ARRAY REVIEW
Arrays – A Review

• **Formal Def:** A *statically-sized, contiguously allocated collection of homogenous data elements*
• Collection of homogenous data elements
  – Multiple variables of the same data type
• Contiguously allocated in memory
  – One right after the next
• **Statically-sized**
  – Size of the collection must be a constant and can’t be changed after initial declaration/allocation
• Collection is referred to with **one name**
• Individual elements referred to by an **offset/index** from the start of the array [in C, first element is at index 0]

```c
char c = A[0]; // ’h’
int D[20];
D[0] = 1; D[1] = 2;
```
Arrays As Collections

• If I have several players in a game I could declare a separate variable to track each one’s score:
  – int player1 = N; int player2 = N; int player3 = N; ...
  – PAINFUL!!

• Better idea: Use an array where the index to the desired element can be a variable:
  – for(i=0; i < N; i++){
    player[i] = N;
  }

• Can still refer to individual items if desired: player[2]
Arrays

- Track amount of money (# of coins) 3 people have.
- Homogenous data set (number of coins) for multiple people...perfect for an array
  - `int num_coins[3];`
Arrays

• Track amount of money (# of coins) 3 people have.

• Homogenous data set (number of coins) for multiple people...perfect for an array
  – int num_coins[3];

• Must initialize elements of an array
  – for(int i=0; i < 3; i++)
    num_coins[i] = 0;
Arrays

- Track amount of money (# of coins) 3 people have.
- Homogenous data set (number of coins) for multiple people...perfect for an array
  - int num_coins[3];
- Must initialize elements of an array
  - for(int i=0; i < 3; i++)
    num_coins[i] = 0;
- Can access each persons amount and perform ops on that value
  - num_coins[0] = 5;
    num_coins[1] = 8;
ARRAY ODDS AND ENDS
Static Size/Allocation

• For now, arrays must be declared as fixed size (i.e. a constant known at compile time)
  – Good:
    • int x[10];
    • #define MAX_ELEMENTS 100
      int x[MAX_ELEMENTS];
    • const int MAX_ELEMENTS = 100;
      int x[MAX_ELEMENTS];
  – Bad:
    • int mysize;
      cin >> mysize;
      int x[mysize];
    • int mysize = 10;
      int x[mysize];

Compiler must be able to figure out how much memory to allocate at compile-time
Initializing Arrays

- Integers or floating point types can be initialized by placing a comma separated list of values in curly braces {...}
  - int data[5] = {4,3,9,6,14};
  - char vals[8] = {64,33,18,4,91,76,55,21};

- If accompanied w/ initialization list, size doesn’t have to be indicated (empty [])
  - double stuff[] = {3.5, 14.22, 9.57}; // = stuff[3]

- However the list must be of constants, not variables:
  - BAD: double z = 3.5; double stuff[] = {z, z, z};
Understanding array addressing and indexing

ACCESSING DATA IN AN ARRAY
Exercise

- Consider a train of box cars
  - The initial car starts at point A on the number line
  - Each car is 5 meters long
- Write an expression of where the i-th car is located (at what meter does it start?)
- Suppose a set of integers start at memory address A, write an expression for where the i-th integer starts?
- Suppose a set of doubles start at memory address A, write an expression for where the i-th double starts?
More on Accessing Elements

- Assume a 5-element int array
  - int x[5] = {8,5,3,9,6};
- When you access x[2], the CPU calculates where that item is in memory by taking the start address of x (i.e. 100) and adding the product of the index, 2, times the size of the data type (i.e. int = 4 bytes)
  - x[2] => int. @ address 100 + 2*4 = 108
  - x[3] => int. @ address 100 + 3*4 = 112
  - x[i] @ start address of array + i * (size of array type)
- C does not stop you from attempting to access an element beyond the end of the array
  - x[6] => int. @ address 100 + 6*4 = 124 (Garbage!!)
Passing arrays to other functions

