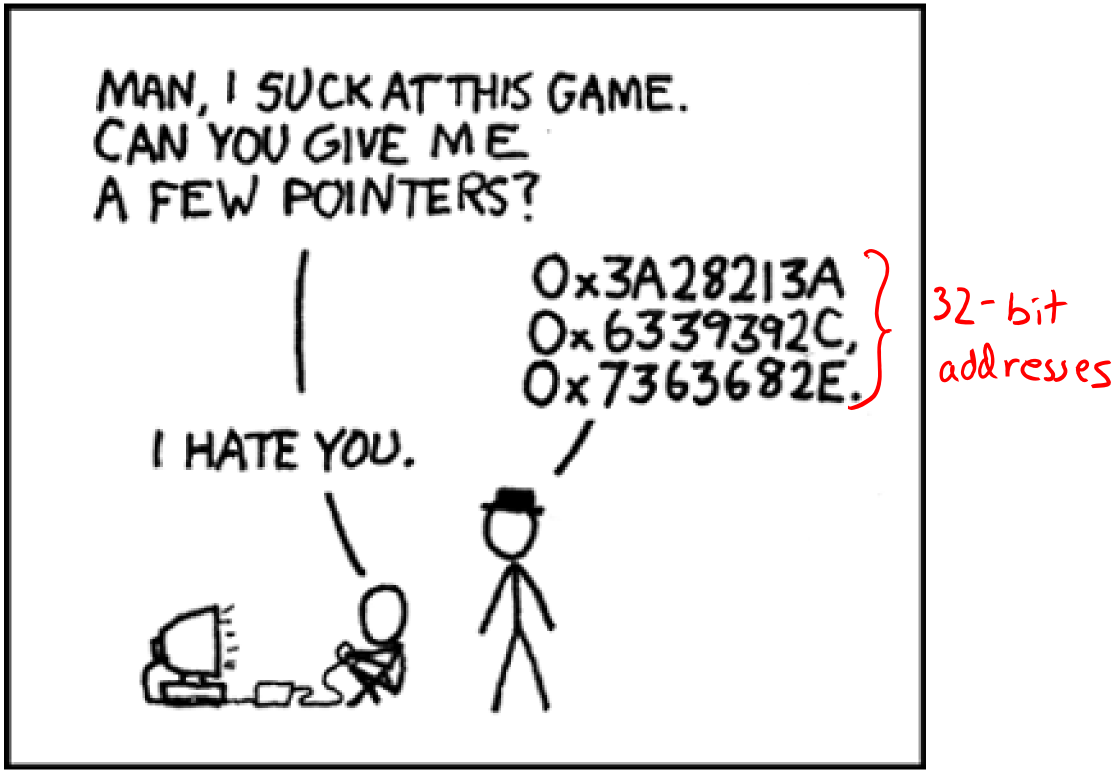
Memory, Data, & Addressing II

CSE 351 Spring 2021

Instructor:
Ruth Anderson

Teaching Assistants:
Allen Aby
Joy Dang
Alena Dickmann
Catherine Guevara
Corinne Herzog
Ian Hsiao
Diya Joy
Jim Limprasert
Armin Magness
Aman Mohammed
Monty Nitschke
Allie Pfleger
Neil Ryan
Alex Saveau
Sanjana Sridhar
Amy Xu



http://xkcd.com/138/

Administrivia

- hw1 due tonight, Friday (4/02) @ 11:59 pm
- Lab 0 and hw2 due Monday (4/05) @ 11:59 pm
- hw3 due Wednesday (4/07) @ 11:59 pm
- Lab 1a coming soon! due next Monday (4/12)
  - Pointers in C
  - Submitted via Gradescope
  - Reminder: last submission graded, *individual* work
- Questions Doc: You must log on with your @uw google account to access!!
  - https://tinyurl.com/CSE351-21sp-Questions
  - Then open the "TODAY's Lecture Questions" doc for 11:30/2:30

Late Days

- You are given 7 late days for the whole quarter
  - Late days can only apply to Labs & Unit Summaries
  - No benefit to having leftover late days
- 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 20% deduction of your score per day
  - Only late work is 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

Memory, Data, and Addressing

- Representing information as bits and bytes
  - Binary, hexadecimal, fixed-widths
- Organizing and addressing data in memory
  - Memory is a byte-addressable array
  - Machine "word" size = address size = register size
  - Endianness – ordering bytes in memory
- **Manipulating data in memory using C**
  - **Assignment**
  - **Pointers, pointer arithmetic, and arrays**
- Boolean algebra and bit-level manipulations

