• input
• reading input from the keyboard
• the std:: class scope notation
• reading strings from the keyboard
• C style strings
• C string library
• shortcut operators
• formatted output
• multidimensional arrays

using cin
– cin is a function that is a standard part of the C++ language, like cout
– cin reads input from the keyboard

• arguments
– >> followed by a variable
– tell cin what to read from the keyboard and where to store it

• example:
#include <iostream>
using namespace std;
int main() {
    int i;
    cout << "enter a number: ";
    cin >> i;
    cout << "you entered: " << i << endl;
} // end of main()

• you can use cin to read integers and characters, as well as other simple data types (e.g., float, double)
• if you want to read in multiple values, you can use multiple calls to cin or one call with multiple parameters
• example of making multiple calls:
    int i, j, k;
    cout << "enter three numbers: ";
    cin >> i;
    cin >> j;
    cin >> k;
• when you put this code in a program and run it, you can enter three numbers separated by whitespace (space, tab or return)
• example of making one call with multiple parameters:
    int i, j, k;
    cout << "enter three numbers: ";
    cin >> i >> j >> k;
• as above, when you put this code in a program and run it, you can enter three numbers separated by whitespace (space, tab or return)

the std:: class scope notation
• both cout and cin belong to something called the std class
• you may see notation like this: std::cout or std::cin.
• the std:: prefix uses something called the scope operator, which is the :: part. it tells the compiler that cout and cin are part of the std class.
• for example:
#include <iostream>
using namespace std;
int main() {
    int i;
    std::cout << "please enter a number: ";
    std::cin >> i;
    std::cout << "your number is " << i << endl;
} // end of main()
• in most cases, `std::cout` means the same thing as `cout` and `std::cin` means the same thing as `cin`.
• the scope operator is used to disambiguate, just in case there were two things defined using the name `cout` or the name `cin`.
• you can think of it like having two people in the class with the same first name. we need their last names to figure out who we are talking about.
• the programs that you write for this class will NOT define anything else called `cout` or `cin`, so you don’t need to worry about the `std::` prefix from that standpoint. however, the textbook uses that notation; and it is good to know anyway.

reading strings from the keyboard
• there are two ways to read input values from the keyboard into a string variable:
  [1] using `cin >>`
  [2] using the `getline()` function
• the first way, using the `>>` operator, will only read until the first whitespace character is read (the term “whitespace” refers to characters like blank spaces, tabs and newlines); this means that the input will stop as soon as the first whitespace character is read.
• however, when reading a string variable using the `getline()` function, the input will stop as soon as the first newline character is read (i.e., the user hits the ENTER key on the keyboard), e.g.:

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s;
    cout << "please enter your name:"; cin >> s;
    cout << s << endl;
} // end of main()
```

• if the user enters `david ortiz` when the program asks please enter your name:
then the value of `s` will be "david"
• HOWEVER, when reading a string variable using the `getline()` function, the input will stop as soon as the first newline character is read (i.e., the user hits the ENTER key on the keyboard), e.g.:

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string s;
    cout << "please enter your name:"; getline( cin, s );
    cout << s << endl;
} // end of main()
```

C style strings
• last class, we covered C++ style strings
• today, we’ll talk about C style strings
• both types of strings allow you to store multiple characters in a single variable
• the underlying data type is still `char`, but only one variable name is used, and a length is associated with it
• in a C++ style string, you can use the string `length()` function
• in a C style string, the last character is a terminator: ‘\0’
you can also use the constant called `NULL`
• with both styles, string constants are surrounded by double quotes: "
• example:

```cpp
string s1 = "ABCDE"; // C++ style string
char s2[6] = "FGHIJ"; // C style string
```
• storage of the C style string:
  ```
  char s2[6] = "FGHIJ";
  ```
looks like this:  

F  G  H  I  J  \0

• so with strings, you really only access the values stored at indices 0 through length − 1, (or size - 2) since the value stored at length is always \0

• NOTICE that you have to make your C style string one character longer than you need, to make room for the terminator character

so in the example, we wanted to have 5 characters in our string ("FGHIJ"), but we declared the string s2 to be of length 6, in order to have room for the NULL (\0) terminator

---

C string library

• in addition to the C++ string library that we talked about last class, there is also a C string library

• to use the C string library, include the header in your C++ source file:
  ```
  #include <cstring>
  ```

• this will give you access to a lot of functions, including the following:
  ```
  strlen()
  strcmp()
  strcpy()
  strncpy()
  ```

---

• string length function:
  ```
  int strlen( char *s );
  ```
  this function returns the number of characters in s; note that this is NOT the same thing as the number of characters allocated for the string array

• string comparison function:
  ```
  int strcmp( const char *s1, const char *s2 );
  ```
  “This function returns an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared.”

