
- Chapter 1
  - Computer Systems (hardware and Software components, memory types)
    - PC, Workstation, Mainframe, Network, input/output
    - memory – primary, secondary, fixed, removable
    - CPU
    - Why 8Bits/9Bits = 1Byte?
  - Programming and Problem Solving using Computers
  - Introduction to C++
  - Program Editing, Compiling, Testing, Debugging, Re-designing, and program/algorithm analysis

- Chapter 2
  - Variables, names, memory addresses, assignments
  - Standard I/O (input/output)
  - Data types, expressions, arithmetic operators
  - Simple flow of control (conditional statements, loops)
  - Programming style, comments, indenting, headers, naming conventions
  - American Standard Code for Information Interchange (ASCII Character set)

- Chapter 4
  - Void type functions
  - Formal parameters, function prototype and function header definitions
  - Precondition and postcondition
  - Function Call-by-value & Call-by-reference
  - Driver (test) programs & Stubs (fake subroutines)


- Chapter 5
  - Basic File Input/Output, I/O Streams
  - External file names and local stream names
    ```
    #include <fstream>
    ifstream in_stream;
    in_stream.open("infile.dat");
    if (in_stream.fail()) {
      ... exit(1); }
    else { in_stream >> next; while (next!=eof) { ... } }
    in_stream.close();
    ```
  - Character I/O: `char c=cin.get(); cout.put(c);`
  - `endl` vs. `\n` vs. `'\n'`
  - The boolean functions `isalpha(char_expr), isdigit(char_expr), etc.`


- Structures
  ```
  struct CDAccount // structure tag
  { double balance; // Public: Member names
    Money cash; // Hierarchical Structures
    double interest_rate;
    int computeOneYearInterest(); // Public Method
    private:
    int term;
    // months until maturity
  };
  ```
  - Classes, objects, members (variables and functions)
  - Differences between classes and structures
  - Inheritance
  - String Encoding, Transmission, Decoding Example.
Chapter 7

- Extended Flow Control (if-else, which-case-break)
- Truth Tables (&& || !), boolean arithmetic/expressions
- Enumeration type

```cpp
eenum MonthLength { JAN_LEN = 31, FEB_LEN = 28, MAR_LEN = 31, APR_LEN = 30, ... NOV_LEN = 30, DEC_LEN};
```
- All kinds of loops (for, do-while, while)
- Debugging nested loops (separate loops and print-test each)
- Loop termination criteria (List Headed by fixed-size; Ask before iterating; List ended by sentinel value; Running out of input (End of file.).)
- Most common problems with loops (off-by-one error, infinite loops)

Chapter 9 & 8

- Declaring and Referencing Arrays
- Why use arrays? Why sort arrays?
- Arrays in Memory

```cpp
adr( a[k] ) = adr( a[0] ) + k*ElementByteSize
```
- Initializing Arrays
- Entire arrays (or elements-of-arrays)
  in Function calls
- The const Parameter Modifier
- Array descriptors: name, size, type, scope
- Operator overloading for ADT's
- Arrays of Classes (arrays of objects)
- Classes of arrays (classes having arrays as members)

Chapter 10

Strings and Multidimensional Arrays

- String Basics
  - cstring Values, cstring Variables, and ANSI C++ string class
  - Predefined cstring Functions
  - Defining cstring Functions
  - cstring input and output
  - cstring-to-Number Conversions and Robust Input
- Multidimensional Arrays
  - Multidimensional Array Basics
  - Arrays of cstrings
- The C++ Standard string class
  - Interface for the Standard string class
  - Arrays of string revisited
  - Namespaces Revisited
10 Strings and Multidimensional Arrays

- We will refer to strings we have dealt with so far as cstrings (character arrays). ANSI C++ Library provides a string class which is introduced in this chapter. We refer to these as simply strings.
- In this chapter we study the ANSI C++ string class and arrays with more than one index.
- Arrays with more than one index called multidimensional arrays (e.g., 3D arrays, 4D arrays, etc.)

10.1 String Basics

cstring Values and cstring Variables (1 of 5)

- ANSI C++ Library provides a string class which is introduced in this chapter. We refer to these as strings.
- We will refer to strings we have dealt with so far as cstrings.
- Members of the C++ Standard Library’s a class string are declared in the header <string>.
- Technically, cstrings are null terminated char arrays.
- In the example,
  ```
  char x[] = "Enter the input.;"
  "Enter the input:" is a cstring literal. The variable x is a cstring.
  ```

cstring Values and cstring Variables (2 of 5)

- A cstring variable is a partially filled array having base type char
- Any array uses positions having index values 0 through one less than the number used.
- The cstring variable signals the last used position by placing the special character, called the null character ‘\0’ in the array one position beyond the last character of the cstring.
- If we declare and initialize a cstring variable s:
  ```
  ```
- If s contains “Hi, Mom!” then the array elements are:
  ```
  Hi   , M om   ! \0
  ```
- The character ‘\0’ is the sentinel marking the end of the cstring.