Reading Review

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

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`
  - B. `p + 10`
  - C. `&x + 10`
  - D. `*(&p)`
  - E. `ar[1]`
  - F. `&ar[2]`
  ```

Addresses and Pointers in C

* is also used with variable declarations

- & = "address of" operator
- * = "value at address" or "dereference" operator

```
int* ptr;
```

Declares a variable, `ptr`, that is a pointer to (*i.e.* holds the address of) an `int` in memory

```
int x = 5;
int y = 2;
```

Declares two variables, `x` and `y`, that hold `int`s, and *initializes* them to 5 and 2, respectively

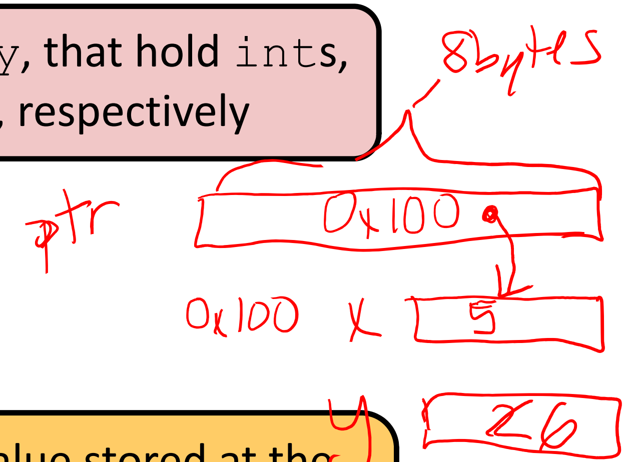
```
ptr = &x;
```

Sets `ptr` to the address of `x` ("`ptr` points to `x`")