ARRAYS AS ARGUMENTS
Passing Arrays as Arguments

• In function declaration / prototype for the formal parameter use
  – “type []” or “type *” to indicate an array is being passed
• When calling the function, simply provide the name of the array as the actual argument
  – In C/C++ using an array name without any index evaluates to the starting address of the array
• C does NOT implicitly keep track of the size of the array
  – Thus either need to have the function only accept arrays of a certain size
  – Or need to pass the size (length) of the array as another argument

```c
void add_1_to_array_of_10(int []);
void add_1_to_array(int *, int);
int main(int argc, char *argv[])
{
    int data[10] = {9,8,7,6,5,4,3,2,1,0};
    add_1_to_array_of_10(data);
    cout << “data[0]” << data[0] << endl;
    add_1_to_array(data,10);
    cout << ”data[9]” << data[9] << endl;
    return 0;
}
void add_1_to_array_of_10(int my_array[])
{
    int i=0;
    for(i=0; i < 10; i++){
        my_array[i]++;
    }
}
void add_1_to_array(int *my_array, int size)
{
    int i=0;
    for(i=0; i < size; i++){
        my_array[i]++;
    }
}
```
Passing Arrays as Arguments

- In function declaration / prototype for the *formal* parameter use *type [*]
- When calling the function, simply provide the name of the array as the *actual* argument
- Scalar values (int, double, char, etc.) are “*passed-by-value*”
  - Copy is made and passed
- Arrays are “*passed-by-reference*”
  - We are NOT making a copy of the entire array (that would require too much memory and work) but passing a reference to the actual array (i.e. an address of the array)
  - Thus any changes made to the array data in the called function will be seen when control is returned to the calling function.

```c
void f1(int []);

int main(int argc, char *argv[])
{
    int data[10] = {10,11,12,13,14,15,16,17,18,19};
    cout << “Loc. 0=” << data[0] << endl;
    cout << “Loc. 9=” << data[9] << endl;
    f1(data);
    cout << “Loc. 0=” << data[0] << endl;
    cout << “Loc. 9=” << data[9] << endl;
    return 0;
}
```

```c
void f1(int my_array[])
{
    int i=0;
    for(i=0; i < 10; i++)
    {
        my_array[i]++;
    }
}
```

Output:

Loc. 0=10
Loc. 9=19
Loc. 0=11
Loc. 9=20
Null terminated character arrays

C-STRINGS
C Strings

• Character arrays (i.e. C strings)
  – Enclosed in double quotes " "
  – Strings of text are simply arrays of chars
  – Can be initialized with a normal C string (in double quotes)
  – C strings have one-byte (char) per character
  – End with a "null" character = 00 dec. = '\0' ASCII
  – cout "knows" that if a char array is provided as an argument it will print the 0th character and keep printing characters until a ‘\0’ (null) character [really just a value of 0] is encountered
  – cin "knows" how to take in a string and fill in a char array (stops at whitespace)
    • Careful it will write beyond the end of an array if the user enters a string that is too long

```cpp
#include<iostream>
using namespace std;
int main()
{
    char str1[6] = "Hello";
    char str2[] = "Hi\n";
    cout << str1 << str2;
    cin >> str1
}
```
Example: C String Functions

• Write a function to determine the length (number of characters) in a C string
• Write a function to copy the characters in a source string/character array to a destination character array
• Copy the template to your account
  – wget http://ee.usc.edu/~redekopp/cs103/string_funcs.cpp
• Edit and test your program and complete the functions:
  – int strlen(char str[])
  – strcpy(char dst[], char src[])
• Compile and test your functions
  – main() is complete and will call your functions to test them
MULTIDIMENSIONAL ARRAYS
Multidimensional Arrays

• Thus far arrays can be thought of 1-dimensional (linear) sets
  – only indexed with 1 value (coordinate)
  – char x[6] = {1,2,3,4,5,6};

• We often want to view our data as 2-D, 3-D or higher dimensional data
  – Matrix data
  – Images (2-D)
  – Index w/ 2 coordinates (row,col)
Multidimension Array Declaration

- **2D**: Provide size along both dimensions (normally rows first then columns)
  - Access w/ 2 indices
  - Declaration: `int my_matrix[2][3];`
  - Access elements with appropriate indices
    - `my_matrix[0][1]` evals to 3, `my_matrix[1][2]` evals to 2

- **3D**: Access data w/ 3 indices
  - Declaration: `char image[2][4][3];`
  - Up to human to interpret meaning of dimensions
    - Planes x Rows x Cols
    - Rows x Cols x Planes
  - Example:
    
    | Plane 0 | Plane 1 | Plane 0 |
    |---------|---------|---------|
    | 35  3  1 | 7 32 44 23 | 51 |
    | 6 14 72 | 10 59 18 88 | 6 |
    | 10 81 63 | Or 72 61 53 84 | Plane 2 |
    | 40 75 18 | 72 14 72 | |
    | 39 21 7 | | |