cstring Values and cstring Variables (3 of 5)

- It is possible to initialize a cstring variable at declaration:
  ```
  char my_message[20] = "Hi there.;"
  ```
- Notice that cstring variables need not fill the entire array.
- The compiler will count the characters in the initializing string and add one for the null character:
  ```
  char short_string[4] = "abc";
  ```
- This is equivalent to
  ```
  char short_string[4] = {'a', 'b', 'c', '\0'};
  ```
- You must leave space for the null character when you specify size.

cstring Values and cstring Variables (4 of 5)

- Do not confuse these situations:
  ```
  char a_string[] = "abc";  // Inserts terminator ‘\0’
  char not_a_string[4] = {'a', 'b', 'c'};  // Does not insert ‘\0’
  ```
- These are NOT equivalent.
- The first one of these initializations places the required null terminating character ‘\0’ after the ‘a’, ‘b’, and ‘c’. The result is a cstring.
- The second leaves space for the ‘\0’ null character, but it does not insert the null character. The result is NOT a cstring.

Review, Tuesday, June 26, 2001
PIC 10 B

- Reviewed Chapters 05-09, PIC 10A
  (most importantly Ch. 05, Classes and basic inheritance, Ch. 8, operator overloading, friend functions).
- cstrings:
  ```
  char my_message[20] = "Hi there.;"
  char short_string[4] = "abc";  // 4-element char array
  char short_string[4] = {'a', 'b', 'c', '\0'};
  ```
cstring Values and cstring Variables (5 of 5)

- A cstring is an ordinary array with base type char, and may be processed one element at a time:
  
  ```
  int index = 0;
  while (our_string[index] != '0')
  {
    our_string[index] = 'X';
    index++;
  }
  ```

- In processing cstrings take great care not to overwrite the null character. An array that was a cstring that has its terminating character overwritten is NO LONGER a cstring.

- In the loop above, if our_string has no null terminator, the loop will run off into memory, happily writing on every byte in memory beyond the end of our_string until a byte is found with zero value.

PITFALL:
Using = and == with cstrings (1 of 5)

- Values and variables of type cstring when used with = (assignment) and == (comparison for equality) do not behave like built-in (primitive) data types.

- Assigning a value to a cstring in the obvious way is illegal:
  
  ```
  char a_string[10] = "hello"; // ILLEGAL
  ```

- Initializing at declaration is straightforward:
  
  ```
  char a_string[10] = "DoBeDo";
  ```

- The = does not mean assignment, in spite of the appearance.

- In Chapter 11 we will see that in C++, assignment and initialization can have very different behavior.

PITFALL:
Using = and == with cstrings (2 of 5)

- Assignment can be done barehanded, an element at a time:
  
  ```
  char a_string[10] = "Hello";
  char b_string[10];
  int i = 0;
  while(a_string[i] != '0')
  {
    b_string[i] = a_string[i];
    i++;
  }
  ```

- There is a predefined function, strcpy, to assign cstrings:
  
  ```
  char a_string[10] = "Hello";
  char b_string[10];
  ```

PITFALL:
Using = and == with cstrings (3 of 5)

- Comparison of cstrings cannot be done with the == operator. The attempt to compare cstrings with == compiles, it does not get the results you expect.

- Array names carry the address value of the first array element. The result of using == depends on where in memory the cstrings are stored.

- Use of the predefined comparison function strcmp.
  
  ```
  char a_string[10] = "aeolian";
  char b_string[10] = "aeonian";
  if (strcmp(b_string, a_string))
  cout << "The strings are NOT the same.";
  else
  cout << "The strings are the same.";
  ```

- The strcmp function is declared in the <cstring> header.
**PITFALL:**
Using `=` and `==` with cstrings (4 of 5)
- `strcmp` compares cstrings in lexicographic order.
- For successive values of `i` starting at 0, `cstring1[i]` and `cstring2[i]` are compared:
  - If `cstring1[i] < cstring2[i]`, `strcmp` returns a negative number.
  - If `cstring1[i] > cstring2[i]`, `strcmp` returns a positive number.
  - The number may be -1 or +1, or the difference of the encoding of `cstring1[i]` - `cstring2[i]`, or some other value.
- The actual value returned depends on the implementation.
- Do not write code that depends on the value returned.
- Testing then stops.
- If the cstrings are equal up to the end of one of them, the value returned indicates the longer string is greater than the shorter string.
- If the strings are equal in length and have the same characters, the strings are **equal**.

**Predefined cstring Functions (5 of 5)**
- Display 10.1 (next slide) contains a few of the functions from the cstring library.
- You must include `<cstring>` to gain access to these functions.
- `strcpy(target, source)` replaces target with source. Be sure there is enough space in target to hold all of source.
- `strcat(target, source)` appends source to target. The first character of source is copied into the null terminator of target, and all successive characters of source are copied into target. Be sure there is enough space in target for all of source's characters, including source's null terminator.
- `strlen(source)` returns the number of characters up to but not including the null terminator.
- `strcmp(str1, str2)` We discussed this in an earlier slide. Refer to Display 10.1 for detail.