```
y = 1 + *ptr;
```

"Dereference `ptr`"

Sets `y` to "1 plus the value stored at the address held by `ptr`." Because `ptr` points to `x`, this is equivalent to `y=1+x;`



What is `*(&y)` ? → y

Pointer Operators

- & = "address of" operator
- * = "value at address" or "dereference" operator

- Operator confusion

  &x
  *p

  - 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!

data type → char* p;

NOT an operator

Assignment in C

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

	0x00	0x01	0x02	0x03	
0x00	A7	00	32	00	
0x04	00	01	29	F3	x
0x08	EE	EE	EE	EE	
0x0C	FA	CE	CA	FE	
0x10	26	00	00	00	
0x14	00	00	10	00	
0x18	01	00	00	00	y
0x1C	FF	00	F4	96	
0x20	DE	AD	BE	EF	
0x24	00	00	00	00	

current state of memory

Assignment in C

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

little-endian

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

	0x00	0x01	0x02	0x03	
0x00					
0x04	00	01	29	F3	x
0x08					
0x0C					
0x10					
0x14					
0x18	01	00	00	00	y
0x1C					
0x20					
0x24					

Assignment in C

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

`&` = "address of"
`*` = "dereference"

- 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;` 0x 00 00 00 00 (pad)
  - x ← int (4 bytes)

	0x00	0x01	0x02	0x03	
0x00					
0x04	00	00	00	00	x
0x08					
0x0C					
0x10					
0x14					
0x18	01	00	00	00	y