  - Example:
    - `Row 0`:
      
      | Col. 0 | Col. 1 | Col. 2 |
      |--------|--------|--------|
      | 5  3  1 | 6  4  2 | |

  - Example:
    - `Row 1`:
      
      | Col. 0 | Col. 1 | Col. 2 |
      |--------|--------|--------|
      | 5  3  1 | 6  4  2 | |
Passing Multi-Dimensional Arrays

- **Formal Parameter**: Must give dimensions of all but first dimension
- **Actual Parameter**: Still just the array name (i.e. starting address)
- Why do we have to provide all but the first dimension?
- So that the computer can determine where element: data[i][j][k] is actually located in memory

```c
void doit(int my_array[][4][3])
{
    my_array[1][3][2] = 5;
}

int main(int argc, char *argv[])
{
    int data[2][4][3];
    doit(data);
    ...
    return 0;
}
```

![Memory Map]
Linearization of Multidimensional Arrays

- Analogy: Hotel room layout => 3D
  - Access location w/ 3 indices:
    - Floors, Aisles, Rooms
    - But they don’t give you 3 indices, they give you one room number
  - Room #’s are a linearization of the 3 dimensions
    - Room 218 => Floor=2, Aisle 1, Room 8
- When “linear”-izing we keep proximity for only lowest dimension
  - Room 218 is next to 217 and 219
- But we lose some proximity info for higher dimensions
  - Presumably room 218 is right below room 318
  - But in the linearization 218 seems very far from 318

Analogy: Hotel Rooms

1st Digit = Floor
2nd Digit = Aisle
3rd Digit = Room
Linearization of Multidimensional Arrays

- In a computer, multidimensional arrays must still be stored in memory which is addressed linearly (1-Dimensional)
- C/C++ use a policy that lower dimensions are placed next to each other followed by each higher level dimension

```c
int x[2][3];
```

<table>
<thead>
<tr>
<th>Row 0</th>
<th>Col. 0</th>
<th>Col. 1</th>
<th>Col. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Memory:
- x[0][0] = 00 00 00 05
- x[0][1] = 00 00 00 03
- x[0][2] = 00 00 00 01
- x[1][0] = 00 00 00 06
- x[1][1] = 00 00 00 04
- x[1][2] = 00 00 00 02
- x[2][0] = d2 19 2d 81
- ...
Linearization of Multidimensional Arrays

- In a computer, multidimensional arrays must still be stored in memory which is addressed linearly (1-Dimensional)
- C/C++ use a policy that lower dimensions are placed next to each other followed by each higher level dimension

```c
char y[2][4][3];
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>72</td>
</tr>
<tr>
<td>10</td>
<td>81</td>
<td>63</td>
</tr>
<tr>
<td>40</td>
<td>75</td>
<td>18</td>
</tr>
<tr>
<td>42</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>67</td>
<td>25</td>
<td>49</td>
</tr>
<tr>
<td>14</td>
<td>48</td>
<td>65</td>
</tr>
<tr>
<td>74</td>
<td>21</td>
<td>7</td>
</tr>
</tbody>
</table>
Linearization of Multidimensional Arrays

We could re-organize the memory layout (i.e. linearization) while still keeping the same view of the data by changing the order of the dimensions.

```
char y[4][3][2];
```

<table>
<thead>
<tr>
<th>35</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>14</td>
<td>72</td>
</tr>
<tr>
<td>10</td>
<td>81</td>
<td>63</td>
</tr>
<tr>
<td>40</td>
<td>75</td>
<td>18</td>
</tr>
<tr>
<td>42</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>67</td>
<td>25</td>
<td>49</td>
</tr>
<tr>
<td>14</td>
<td>48</td>
<td>65</td>
</tr>
<tr>
<td>74</td>
<td>21</td>
<td>7</td>
</tr>
</tbody>
</table>

Memory

```
<table>
<thead>
<tr>
<th>0</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>03</td>
</tr>
<tr>
<td>3</td>
<td>08</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
</tr>
<tr>
<td>7</td>
<td>67</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Linearization of Multidimensional Arrays

- Formula for location of item at row i, column j in an array with NUMR rows and NUMC columns:

  \[
  \text{Location} = \text{RowOffset} + \text{ColOffset}
  \]

  \[
  \text{RowOffset} = i \times \text{NUMC}
  \]

  \[
  \text{ColOffset} = j
  \]

  \[
  \text{Location} = (i \times \text{NUMC}) + j
  \]