**Display 10.1 Some Predefined Cstring Functions in Cstring**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcpy()</code></td>
<td>Copies the cstring value <code>source</code> into the cstring variable <code>target</code>.</td>
<td></td>
</tr>
<tr>
<td><code>strcat()</code></td>
<td>Concatenates the cstring value <code>source</code> onto the end of the cstring in the cstring variable <code>target</code>.</td>
<td></td>
</tr>
<tr>
<td><code>strlen()</code></td>
<td>Returns the number of characters up to but not including the null terminator.</td>
<td></td>
</tr>
<tr>
<td><code>strcmp()</code></td>
<td>Compares two cstrings. If they are equal, returns 0; otherwise, returns a negative or positive number.</td>
<td></td>
</tr>
</tbody>
</table>

**PITFALL:**
Dangers in Using Functions from `<cstring>`
- There is a very real danger associated with the functions strcpy and strcat.
- Both these functions copy characters until a null character is found in the source string, without regard to whether space is available in the target.
- If there is no space in the target, strcpy and strcat will happily overwrite any variables in memory beyond the target array.
- This may be some of your variables, or it could be something that your system depends on to run correctly.
- There could be no effect whatsoever.
- There could be a segmentation violation or illegal operation error, with your program crashing, and no further problems.
- The operating system could crash and burn.
- Nothing apparent may happen. But the next application started could crash and burn on loading. So, be careful!

**Defining cstring functions**
- The strcpy and strcat functions have problems.
- The Standard Library defines versions that have an additional parameter that can avoid some of these problems.
- To learn to write safe cstring functions, we write a string_copy function with an additional parameter to make the function safer.
- The added parameter takes an argument that is the declared size of the target argument.
Display 10.2 The function string_copy (1 of 2)

// Program to demonstrate the function string_copy
#include <iostream>
#include <cstring>

void string_copy(char target[], const char source[], int target_size);

// Precondition: target_size is the declared size of the cstring variable target.
// The array source contains a cstring value terminated with '\0'.
// Postcondition: The value of target has been set to the cstring value in source,
// provided the declared size of target is large enough. If target is not large
// enough to hold the entire cstring, a cstring equal to as much of the value of
// source as will fit is stored in target.

int main()
{
    using namespace std;
    char short_string[11]; // Can hold cstrings of up to 10 characters.
    string_copy(short_string, "Hello", 11);
    cout << short_string << " STRING ENDS HERE.\n";
    char long_string[] = "This is rather long.\n";
    string_copy(short_string, long_string, 11);
    cout << short_string << " STRING ENDS HERE.\n";
    return 0;
}

Display 10.2 The function string_copy (2 of 2)

// Uses header file cstring or string.h:
void string_copy(char target[], const char source[], int target_size)
{
    using namespace std;
    int new_length = strlen(source);
    if (new_length > (target_size - 1))
        new_length = target_size - 1; // That is all that will fit.
    int index;
    for (index = 0; index < new_length; index++)
    {
        target[index] = source[index];
    }
    target[index] = '\0';
}

cstring Input and Output (1 of 3)

• cstrings may be output using the insertion operator «<<
cout << short_string << " STRING ENDS HERE.\n";
• cstrings may receive input using the extraction operator >>
cin >> short_string >> some_other_string;
• HOWEVER: Remember that extraction ignores all white space, and
  that extraction from istream objects stops at whitespace.
• Whitespace is blanks, tabs, and line breaks.
• The code
  char a[80], b[80];
cin.getline(a, 80);
cout << a << b << "END OF OUTPUT.\n";
produces a dialog like:

Do be do to you! Do be do to you! END OF OUTPUT.


cstring Input and Output (2 of 3)

• To get an entire line, you can write a loop to extract the line a word at
  a time, but this won’t read the blanks.
• To get an entire line, you can use the predefined member getline.
  getline has two arguments: a cstring and a number of characters to
  extract to the cstring, allowing for the null terminator.
• Typically this is the declared size of the variable

Example: This code
char a[80];
cin.getline(a, 80);
cout << a << "END OF OUTPUT.\n";
produces a dialog like:

Do be do to you! Do be do to you! END OF OUTPUT.


cstring Input and Output (3 of 3)

• The getline member function stops reading when a number of
  characters equal to the second argument have been read:

Example: This code
char a[80];
cin.getline(a, 5);
cout << a << "END OF OUTPUT.\n";
produces a dialog like:

Do be do to you! Do be do to you! END OF OUTPUT.