• copying functions:
  ```
  char *strcpy( char *dest, char *source );
  ```
  − copies characters from source array into dest array up to NULL
  ```
  char *strncpy( char *dest, char *source, int num );
  ```
  − copies characters from source array into dest array; stops after num characters (if no NULL before that); appends NULL

---

shortcut operators

• also called “increment” and “decrement” operators

• increment operator: ++
  meaning: add one and assign
  example: i++;
  is the same as: i = i + 1;

• decrement operator: --
  meaning: subtract one and assign
  example: i--;
  is the same as: i = i - 1;
formatted output

- part of iostream library
- `cout.setf()` defines the type of output field
- `cout.precision()` sets the decimal precision (for real numbers)
- `cout.width()` sets the width of the output field

example:
```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "here’s a table with lined-up columns:\n";
    cout.width( 10 );
    cout << "monday";  
    cout.width( 10 );
    cout << "tuesday";
    cout.width( 10 );
    cout << "wednesday";
    cout << endl;
    cout.width( 10 );
    cout << "1";
    cout.width( 10 );
    cout << "2";
    cout.width( 10 );
    cout << "3";
    cout << endl;
}
```

output:
here’s a table with lined-up columns:

```
monday  tuesday  wednesday
1       2       3
```

another example:
```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    const int A = 5;
    const double B = 3.4568;
    double C;
    cout << "output using fixed precision, 2 decimal places:\n";
    cout.setf( ios::fixed );
    cout.precision( 2 );
    cout << "B=" << B << endl;
    cout << "output using width=10, left justified:\n";
    cout.setf( ios::left );
    cout.width( 10 );
    cout << "B= " << B << endl;
    cout << "output using width=10, right justified:\n";
    cout.setf( ios::right );
    cout.width( 10 );
    cout << "B= " << B << endl;
    C = sin( B );
    cout.setf( ios::right );
    cout.width( 10 );
    cout << "C= " << C << endl;
}
```

output:
output using fixed precision, 2 decimal places:

```
B=3.46
```
output using width=10, left justified:

```
B= 3.46
```
output using width=10, right justified:

```
B=3.46
```
you have to repeat the formatting if you want the same thing again:

```
C=-0.31
```
multidimensional arrays

- so far, we have only talked about one-dimensional arrays, where you can think of the data as being stored in EITHER a single row OR a single column in spreadsheet
- but we can also define multi-dimensional arrays (arrays with more than one dimension)
- today we’ll talk about two-dimensional arrays.
- note that you can define multi-dimensional arrays (i.e., more than 2 dimensions), but that is beyond the scope of this class
- with two-dimensional arrays, you can think of it like data being stored in BOTH rows and columns.
- you define a two-dimensional array by adding another set of square brackets, with a value in between indicating the size of the dimension, for example:
  ```
  int myArray[3][4];
  ```
  which declares a two-dimensional array with 3 rows and 4 columns
- we will refer to the two dimensions as rows and columns so that you can visualize a spreadsheet, like this:

```
3 1 1 9
2 5 6 8
5 7 4 2
```

- C++ defines arrays as “row major”, which means that the row dimension comes first, then the column dimension
- example:
  ```
  #include <iostream>
  using namespace std;
  #include <time.h>
  #include <stdlib.h>

  // declare global variables
  const int LENGTH = 3;
  const int WIDTH = 4;

  int main() {
    int myArray[LENGTH][WIDTH];
    srand( time( NULL ));

    // initialize array
    for ( int y=0; y<LENGTH; y++ ) {
      for ( int x=0; x<WIDTH; x++ ) {
        myArray[y][x] = ( rand() % 10 ) + 1;
      } // end for x
    } // end for y
    // print array
    for ( int y=0; y<LENGTH; y++ ) {
      for ( int x=0; x<WIDTH; x++ ) {
        cout << "(" << x << ", " << y << ") = " << myArray[y][x] << " 
";
      } // end for x
      cout << endl;
    } // end for y

    return 0;
  }
  ```