0x1C					
0x20					
0x24					

Assignment in C

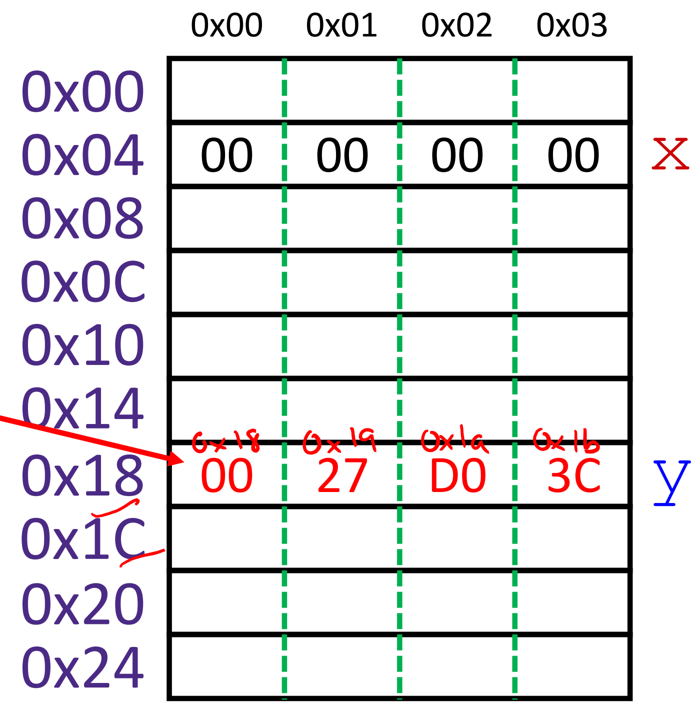
32-bit example
(pointers are 32-bits wide)

`&` = "address of"
`*` = "dereference"

- 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;`

MSB LSB

little endian!

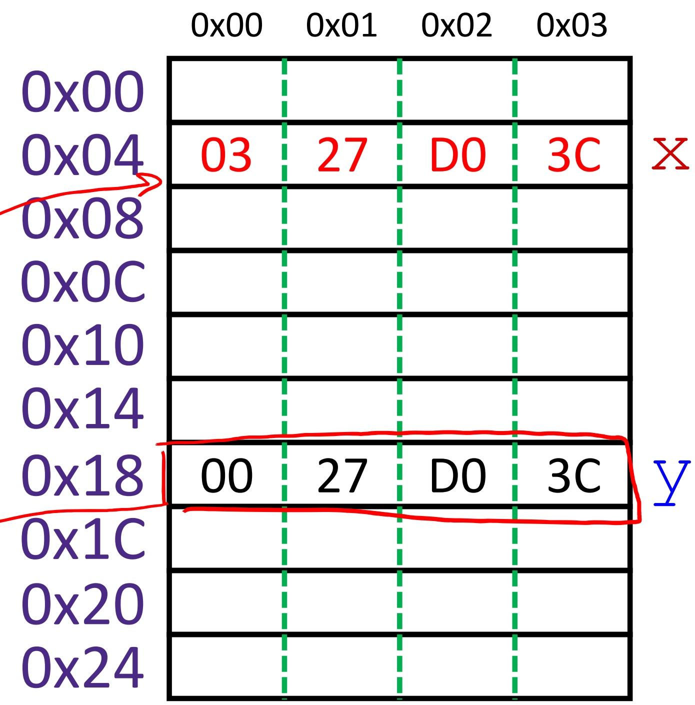


Assignment in C

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

`&` = "address of"
`*` = "dereference"

- 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`

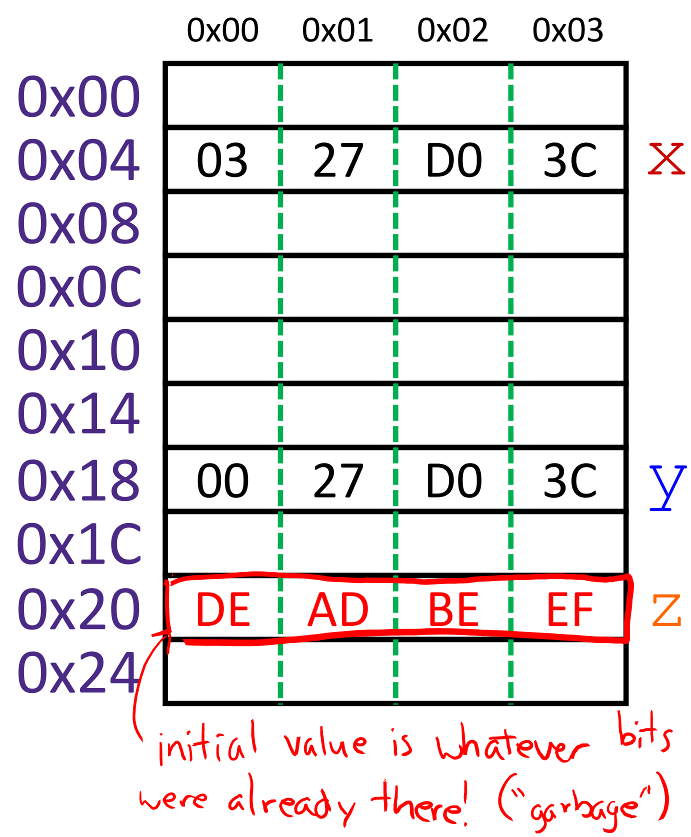


Assignment in C

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

`&` = "address of"
`*` = "dereference"

- 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;` ← pointer to an int
  - `z` is at address 0x20



Assignment in C

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

& = "address of"
* = "dereference"

- 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;`
  - Get address of `y`, "add 3", store in `z`

Pointer arithmetic

	0x00	0x01	0x02	0x03	
0x00					
0x04	03	27	D0	3C	x
0x08					
0x0C					
0x10					
0x14					
0x18	00	27	D0	3C	y
0x1C					
0x20	24	00	00	00	z
0x24					

Pointer Arithmetic

- Pointer arithmetic is scaled by the size of target type
  - In this example, `sizeof(int)` = 4
- `int* z = &y + 3;`
  - Get address of `y`, add `3*sizeof(int)`, store in `z`
