Memory, Data, & Addressing II
CSE 351 Winter 2021

Instructor:
Mark Wyse

Teaching Assistants:
Kyrie Dowling
Catherine Guevara
Ian Hsiao
Jim Limprasert
Armin Magness
Allie Pfleger
Cosmo Wang
Ronald Widjaja

http://xkcd.com/138/
Administrivia

❖ Lab 0 due today @ 11:59 pm
  ▪ You will revisit these concepts later!

❖ hw1 due today @ 11:59 pm

❖ hw2 due Monday, hw3 due Wednesday @ 11:00 am
  ▪ Autograded, unlimited tries, no late submissions

❖ Lab 1a released today, due next Friday (1/15)
  ▪ Pointers in C
  ▪ Reminder: last submission graded, individual work
National Events, Resources, and Week 1

- Blog post by UW President Cauce:
  - https://www.washington.edu/president/2021/01/06/misinformation-disinformation-and-the-assault-on-democracy/

- Be there for each other, check in with friends and classmates, give space to process

- Support resources:
  - CSE Undergraduate Advising: ugrad-adviser@cs.washington.edu
  - Hall Health and Schmitz Hall Counseling Center: https://wellbeing.uw.edu/topic/mental-health/
  - SafeCampus is the UW's central reporting office if you are concerned for yourself or a friend. They have trained specialists who will take your call and connect you with appropriate resources. They are available 24/7 at 206-685-SAFE (206-685-7233).

- CSE 351: all week 1 work due Sunday 1/10 @ 11:59pm
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
  B. p + 10
  C. &x + 10
  D. *(&p)
  E. ar[1]
  F. &ar[2]
### 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 holds “garbage”)
❖ `int x, y;`

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

<table>
<thead>
<tr>
<th>Address</th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>A7</td>
<td>00</td>
<td>32</td>
<td>00</td>
</tr>
<tr>
<td>0x04</td>
<td>00</td>
<td>01</td>
<td>29</td>
<td>F3</td>
</tr>
<tr>
<td>0x08</td>
<td>EE</td>
<td>EE</td>
<td>EE</td>
<td>EE</td>
</tr>
<tr>
<td>0x0C</td>
<td>FA</td>
<td>CE</td>
<td>CA</td>
<td>FE</td>
</tr>
<tr>
<td>0x10</td>
<td>26</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x14</td>
<td>00</td>
<td>00</td>
<td>10</td>
<td>00</td>
</tr>
<tr>
<td>0x18</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1C</td>
<td>FF</td>
<td>00</td>
<td>F4</td>
<td>96</td>
</tr>
<tr>
<td>0x20</td>
<td>DE</td>
<td>AD</td>
<td>BE</td>
<td>EF</td>
</tr>
<tr>
<td>0x24</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

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

![32-bit example (pointers are 32-bits wide)]

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;`
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”

```
int x, y;

x = 0;
y = 0x3CD02700;
x = y + 3;
```
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”

---

Here is an example of assigning a value to a variable in C:

```c
int x, y;

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

This code first declares `x` and `y` as integer variables. Then, it assigns the value 0 to `x`, the value `0x3CD02700` to `y`, and finally, it assigns the value of `y` plus 3 to `x`. This demonstrates how to perform arithmetic operations on values stored in memory locations and assign the result back to another location.
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;`
  - Get address of `y`, “add 3”, store in `z`

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;`

**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”
Addresses and Pointers in C

- Draw out a box-and-arrow diagram for the result of the following C code:

```c
int* ptr;

int x = 5;

int y = 2;

ptr = &x;

y = 1 + *ptr;
```
Arrays in C

Declaration: `int a[6];`

- **element type**
- **name**
- **number of elements**

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)

<table>
<thead>
<tr>
<th></th>
<th>0x0</th>
<th>0x1</th>
<th>0x2</th>
<th>0x3</th>
<th>0x4</th>
<th>0x5</th>
<th>0x6</th>
<th>0x7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `a[0]` 0x10
- `a[2]` 0x18
- `a[4]` 0x20

- `a[1]` 0x14
- `a[3]` 0x1C
- `a[5]` 0x20
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.

```plaintext
0x00
0x08
a[0] 0x10
a[2] 0x18
a[4] 0x20
0x28
0x30
0x38
0x40
0x48
```

```
0x0
0x1 0x5F
0x2 0x01
0x3 0x00
0x4 0x00
```
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.

![Memory Layout](image)
Arrays in C

Declaration: `int a[6];`

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

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

Pointers: `int* p;`

equivalent:

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

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
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` (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)

- `p[1] = 0xB;`
- `*(p+1) = 0xB;`
- `p = p + 2;`
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;`
- equivalent `*(p+1) = 0xB;`
- `p = p + 2;`
- `*p = a[1] + 1;`

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
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>Address (hex)</th>
<th>Data (hex)</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) 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***)
  - No “String” keyword, unlike Java
  - Elements are one-byte **ASCII codes** for each character

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

**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")

<table>
<thead>
<tr>
<th>Decimal</th>
<th>80</th>
<th>108</th>
<th>101</th>
<th>97</th>
<th>115</th>
<th>101</th>
<th>32</th>
<th>118</th>
<th>111</th>
<th>116</th>
<th>101</th>
<th>33</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>0x50</td>
<td>0x6C</td>
<td>0x65</td>
<td>0x61</td>
<td>0x73</td>
<td>0x65</td>
<td>0x20</td>
<td>0x76</td>
<td>0x6F</td>
<td>0x74</td>
<td>0x65</td>
<td>0x21</td>
<td>0x00</td>
</tr>
<tr>
<td>Text</td>
<td>'P'</td>
<td>'l'</td>
<td>'e'</td>
<td>'a'</td>
<td>'s'</td>
<td>'e'</td>
<td>'v'</td>
<td>'o'</td>
<td>'t'</td>
<td>'e'</td>
<td>'!'</td>
<td>'\0'</td>
<td></td>
</tr>
</tbody>
</table>
Endianness and Strings

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

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

- **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

**Examples**

<table>
<thead>
<tr>
<th>IA32, x86-64 (little-endian)</th>
<th>SPARC (big-endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 31</td>
<td>31 0x00 '1'</td>
</tr>
<tr>
<td>0x01 32</td>
<td>32 0x01 '2'</td>
</tr>
<tr>
<td>0x02 33</td>
<td>33 0x02 '3'</td>
</tr>
<tr>
<td>0x03 34</td>
<td>34 0x03 '4'</td>
</tr>
<tr>
<td>0x04 35</td>
<td>35 0x04 '5'</td>
</tr>
<tr>
<td>0x05 00</td>
<td>00 0x05 '\0'</td>
</tr>
</tbody>
</table>
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");
}

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)