• These cstring i/o techniques work the same for file i/o:
• If in_stream has been declared and connected to a file, this code will
  input 79 or fewer characters (up to the end of line) into cstring
  variable a.
char a[80];
in_stream.getline(a, 80);


cstring-to-number Conversions and Robust Input (1 of 3)

- '1', '1' and 1 are all different.
- 1 is an int constant, also called a literal.
- '1' is a char constant. It occupies one byte and is represented by some encoding. In C++ the value is the ASCII encoding, which has the decimal value 49. Recall we talked about ASCII encoding in PIC10A.
- "1" is a cstring constant. It occupies two bytes, one for the encoding of the character 1 and one for the null terminator.
- In a program in any language, you cannot ignore the difference between these objects.
- Robust numeric input may be written by inputting a cstring, extracting the digit characters and building the number from the digits.

cstring-to-number Conversions and Robust Input (2 of 3)

- Once you have a cstring containing the digits that represent an int, use the predefined function atoi
- atoi is named and pronounced: Ascii TO Integer
- atoi takes a cstring argument and returns the int value represented by the digit characters in cstring.
- atoi returns 0 if the cstring contains a non-digit character.
- Example: atoi("#37") returns 0.
- The atoi function is declared in the <cstdlib> header.

cstring-to-number Conversions and Robust Input (3 of 3)

- The function atof is named and pronounced: Ascii TO Floating point.
- atof is similar to atoi. It converts its cstring argument to the double value the cstring represents. Like atoi, the function atof returns 0.0 if the cstring argument does not represent a double.
- Display 10.3 demonstrates read_and_clean, and Display 10.4 demonstrates Robust Input Functions.

Display 10.3 cstrings to Integers (1 of 3)

```cpp
// Demonstrates the function read_and_clean.
#include <iostream>
#include <cstdlib>
#include <cctype>

void read_and_clean(int& n);
// Reads a line of input. Discards all symbols except the digits. Converts the cstring to an integer and sets n equal to the value of this integer.
void new_line( );
// Discards all the input remaining on the current input line.
// Also discards the "n" at the end of the line.

int main( )
{
    using namespace std;
    int n;
    char ans;
    do
    { cout << "Enter an integer and press return: ";
        read_and_clean(n);
        cout << "That string converts to the integer " << n << endl;
        cout << "Again? (yes/no): ";
        cin >> ans;
    } while ( (ans != 'n') && (ans != 'N') );
    return 0;
}
```

Display 10.3 cstrings to Integers (2 of 3)

```cpp
// Uses iostream, cstdlib, and cctype:
void read_and_clean(int& n)
{
    using namespace std;
    const int ARRAY_SIZE = 6;
    char digit_string[ARRAY_SIZE];
    char next;
    cin.get(next);
    int index = 0;
    while (next != '
')
    { if (isdigit(next) && (index < ARRAY_SIZE - 1))
        { digit_string[index] = next;
            index++;
        }
        cin.get(next);
    }
    digit_string[index] = '0';
    n = atoi(digit_string);
}
```

Display 10.3 cstrings to Integers (3 of 3)

```cpp
// Uses iostream:
void new_line( )
{
    using namespace std;
    char symbol;
    do
    { cin.get(symbol);
    } while (symbol != "\n");
}
```
Demonstration program for improved version of get_int.

```cpp
#include <iostream>
#include <cstdlib>
#include <cctype>

// Reads a line of input. Discards all symbols except the digits. Converts
// the cstring to an integer and sets n equal to the value of this integer.

void read_and_clean(int& n);

// Discards all the input remaining on the current input line.
// Also discards the 'n' at the end of the line.
void get_int(int& input_number);

int main()
{ 
  using namespace std;
  int input_number;
  get_int(input_number);
  cout << "Final value read in = " << input_number << endl;
  return 0;
}
```

Uses iostream, cstdlib, and cctype:

```cpp
//Uses iostream, cstdlib, and cctype:
void read_and_clean(int& n)
{
  using namespace std;
  const int ARRAY_SIZE = 6;
  char digit_string[ARRAY_SIZE];
  char next;
  cin.get(next);
  int index = 0;
  while (next != '
')
  {
    if (isdigit(next)) && (index < ARRAY_SIZE - 1)
    {
      digit_string[index] = next;
      index++;
    }
    cin.get(next);
  }
  digit_string[index] = 0;
  n = atoi(digit_string);
}
```

Multidimensional Array Basics (1 of 2)

It is useful to have an array with more than one index. In C++, this is implemented using an array with an array type as base type. Such an array is declared as following:

```
char page[30][100];
```

The indexed variables for this array are:

```
page[0][0] , page[0][1] , . . . page[0][99]
page[1][0] , page[1][1] , . . . page[1][99]
```

Column index \(k\)
Multidimensional Array Basics (2 of 2)

- We said that a two-dimensional array is an array with a base type that is an array type. In other words, two-dimensional array is an array of arrays.
- The array `char page[30][100];` is a one dimensional array of size 30, whose base type is an array of size 100 with base type char.
- Each entry in the array of size 30 is an array of char of size 100.
- Most of the time, the programmer can treat a two-dimensional array as if it were an array with two indices.
- There are two situations where being an arrays of arrays is evident:
  - One is when a function with an array parameter for a two dimensional array: `void display( const char p[][100], int size);`
  - With a two-dimensional array parameter the first dimension is ignored even if specified, and the compiler does not use it. This necessitates a size parameter.
  - This makes sense if you think of the multidimensional array parameter as an array of arrays. The first dimension is the index, the rest describe the base type.
- With a higher-dimension array parameters the first dimension is usually not specified, but all the rest of the dimensions must be specified.