  - `&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 *casting*

Assignment in C

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

`&` = "address of"
`*` = "dereference"

- `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?

	0x00	0x01	0x02	0x03	
0x00					
0x04	03	27	D0	3C	x
0x08					
0x0C					
0x10					
0x14					
0x18	00	27	D0	3C	y (RHS)
0x1C					
0x20	24	00	00	00	z
0x24					(LHS)

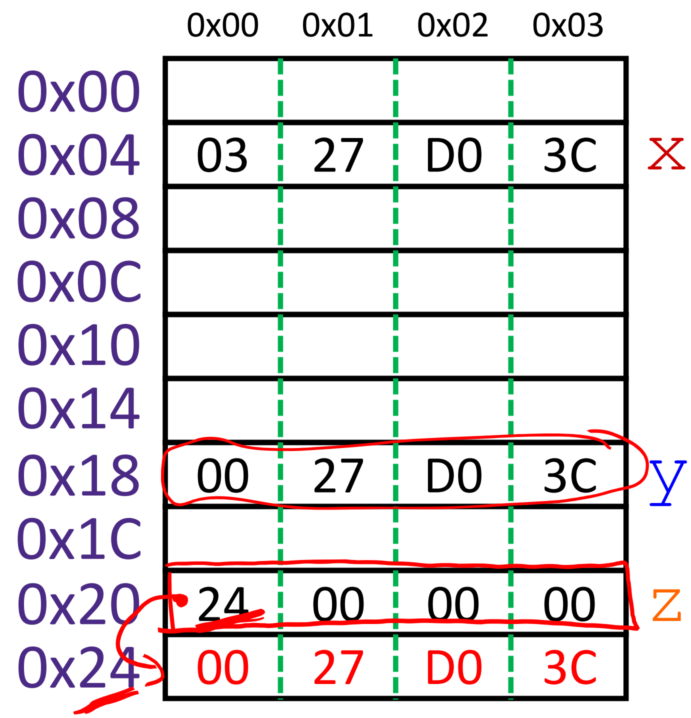
Assignment in C

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

`&` = "address of"
`*` = "dereference"

- `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`

The target of a pointer is also a location



Arrays in C

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

a (array name) returns the array's address

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

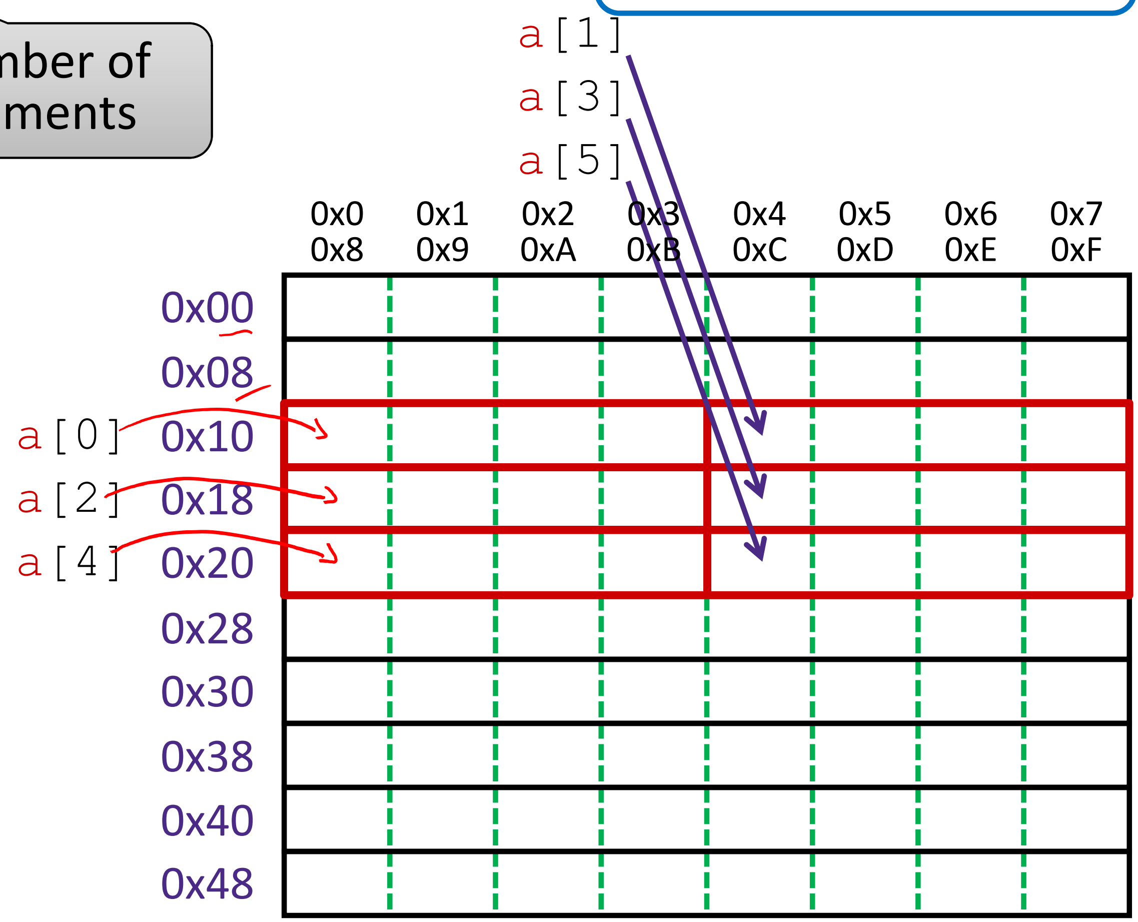
4 bytes each

element type

name

number of elements

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



Arrays in C

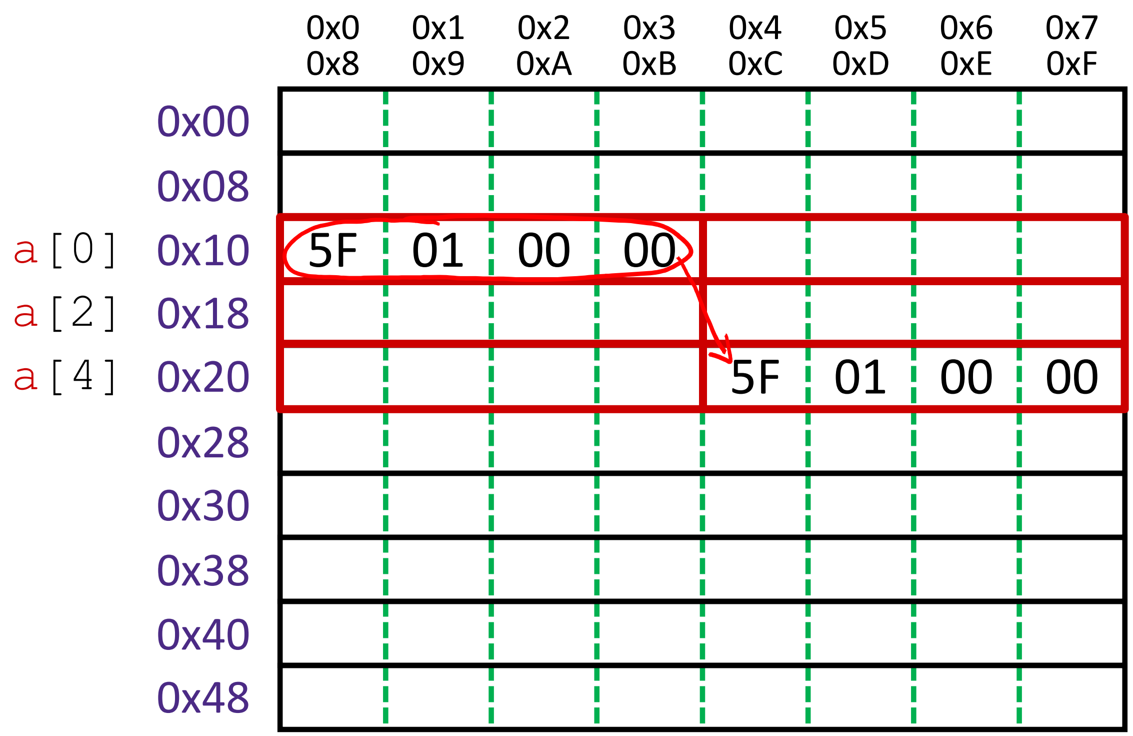
Arrays are adjacent locations in memory storing the same type of data object

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

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

Declaration: `int a[6];`

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



Arrays in C

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