Declaration:    \[
\text{int x[2][3]; // NUMR=2, NUMC = 3;}
\]

<table>
<thead>
<tr>
<th>Row 0</th>
<th>Row 1</th>
<th>Row 2</th>
<th>Row 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Col. 0</td>
<td>Col. 1</td>
<td>Col. 2</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Access: \[
\text{x[i][j]};
\]
Linearization of Multidimensional Arrays

- Formula for location of item at plane $p$, row $i$, column $j$ in array with NUMP planes, NUMR rows, and NUMC columns

```
int x[2][4][3]; // NUMP=2, NUMR=4, NUMC=3
```

Access: $x[p][i][j]$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>72</td>
</tr>
<tr>
<td>10</td>
<td>81</td>
<td>63</td>
</tr>
<tr>
<td>40</td>
<td>75</td>
<td>18</td>
</tr>
</tbody>
</table>

Memory

```
100 35
104 03
108 01
116 06
120 14
... ...
```

```
74 21 7
```
Revisited: Passing Multi-Dimensional Arrays

- Must give dimensions of all but first dimension
- This is so that when you use ‘myarray[p][i][j]’ the computer and determine where in the linear addresses that individual index is located in the array
  - [p][i][j] = start address + (p*NUMR*NUMC + i*NUMC + j)*sizeof(int)
  - [1][3][2] in an array of nx4x3 becomes: 1*(4*3) + 3(3) + 2 = 23 ints = 23*4 = 92 bytes into the array

```c
void doit(int my_array[][4][3])
{
    my_array[1][3][2] = 5;
}

int main(int argc, char *argv[])
{
    int data[2][4][3];
    doit(data);
    ...
    return 0;
}
```

```
35   3   1                   12
 6   14  72                  49
10   81  63
40   75  18
 74  21  7
```

```
100  35
104  03
108  01
112  06
116  14
120  18
124  18
128  18
132  18
136  18
140  18
144  18
148  18
152  18
156  18
Memory
```
Using 2- and 3-D arrays to create and process images

IMAGE PROCESSING
Practice: Drawing

• Download the BMP library code:
  – In your examples directory on your VM create a new subdirectory: gradient
    • $ rm –rf gradient
    • $ mkdir gradient
    • $ cd gradient
    • $ wget http://ee.usc.edu/~redekopp/cs103/gradient.tar
    • $ tar xvf gradient.tar
  – Code to read (open) and write (save) .BMP files is provided in bmplib.h and bmplib.cpp
  – Look at bmplib.h for the prototype of the functions you can use in your main() program in gradient.cpp

• gradient.cpp contains a main function and two global arrays: bwimage[255][255] and rgbimage[255][255][3]
  – bwimage is a 256x256 image with grayscale pixels (0=black, 255=white)
Multi-File Programs

• We need a way to split our code into many separate files so that we can partition our code
  – We often are given code libraries from other developers or companies
  – It can also help to put groups of related functions into a file
• bmplib.h has prototypes for functions to read, write, and show .BMP files as well as constant declarations
• bmplib.cpp has the implementation of each function
• gradient.cpp has the main application code
  – It #include's the .h file so as to have prototypes and constants available

Key Idea: The .h file tells you what library functions are available; The .cpp file tells you how it does it
Multi-file Compilation

• Three techniques to compile multiple files into a single application
  – Use 'make' with a 'Makefile' script
    • We will provide you a 'Makefile' whenever possible and it contains directions for how to compile all the files into a single program
    • To use it just type 'make' at the command prompt
  – Compile all the .cpp files together like:
    $ g++ -g -o gradient gradient.cpp bmplib.cpp
    • Note: NEVER compile .h files
Multi-file Compilation

• Three techniques to compile multiple files into a single application
  – Compile each .cpp files separately into an "object file" (w/ the 
    –c option) and then link them altogether into one program:
    $ g++ -c bmplib.cpp –o bmplib.o
    $ g++ -c gradient.cpp –o gradient.o
    $ g++ -g –o gradient gradient.o bmplib.o
  – The first two command produce .o (object) files which are 
    non-executable files of 1's and 0's representing the code
  – The last command produces an executable program by 
    putting all the .o files together
  – Don't do this approach in 103, but it is approach 'Makefiles' 
    use and the way most real programs are compiled
Practice: Drawing

• Draw an X on the image
  – Try to do it with only a single loop, not two in sequence

