-hand side;
  - LHS must evaluate to a 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”
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a 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;` // expect 0x1b
  - Get address of y, “add 3”, store in z

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

& = “address of”
* = “dereference”

Pointer arithmetic
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;`

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`
Arrays in C

Declaration: `int a[6];` // &a is 0x10

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

a (array name) returns 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` (array name) returns the array’s address.
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes.

![Memory Layout Diagram]

- `a[0]`: 0x10, 0x01 0x00 0x00
- `a[2]`: 0x18
- `a[4]`: 0x20

---

0x00 0x08 0x01 0x09 0x02 0x0A 0x03 0x0B 0x04 0x0C 0x05 0x0D 0x06 0x0E 0x07 0x0F
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` (array name) returns the array’s address

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

```
0x00  0x08  0x10  0x18  0x20  0x28  0x30  0x38  0x40  0x48
0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x08  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x10  0x5F  01  00  00  0x5F  01  00  00  0x5F  01  00  00
0x18  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x20  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x28  0xAD  0B  00  00  0xAD  0B  00  00  0xAD  0B  00  00
```

`&a[i]` is the address of `a[0]` plus `i` times the element size in bytes.

`a[−1]` is the address of `a[0]` minus one 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;`

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

`&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` (array name) returns the array’s address
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;`  

equivalent `p = a;`  
`p = &a[0];`  
`*p = 0xA;`

`p[i] ↔ *p + i`

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

`a` (array name) returns the array’s address

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

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

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

pointer arithmetic: `0x10 + 1 → 0x14`
`p = p + 2;`

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

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

pointer arithmetic: `0x10 + 1 → 0x14`
`p = p + 2;`
Arrays in C

Declaration: \texttt{int a[6];}

Indexing: \texttt{a[0] = 0x015f;}
\texttt{a[5] = a[0];}

No bounds \texttt{a[6] = 0xBAD;}
checking: \texttt{a[-1] = 0xBAD;}

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

\texttt{p[1] = 0xB;}
\texttt{*p + 1 = 0xB;}
\texttt{p = p + 2;}
\texttt{*p = a[1] + 1;}

Arrays are adjacent locations in memory storing the same type of data object
\texttt{a} (array name) returns the array’s address
\texttt{\&a[i]} is the address of \texttt{a[0]} plus \texttt{i} times the element size in bytes

\begin{enumerate}
\item \texttt{p = \&a[0];}
\item \texttt{*p = 0xA;}
\item \texttt{p[1] = 0xB;}
\item \texttt{*p + 1 = 0xB;}
\item \texttt{p = p + 2;}
\item \texttt{*p = a[1] + 1;}
\end{enumerate}
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 5?

- Vote in Ed Lessons

```c
void main() {
    int a[] = {0x5, 0x10};
    int* p = a;
    p = p + 1;
    *p = *p + 1;
}
```

<table>
<thead>
<tr>
<th>Data (hex)</th>
<th>Address (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>0x100</td>
</tr>
<tr>
<td>a[1]</td>
<td>0x104</td>
</tr>
<tr>
<td>p</td>
<td>0x101</td>
</tr>
</tbody>
</table>

- (A) 0x101 0x5 0x11
- (B) 0x104 0x5 0x11
- (C) 0x101 0x6 0x10
- (D) 0x104 0x6 0x10
# Representing strings (Review)

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

## ASCII: American Standard Code for Information Interchange

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>space</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>”</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>’</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>:</td>
</tr>
<tr>
<td>59</td>
<td>;</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>63</td>
<td>?</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>124</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>126</td>
<td>~</td>
</tr>
<tr>
<td>127</td>
<td>del</td>
</tr>
</tbody>
</table>
Representing strings (Review)

- **C-style string stored as an array of bytes (char*)**
  - No “String” keyword, unlike Java
  - Elements are one-byte ASCII codes for each character
  - Last character followed by a 0 byte (\"\0\")
    (a.k.a. the *null character*)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>83</td>
<td>0x53</td>
<td>'S'</td>
</tr>
<tr>
<td>116</td>
<td>0x74</td>
<td>'t'</td>
</tr>
<tr>
<td>97</td>
<td>0x61</td>
<td>'a'</td>
</tr>
<tr>
<td>121</td>
<td>0x79</td>
<td>'y'</td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td>''</td>
</tr>
<tr>
<td>115</td>
<td>0x73</td>
<td>'s'</td>
</tr>
<tr>
<td>97</td>
<td>0x61</td>
<td>'a'</td>
</tr>
<tr>
<td>102</td>
<td>0x66</td>
<td>'f'</td>
</tr>
<tr>
<td>101</td>
<td>0x65</td>
<td>'e'</td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td>''</td>
</tr>
<tr>
<td>87</td>
<td>0x57</td>
<td>'W'</td>
</tr>
<tr>
<td>65</td>
<td>0x41</td>
<td>'A'</td>
</tr>
<tr>
<td>0</td>
<td>0x00</td>
<td>'\0'</td>
</tr>
</tbody>
</table>

String literal: “Stay safe UA” uses 13 bytes (double quotes)
## Endianness and Strings

**char** `s[6] = "12345";`

- **String literal**
  - 0x31 = 49 decimal = ASCII ‘1’

<table>
<thead>
<tr>
<th>IA32, x86-64 (little-endian)</th>
<th>SPARC (big-endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 31</td>
<td>0x00 31</td>
</tr>
<tr>
<td>0x01 32</td>
<td>0x01 32</td>
</tr>
<tr>
<td>0x02 33</td>
<td>0x02 33</td>
</tr>
<tr>
<td>0x03 34</td>
<td>0x03 34</td>
</tr>
<tr>
<td>0x04 35</td>
<td>0x04 35</td>
</tr>
<tr>
<td>0x05 00</td>
<td>0x05 00</td>
</tr>
</tbody>
</table>

- **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)
Examining Data Representations

❖ Code to print byte representation of data
  ▪ Treat any data type as a byte array by casting its address 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%.2hhX\n", start+i, *(start+i));
    printf("\n");
}
```

❖ printf directives:
  ▪ %p   Print pointer
  ▪ \t   Tab
  ▪ %.2hhX Print value as char (hh) in hex (X), padding to 2 digits ( . 2)
  ▪ \n   New line
Examining Data Representations

❖ Code to print byte representation of data
  ▪ Treat any data type as a *byte array* by **casting** its address 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%2hhX\n", start+i, *(start+i));
    printf("\n"); /* format string */
}
```

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

```c
int x = 123456; // 0x00 01 E2 40
printf("int x = %d;\n", x);
show_int(x);  // show_bytes((char *) &x, sizeof(int));
```

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

```c
int x = 123456;
0x7fffb245549c  0x40
0x7fffb245549d  0xE2
0x7fffb245549e  0x01
0x7fffb245549f  0x00
```
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)