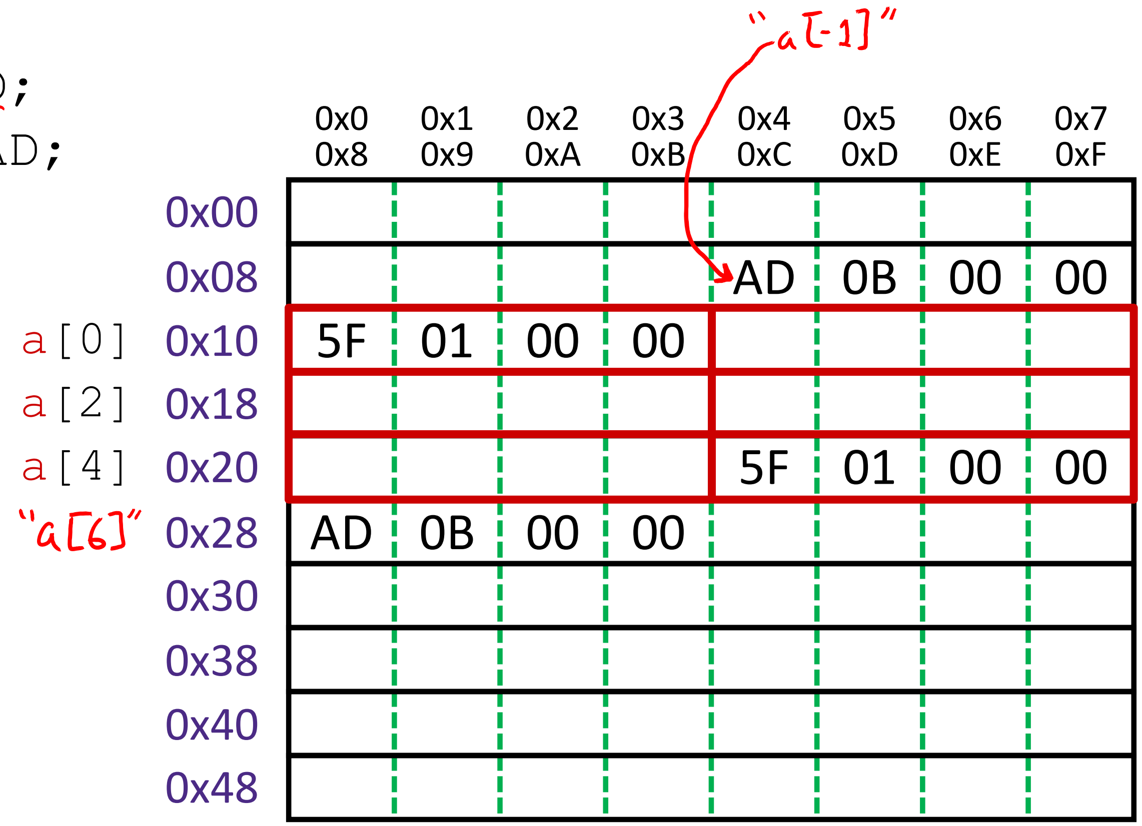
`a` (array name) returns the array's address

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

Declaration: `int a[6];`

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

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



Arrays in C

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

Declaration: `int a[6];`

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

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

Pointers:
```
int* p;
p = a;
p = &a[0];
*p = 0xA;
```
(equivalent: `p = a;` and `p = &a[0];`)

		0x0 / 0x8	0x1 / 0x9	0x2 / 0xA	0x3 / 0xB	0x4 / 0xC	0x5 / 0xD	0x6 / 0xE	0x7 / 0xF
	0x00								
	0x08					AD	0B	00	00
a[0]	0x10	0A	00	00	00				
a[2]	0x18								
a[4]	0x20					5F	01	00	00
	0x28	AD	0B	00	00				
	0x30								
	0x38								
p	0x40	10	00	00	00	00	00	00	00
	0x48								

Arrays in C

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

Declaration: `int a[6];`

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

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

Pointers: `int* p;`

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

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

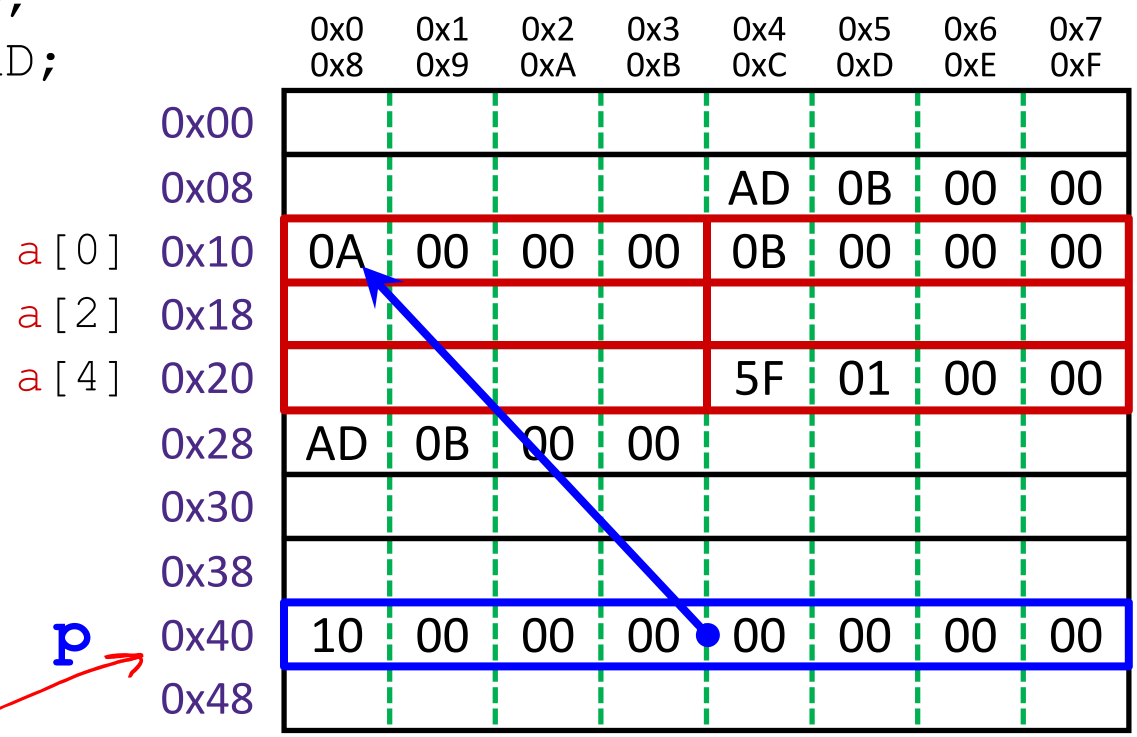
equivalent {
```
p[1] = 0xB;
*(p+1) = 0xB;
```
}

pointer arithmetic: 0x10 + 1 → 0x14