Display 10.5 presents a program that uses a two-dimensional array named `grade` to store then display grade records for a small class.

The first index designates a student, the second designates a grade.

The grade of student 4 on quiz 1 is recorded in `grade[3][0]`.

The program has an array, `quiz_ave` to hold a list of class averages for each quiz over all student grades in the class, and an array `st_ave` to hold a list of student averages over the quizzes that student has taken.

Multidimensional Array Parameters

When a multidimensional array parameter is given in a function heading or prototype, the size of the first dimension is not given, but the remaining dimension sizes must be given in square brackets. Since the first dimension size is not given, you usually need an additional parameter of type int that gives the size of this first dimension. Below is an example of a function prototype with a two-dimensional array parameter `p`:

```c
void get_page(char p[][100], int size_dimension);
```

A Programming Example

A Two-Dimensional Grading Program.

- Display 10.5 presents a program that uses a two-dimensional array named `grade` to store then display grade records for a small class.
- The first index designates a student, the second designates a grade.
- The grade of student 4 on quiz 1 is recorded in `grade[3][0]`.
- The program has an array, `quiz_ave` to hold a list of class averages for each quiz over all student grades in the class, and an array `st_ave` to hold a list of student averages over the quizzes that student has taken.

Multidimensional Array Basics (1 of 2)

- We said that a two-dimensional array is an array with a base type that is an array type. In other words, two-dimensional array is an array of arrays.
- The array `char page[30][100];` is a one dimensional array of size 30, whose base type is an array of size 100 with base type char.
- Each entry in the array of size 30 is an array of char of size 100.
- Most of the time, the programmer can treat a two-dimensional array as if it were an array with two indices.
- There are two situations where being an arrays of arrays is evident:
  - One is when a function with an array parameter for a two dimensional array: `void display( const char p[][100], int size);`
  - With a two-dimensional array parameter the first dimension is ignored even if specified, and the compiler does not use it. This necessitates a size parameter.
  - This makes sense if you think of the multidimensional array parameter as an array of arrays. The first dimension is the index, the rest describe the base type.
- With a higher-dimension array parameters the first dimension is usually not specified, but all the rest of the dimensions must be specified.
**Display 10.5 Two-dimensional Array (2 of 5)**

```
void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[]);
// Precondition: Global constant NUMBER_STUDENTS and NUMBER_QUIZZES
// are the dimensions of the array grade. Each of the indexed variables
// grade[st_num-1, quiz_num-1] contains the score for student st_num
// on quiz quiz_num.
// Postcondition: Each quiz_ave[quiz_num-1] contains the average for quiz
// numbered quiz_num.
void display(const int grade[][NUMBER_QUIZZES],
const double st_ave[], const double quiz_ave[]);
// Precondition: Global constant NUMBER_STUDENTS and
// NUMBER_QUIZZES are the dimensions of the array grade.
// Each of the indexed variables grade[st_num-1, quiz_num-1] contains
// the score for student st_num on quiz quiz_num. Each st_ave[st_num-1]
// contains the average for student stu_num. Each quiz_ave[quiz_num-1]
// contains the average for quiz numbered quiz_num.
// Postcondition: All the data in grade, st_ave, and quiz_ave have been output.
```

**Display 10.5 Two-dimensional Array (3 of 5)**

```
int main()
{
    using namespace std;
    int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
    double st_ave[NUMBER_STUDENTS];
    double quiz_ave[NUMBER_QUIZZES];
    grade[0][0] = 10; grade[0][1] = 10; grade[0][2] = 10;
    grade[1][0] = 2; grade[1][1] = 0; grade[1][2] = 1;
    grade[2][0] = 8; grade[2][1] = 6; grade[2][2] = 9;
    grade[3][0] = 8; grade[3][1] = 4; grade[3][2] = 10;
    compute_st_ave(grade, st_ave);
    compute_quiz_ave(grade, quiz_ave);
    display(grade, st_ave, quiz_ave);
    return 0;
}
```

**Display 10.5 Two-dimensional Array (4 of 5)**

```
void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[])
{
    for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
    { // Process one st_num:
        double sum = 0;
        for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
        { // sum contains the sum of the quiz scores for student number st_num.
            sum += grade[st_num-1][quiz_num-1];
        }
        st_ave[st_num-1] = sum / NUMBER_QUIZZES;
    }
}
```

**Display 10.5 Two-dimensional Array (5 of 5)**

```
void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[])
{
    for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
    { // Process one quiz (for all students):
        double sum = 0;
        for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
        { // sum contains the sum of all student scores on quiz number quiz_num.
            sum += grade[st_num-1][quiz_num-1];
        }
        quiz_ave[quiz_num-1] = sum / NUMBER_STUDENTS;
    }
}
```