• Draw a single period of a sine wave
  – Hint: enumerate each column, x, with a loop and figure out the appropriate row (y-coordinate)
Practice: Drawing

• Modify gradient.cpp to draw a black cross on a white background and save it as 'output1.bmp'
• Modify gradient.cpp to draw a black X down the diagonals on a white background and save it as 'output2.bmp'
• Modify gradient.cpp to draw a gradient down the rows (top row = black through last row = white with shades of gray in between
• Modify gradient.cpp to draw a diagonal gradient with black in the upper left through white down the diagonal and then back to black in the lower right
Image Processing

• Go to your gradient directory
  – $ wget http://cs103.usc.edu/files/graphics/elephant.bmp

• Here is a first exercise...produce the "negative"

```c
#include "bmplib.h"
int main() {
    unsigned char image[SIZE][SIZE];
    readGSBMP("elephant.bmp", image);
    for (int i=0; i<SIZE; i++) {
        for (int j=0; j<SIZE; j++) {
            image[i][j] = 255 - image[i][j]; // invert color
        }
    }
    showGSBMP(image);
}
```
Practice: Image Processing

• Perform a diagonal flip

• Tile

• Zoom
Selected Grayscale Solutions

• X
  – http://cs103.usc.edu/files/graphics/x.cpp

• Sin
  – http://cs103.usc.edu/files/graphics/sin.cpp

• Diagonal Gradient
  – http://cs103.usc.edu/files/graphics/gradient_diag.cpp

• Elephant-flip
  – http://cs103.usc.edu/files/graphics/eg3-4.cpp

• Elephant-tile
  – http://cs103.usc.edu/files/graphics/eg3-5.cpp

• Elephant-zoom
  – http://cs103.usc.edu/files/graphics/zoom.cpp
Color Images

• Color images are represented as 3D arrays (256x256x3)
  – The lower dimension are Red, Green, Blue values
• Base Image
• Each color plane inverted
• Grayscale
  – Using NTSC formula: 
    \[ .299R + .587G + .114B \]
Color Images

• Glass filter
  – Each destination pixel is from a random nearby source pixel
    • http://cs103.usc.edu/files/graphics/glass.cpp

• Edge detection
  – Each destination pixel is the difference of a source pixel with its south-west neighbor
Color Images

• Smooth
  – Each destination pixel is average of 8 neighbors
    • [http://cs103.usc.edu/files/graphics/smooth.cpp](http://cs103.usc.edu/files/graphics/smooth.cpp)
Selected Color Solutions

- **Color fruit – Inverted**
  - [http://cs103.usc.edu/files/graphics/eg4-1.cpp](http://cs103.usc.edu/files/graphics/eg4-1.cpp)

- **Color fruit – Grayscale**
  - [http://cs103.usc.edu/files/graphics/eg4-3.cpp](http://cs103.usc.edu/files/graphics/eg4-3.cpp)

- **Color fruit – Glass Effect**
  - [http://cs103.usc.edu/files/graphics/glass.cpp](http://cs103.usc.edu/files/graphics/glass.cpp)

- **Color fruit – Edge Detection**
  - [http://cs103.usc.edu/files/graphics/eg5-4.cpp](http://cs103.usc.edu/files/graphics/eg5-4.cpp)

- **Color fruit – Smooth**
  - [http://cs103.usc.edu/files/graphics/smooth.cpp](http://cs103.usc.edu/files/graphics/smooth.cpp)
Using arrays as a lookup table

LOOKUP TABLES
Arrays as Look-Up Tables

- Use the value of one array as the index of another
- Suppose you are given some integers as data [in the range of 0 to 5]
- Suppose computing squares of integers was difficult (no built-in function for it)
- Could compute them yourself, record answer in another array and use data to “look-up” the square

```c
// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};
// The LUT
int squares[6] = {0, 1, 4, 9, 16, 25};

for(int i=0; i < 8; i++){
    int x = data[i]
    int x_sq = squares[x];
    cout << i << "," << sq[i] << endl;
}

// the data
int data[8] = {3, 2, 0, 5, 1, 4, 5, 3};
// The LUT
int squares[6] = {0, 1, 4, 9, 16, 25};

for(int i=0; i < 8; i++){
    int x_sq = squares[data[i]];
    cout << i << "," << sq[i] << endl;
}
```
Example