```
p = p + 2;
```

0x10 + 2 → 0x18

p[i] ⟺ *(p + i)



a + 2*sizeof(int) = 0x18

Arrays in C

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

Declaration: `int a[6];`

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

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

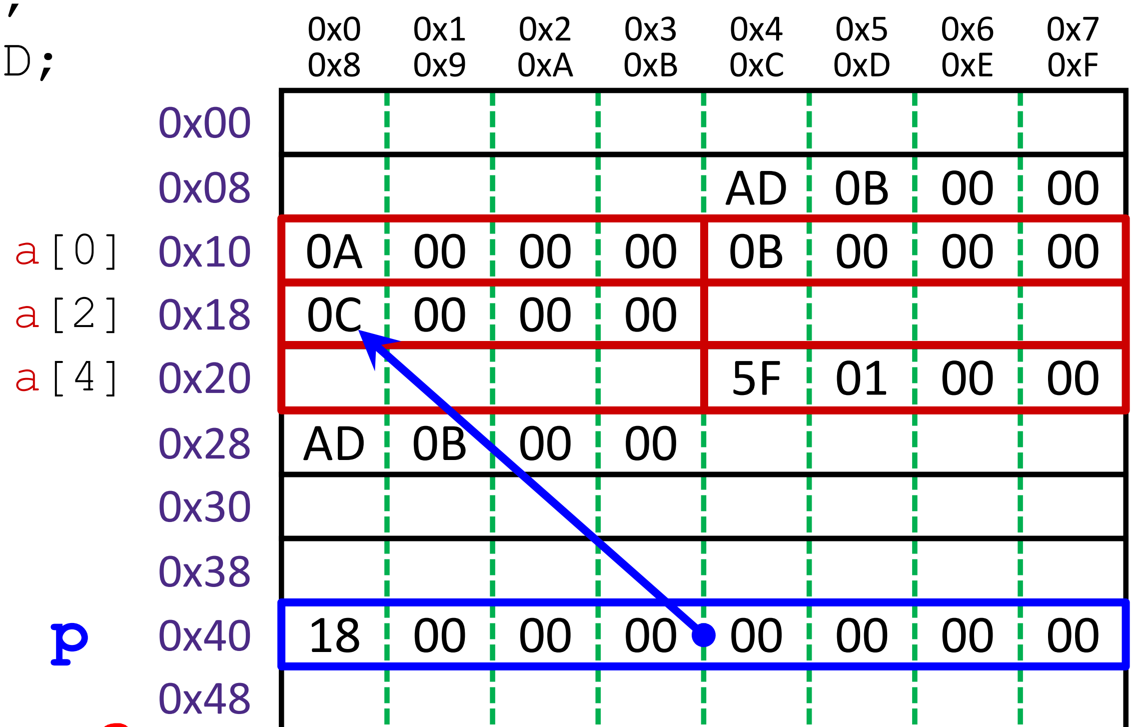
Pointers: `int* p;`

equivalent {
```
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;
```
}
```
p = p + 2;
```

store at 0x18 → `*p = a[1] + 1;` 0xB + 1 = 0xC

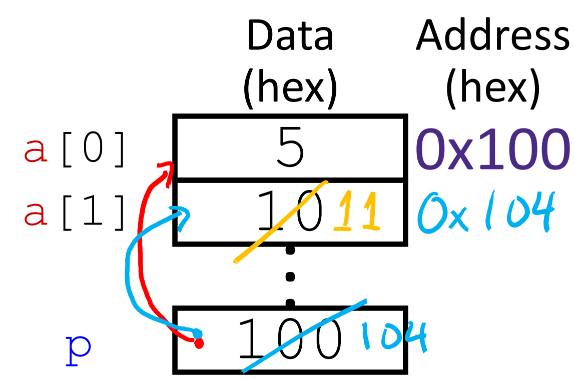


(no pointer arithmetic)

Question: The variable values after Line 3 executes are shown on the right. What are they after Line 5?

- No voting