**Display 10.6 The Two-dimensional Array grade**

```
// Uses iostream and iomanip:
void display(const int grade[][NUMBER_QUIZZES],
const double st_ave[], const double quiz_ave[])
{
    using namespace std;
    cout << setprecision(1);
    cout << setw(10) << "Student" << setw(5) << "Ave" << setw(15) << "Quizzes
";
    for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
    { // Display for one st_num:
        cout << setw(10) << st_num << setw(5) << st_ave[st_num-1] << " ";
        for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
        { cout << setw(5) << grade[st_num-1][quiz_num-1];
        }
        cout << endl;
    }
    cout << "Quiz averages = ";
    for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
    { cout << setw(5) << quiz_ave[quiz_num-1];
    }
    cout << endl;
}
```
2 Dice Modeling Program
Write a program that simulates the rolling of two dice. The program should use `rand` to roll the first die and should use `rand` again to roll the second die. The sum of the two values should then be calculated. Note: Since each die can show an integer value from 1 to 6, then the sum of the two values will vary from 2 to 12, with 7 being the most frequent sum and 2 and 12 being the least frequent sums. The figure below shows the 36 possible combinations of the two dice. Our program should roll the two dice 36,000 times. Use a single-subscripted array to tally the numbers of times each possible sum appears. Print the results in a tabular format. Also, determine if the totals are reasonable (i.e., there are six ways to roll a 7, so approximately one sixth of all the rolls should be 7).

2 Dice Modeling Program – Problem Understanding
1. roll the two dice 36,000 times.
2. Tally the numbers of times each possible sum appears.
3. Print the results in a tabular format.
4. Determine if the totals are reasonable.
2 Dice Modeling Program – Modular Specifications - Main

Main Driver

- Instantiate two objects of type Die
- Request 36,000 die rolls from both objects
- Tally the observed sums and send them to Reporter-Module for Printing
- Return 0, if all is Okay.

2 Dice Modeling Program – Modular Specifications – Class Die

Class DIE

- Abstract class which allows us to roll any number of times a regular die and obtain the observed values.

Methods: int rollDie();

Variables: int die_roll_value;

Constructors: default Die();

Non-trivial: Die(int value);

2 Dice Modeling Program – Modular Specifications – Reporting Module

void reportModule(int i2, int i3, int i4, ..., int i12)
{ // Called with observed frequencies of Sums
  // Reporting in tabular form these and the
  // Expected sums for a pair of regular dice

  Reporting Module

}

2 Dice Modeling Program – Modular Specifications – Reasonableness Module

Deals with and discusses how reasonable are the OBSERVED Dice sums and the EXPECTED ones. Are the Dice fair?!?

Reasonableness Module

2 Dice Modeling Program – Modular Specifications - Implementation

Main Driver

Class DYE

Reporting Module

Reasonableness Module

2 Dice Modeling Program – Modular Specifications - Testing

Main Driver

Class DYE

Reporting Module

Reasonableness Module
Arrays of cstrings

- A cstring is an array of base type char.
- Consequently an array of cstrings is a two-dimensional array of base type char.
- A cstring must hold a null terminator, '\0', so each element of this array of 3 cstrings can hold at most 19 characters:
  ```
  char name[3][20];
  ```
- Like any array, you can manipulate an array of cstrings by using both index values in nested loops.
- It is nicer to treat the cstrings as entities:
  ```
  cout << "Enter 5 names, one per line:\n";
  for (int index = 0; index < 5; index++)
    cin >> name[index];
  ```
- Output to the screen is also straightforward:
  ```
  ```

10.3 The C++ Standard string class

- Using cstrings with predefined cstring functions is not as safe as we would like.
- Using strcpy to copy a longer cstring to another (shorter) cstring will overwrite memory that may be important to your program. If you are fortunate, it will be only your program that is the casualty. Your operating system may crash, or someone else’s program running on the same system could generate errors.

- The Standard Library supplied class, string, provides far more utility than the cstrings C++ gets by way of its C heritage.
- Class strings behave very much like built-in data types and are far safer than cstrings.
- Let s1, s2, and s3 be objects of class string, and suppose s1 and s2 have string values. Then + may be used for concatenation:
  ```
  s3 = s1 + s2;
  ```
- Additional space needed is allocated for s3 automatically.
- The default constructor (???) generates an empty string.
- There is a constructor that takes a cstring argument:
  ```
  string phrase, word1("Hello "), word2("World");
  phrase = word1 + word2;
  cout << phrase << endl;
  ```
- The output will be
  ```
  Hello World
  ```
However -- there is a considerable difference here:

There is no guarantee whether a + b or c + d is evaluated first, nor
Example:

ANSI C++ does not specify the order of evaluation for terms in an
expression. Writing code that depends on the order of evaluation is
illegal. Unfortunately, most compilers do not catch this error. The
reason is code parallelization and optimization.