• Using an array as a Look-Up Table
  – wget http://ee.usc.edu/~redekopp/cs101/cipher.cpp
  – Let’s create a cipher code to encrypt text
  – abcdefghijklmnopqrstuvwxyz => ghijklmaefnzyqbcdrstuopvwx
  – char orig_string[] = “helloworld”;
  – char new_string[11];
  – After encryption:
    • new_string = “akzzbpbbrzj”
  – Define another array
    • char cipher[27] = “ghijklmaefnzyqbcdrstuopvwx”;
    • How could we use the original character to index (“look-up” a value in) the cipher array
Enumerations

- Associates an integer (number) with a symbolic name
  
  ```
  enum [optional_collection_name] {Item1, Item2, ... ItemN}
  - Item1 = 0
  - Item2 = 1
  - ...
  - ItemN = N-1
  ```

- Use symbolic item names in your code and compiler will replace the symbolic names with corresponding integer values

```cpp
// First enum item is associated with 0
enum Colors {BLACK,BROWN,RED,...,WHITE};

int pixela = RED;  // pixela = 2;
int pixelb = BROWN; // pixelb = 1;
```

```
const int BLACK=0;
const int BROWN=1;
const int RED=2;
const int WHITE=7;

int pixela = RED;
int pixelb = BROWN;
...
```

Hard coding symbolic names with given codes

Using enumeration to simplify
Review on your own...

COMMON ARRAY DESIGN PATTERNS
Design Pattern: Search

- A design pattern is a common recurrence of an approach
- Search: Find one item in an array/list/set of items
- Pattern:
  - Loop over each item likely using an incrementing index
  - For each item, use a conditional to check if it matches the search criteria
  - If it does match, take action (i.e. save index, add value to some answer, etc.) and possibly break, else, do nothing, just go on to next

```cpp
// search 'data' array of size 'len' for 'target' value
bool search(int data[], int len, int target) {
    bool found = false;
    for(int i=0; i < len; i++){
        if(data[i] == target){
            found = true;
            break;
        }
    }
    return found;
}
```
Design Pattern: Search

• What's not a search:
  – Indicating the search failed if a single element doesn't match
  – Consider data = \{4, 7, 9\} and target = 7
  – 4 won't match and set found=false and stop too soon

```cpp
// search 'data' array of size 'len' for 'target' value
bool search(int data[], int len, int target)
{
    bool found = false;
    for(int i=0; i < len; i++){
        if(data[i] == target)
            return true;
        else
            return false;
    }
    return false;
}
```
Design Pattern: Search

- What's not a search:
  - Indicating the search failed if a single element doesn't match
  - Consider data = {4, 7, 9} and target = 7
  - 4 won't match and set found=false and stop too soon
  - 7 will match and set found = true, but only for a second...
  - 9 won't match and set found = false...forgetting that 7 was found

```cpp
// search 'data' array of size 'len' for 'target' value
bool search(int data[], int len, int target)
{
    bool found = false;
    for(int i=0; i < len; i++){
        if(data[i] == target)
            found = true;
        else
            found = false;
    }
    return found;
}
```
Design Pattern: Search

• What's not a search:
  – **Declaring your result variable inside the for loop**
  – **Bool** found only lives in the current scope (i.e. the 'if' statement and will not be visible afterwards when you need it

```java
// search 'data' array of size 'len' for 'target' value
for(int i=0; i < len; i++){
    if(data[i] == target)
        bool found = true;
    break;
}  // found is deallocated here..too early!
// check found for result of search
```
Design Pattern: Reduction

• Reduction: Combine all items in an array/list/set to produce one value (i.e. sum, check if all meet a certain criteria, etc.)

• Pattern:
  – Declare a variable to hold the reduction
  – Loop over each item likely using an incrementing index
  – For each item, combine it appropriately with your reduction variable

```c
// sums 'data' array of size 'len'
int sum = 0;
for(int i=0; i < len; i++){
    sum = sum + data[i];  // sum += data[i]
}
// use sum
```
Design Pattern: Reduction

• Reduction: Combine all items in an array/list/set to produce one value (i.e. sum, check if all meet a certain criteria, etc.)

• Pattern:
  – Declare a variable to hold the reduction
  – Loop over each item likely using an incrementing index
  – For each item, combine it appropriately with your reduction variable

```cpp
// checks if all elements are positive
bool allPos = true;
for(int i=0; i < len; i++){
    allPos = allPos && (data[i] > 0);
}
```

• Could also be accomplished as a search for a negative