```
1   void main() {
2     int a[] = {0x5,0x10};
3     int* p = a;
4     p = p + 1;
5     *p = *p + 1;
6   }
```



	p	a[0]	a[1]
(A)	0x101	0x5	0x11
(B)	0x104	0x5	0x11
(C)	0x101	0x6	0x10
(D)	0x104	0x6	0x10

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

decimal character

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

in C, use single quotes.

char c = '3';
↑
gets the value 51

ASCII: American Standard Code for Information Interchange

Representing strings

- 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. "null terminator")

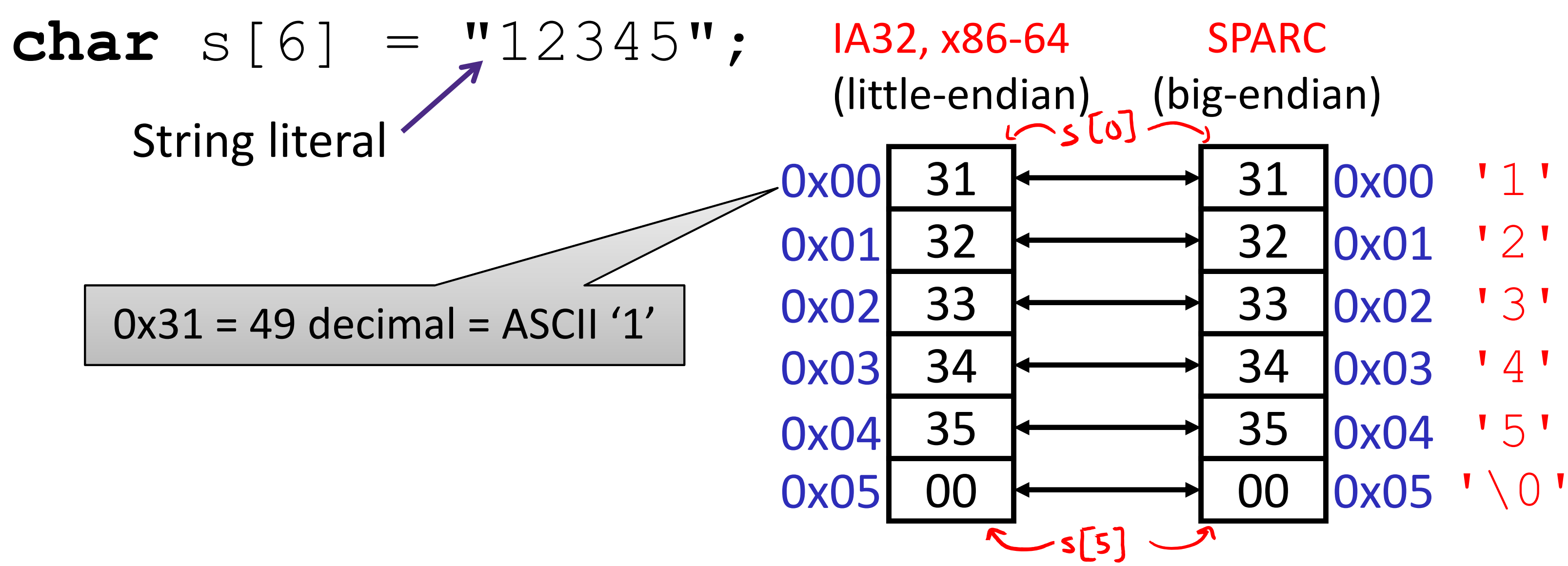
Decimal:	80	108	101	97	115	101	32	118	111	116	101	33	0
Hex:	0x50	0x6C	0x65	0x61	0x73	0x65	0x20	0x76	0x6F	0x74	0x65	0x21	0x00
Text:	'P'	'l'	'e'	'a'	's'	'e'		'v'	'o'	't'	'e'	'!'	'\0'

6 characters 1 4 1 1

string literal: "Please vote!" uses 13 bytes
(double quotes)

Endianness and Strings

C (char = 1 byte)



- 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

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 !!

```
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 !!

```
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

pointer arithmetic on char*

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

int*

4 bytes

"cast" (treat as)

show_bytes Execution Example

```
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

```
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)