Example:  (a + b) * (c + d)

There is no guarantee whether a + b or c + d is evaluated first, nor
does it make any difference in this case.

However -- There is a considerable difference here:

int i = 0;
cout << i << " " << i++ << endl;
// some compilers evaluate the expressions i and i++ right
// to left before calling the operator << overloading.
// giving the result 1 0
// A different compiler might give the result 0 1

The class string overloads the << (insertion) and
>> (extraction) operators with stream left
arguments and string right hand arguments with
familiar behavior.

Overloaded >> operator skips leading
whitespace and reads nonwhite characters up to
the next white space.

To get an entire line of input for cstrings, we
used the getline member of the istream class.

To get an entire line of input for class string
objects, we use a stand alone version of getline.

NOTE THAT class string objects do not range check index values.

If you want range checked indexing into strings, use the string
member function at(int index).

str1.at(9); // Checks index value 9 for legality in str1.
// If legal, returns the character at index value 9.

// If legal, returns the character at index value 9.
// inserting into str1 all input up to 'n'
getline(str1); // discards the '\n'

//A different compiler might give the result 0 1
// Some compilers evaluate the expressions i and i++ right
// to left before calling the operator << overloading.
// giving the result 1 0

// Some compilers evaluate the expressions i and i++ right
// to left before calling the operator << overloading.
Programming Tip
The Ignore Member Function
- With `cin >> intVariable`, everything entered beyond the integer just read in will still be available on the input stream, ready for further extraction. This includes the return key pressed to make the line of data available.
- This data will cause the getline function to misbehave.
- We presented one fix, the `new_line` function, remember?
- A standard fix is to use the predefined `cin` member function `ignore`, whose prototype is:
  ```cpp
  istream& ignore(int count, char delimiter);
  ```
- This function will read `count` characters unless it reads a delimiter character first. All the characters are discarded.

Pitfall
Mixing `cin >> variable` and `getline` can lose input.
- Careless mixed use of `cin >> variable` and `getline` can lose data in strange ways.
- `cin >> variable` skips leading whitespace and leaves the newline (`'n'`) character on the input stream.
- `getline` reads everything up to and including the `'n'`, keeps the data and discards the `'n'`.
- Use of `cin >> variable` leaves a `'n'` that makes a `getline` see an empty string.
- Use the `new_line` function from the text or:
  ```cpp
  cin.ignore(10000, 'n');
  ```
  to discard up to 10,000 characters or up to the newline.

Programming Example
Palindrome Testing (1 of 2)
- In PIC 10A we had a HW problem that determines whether a string is a palindrome.
- A palindrome has the same characters read front to back as it does read back to front. Examples (ignore punctuation, case, and blanks):
  - Able was I ere I saw Elba.
  - Madam, I’m Adam.
  - Rats live on no evil star.
- Back then we did not have access to the following string facilities:
  ```cpp
  string str; // default constructor - defines empty string
  getline(cin, str); // fetches an entire line of input
  isPal(str) // boolean function that tests for palindrome
  ```

Programming Example
Palindrome Testing (2 of 2)
- The `isPal` function defines a string containing the characters we want removed (punctuation and space)
- A palindrome string:
  ```cpp
  string reverse(const string& str);
  // returns a copy of arg corresponding to parameter
  // str with characters in reverse order.
  string removePunct(const string& src, const string& punct);
  // returns copy of string src with characters
  // in string punct removed
  string makeLower(const string& s);
  // returns a copy of parameter s that has all upper case
  // characters forced to lower case, other characters unchanged.
  // Uses <string>, which provides tolower
  bool isPal(const string& this_String);
  // this_String is a palindrome, return true; else return false;
  ```
- You can now test if a string is a palindrome:
  ```cpp
  void swap(char& lhs, char& rhs);
  // swaps char args corresponding to parameters lhs and rhs
  int main()
  { string str;
    cout << "Enter a candidate for palindrome test " << 
    "followed by pressing return.in\n";
    getline(cin, str);
    if (isPal(str)) cout << """ << str + "" is a palindrome ";
    else cout << """ << str + "" is not a palindrome ";
    cout << enddi; return 0;
  }
  ```
- The `isPal` function checks if a string is a palindrome:
  ```cpp
  bool isPal(const string& this_String);
  { uses makeLower, removePunct. if this_String is a palindrome,
    return true; else return false;
  }
  ```

Display 10.10 Palindrome Testing Program (1 of 5)

```cpp
// test for palindrome property
#include <iostream>
#include <string>
#include <cctype>
using namespace std;
void swap(char& lhs, char& rhs);
// swaps char args corresponding to parameters lhs and rhs
string reverse(const string& str);
// returns a copy of arg corresponding to parameter
// str with characters in reverse order.
string removePunct(const string& src, const string& punct);
// returns copy of string src with characters
// in string punct removed
string makeLower(const string& s);
// returns a copy of parameter s that has all upper case
// characters forced to lower case, other characters unchanged.
// Uses <string>, which provides tolower
bool isPal(const string& this_String);
// this_String is a palindrome, return true; else return false;
```
string reverse(const string& str) {
  int start = 0;
  int end = str.length();
  string tmp(str);
  while (start < end) {
    end--;
    swap(tmp[start], tmp[end]);
    start++;
  }
  return tmp;
}

// Returns arg that has all upper case characters forced to lower case,
// other characters unchanged. makeLower uses <string>, which
// provides tolower
string makeLower(const string& s) //uses <cctype>
{
  string temp(s); //This creates a working copy of s
  for (int i = 0; i < s.length(); i++) temp[i] = tolower(s[i]);
  return temp;
}

// Returns a copy of src with characters in punct removed
string removePunct(const string& src, const string& punct) {
  string no_punct;
  int src_len = src.length();
  int punct_len = punct.length();
  for (int i = 0; i < src_len; i++) {
    string aChar = src.substr(i, 1);
    int location = punct.find(aChar, 0);
    if (location < 0 || location >= punct_len)
      no_punct = no_punct + aChar; //aChar not in punct -- keep it
  }
  return no_punct;
}

// uses functions makeLower, removePunct. Returned
// value: if this_Str is a palindrome, return true;
// else return false;
bool isPal(const string& this_Str) {
  string punctuation("\.,;:\?!'" "); //includes a blank
  string str(this_Str);
  str = makeLower(str);
  string lowerStr = removePunct(str, punctuation);
  return lowerStr == reverse(lowerStr);
}

Arrays of string Revisited

- Remember, string is a type that acts exactly like any other type.
- You can have arrays whose base type is string:
  string list[20];
- This is an array of 20 string objects.
- This array can be filled as follows:
  cout << "Enter 20 names, one per line: 
";
  for (int i = 0; i < 20; i++) getline(cin, list[i]);
- Output is the same as for cstrings
  cout << list[i] << endl;
Namespaces Revisited

- Display 10.12 is a version of Display 10.10 where we have handled the namespace issues differently.
- Display 10.10 has only one using directive that applies to the entire file: using namespace std;
- In Display 10.12, we keep the scope of the using directives to a single function, and do not place using directives in swap because none is needed there.
- Names in function headers are qualified with std::, as in std::string.

```cpp
// test for palindrome property
#include <iostream>
#include <string>
#include <cctype>
void swap(char& lhs, char& rhs);
// swaps char args corresponding to parameters lhs and rhs
std::string reverse(const std::string& str);
// returns a copy of arg corresponding to parameter
// str with characters in reverse order.
std::string removePunct(const std::string& src, const std::string& punct);
// returns copy of string src with characters in string punct removed
std::string makeLower (const std::string& s);
// returns a copy of parameter s that has all upper case
// characters forced to lower case, other characters unchanged.
// Uses <string>, which provides tolower
bool isPal(const std::string& this_String);
// uses makeLower, removePunct. If this_String is a palindrome,
// return true; else return false;
```

```cpp
int main()
{ using namespace std;
    string str;
    cout << "Enter a candidate for palindrome test "
        << "in followed by pressing return.\n";
    getline(cin, str);
    if (isPal(str))
        cout << "" << str + "" is a palindrome ";
    else
        cout << "" << str + "" is not a palindrome ";
    cout << endl;
    return 0;
}
```

```cpp
void swap(char& lhs, char& rhs)
{ char tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

```cpp
std::string reverse(const std::string& str)
{ using namespace std;
    int start = 0;
    int end = str.length();
    string tmp(str);
    while (start < end)
    { end--;
        swap(tmp[start], tmp[end]);
        start++;
    }
    return tmp;
}
```

```cpp
std::string makeLower(const std::string& s) //uses<cctype>
{
    using namespace std;
    string temp(s); //This creates a working copy of s
    for (int i = 0; i < s.length(); i++)
    { string aChar = s.substr(i,1);
        int location = punct.find(aChar, 0);
        // find location of successive characters of src in punct
        if (location < 0 || location >= punct.length())
            temp[i] = tolower(s[i]);
    }
    return temp;
}
```

```cpp
std::string removePunct(const std::string& src, const std::string& punct)
{ using namespace std;
    string no_punct;
    int src_len = src.length();
    int punct_len = punct.length();
    for(int i = 0; i < src_len; i++)
    { string aChar = src.substr(i,1);
        int location = punct.find(aChar, 0);
        // find location of successive characters of src in punct
        if (location < 0 || location >= punct_len)
            no_punct = no_punct + aChar; //aChar not in punct -- keep it
    }
    return no_punct;
}
```
bool isPal(const std::string& this_String)
{
    using namespace std;
    string punctuation(",:.;?!'" "); // includes a blank
    string str(this_String);
    str = makeLower(str);
    string lowerStr = removePunct(str, punctuation);
    return lowerStr == reverse(lowerStr);
}