Functions and Recursion in C++ (with. Array)

Contents

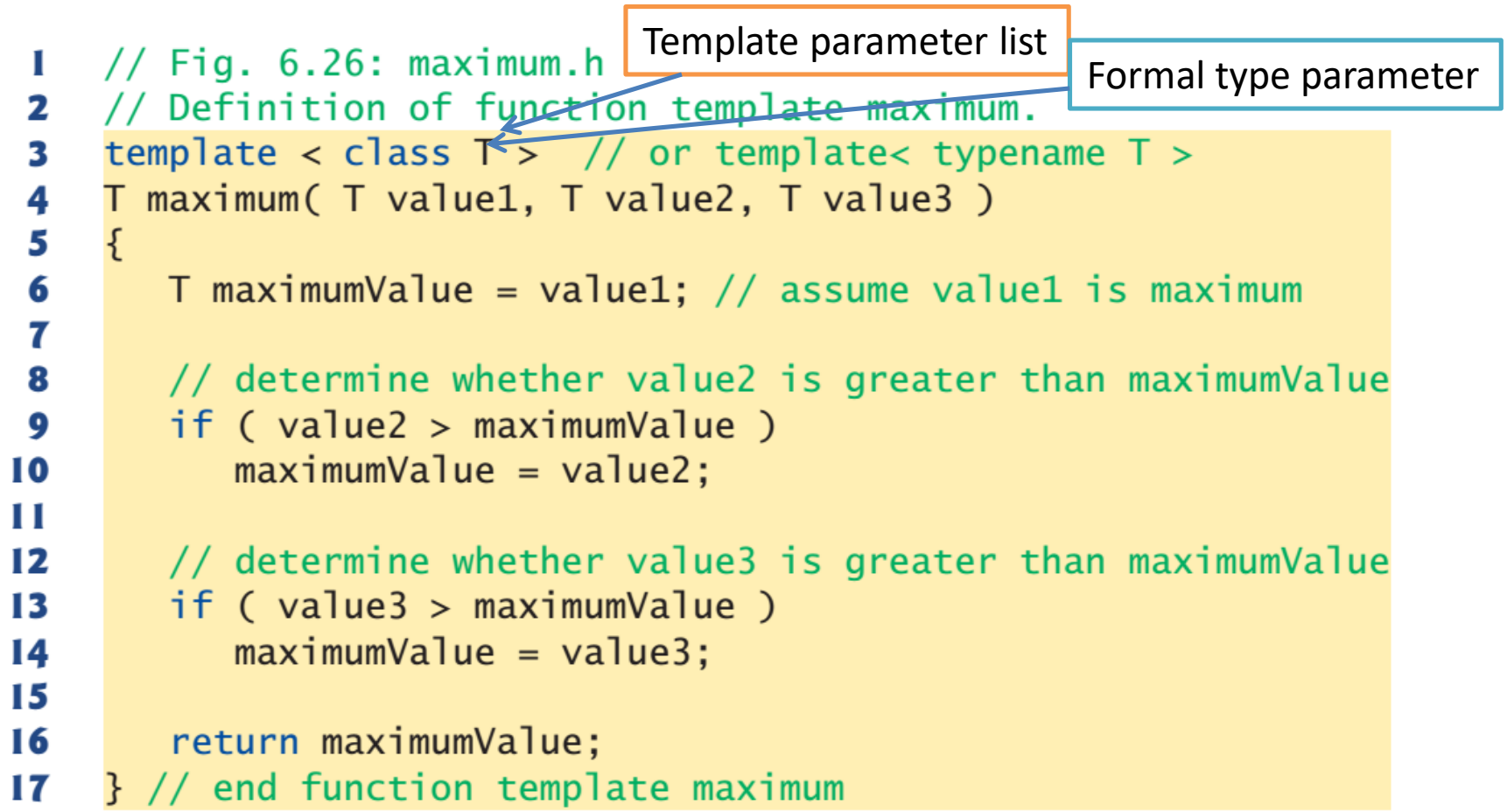
- Function Template
- Recursion: Fibonacci Series
- Functions and Recursion: What we learned

- Case Study: Case Study: Class GradeBook Using a Two Dimensional Array

Function Template

- Overloaded functions are normally used to perform similar operations that involve different program logic on different data types.
- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.
- Write a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately.
- Defining a single function template essentially defines a whole family of overloaded functions.

Function Template



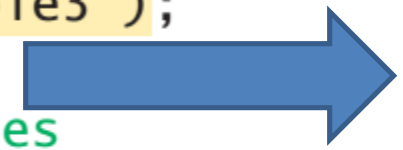

```
1  // Fig. 6.26: maximum.h
2  // Definition of function template maximum.
3  template < class T >  // or template< typename T >
4  T maximum( T value1, T value2, T value3 )
5  {
6     T maximumValue = value1; // assume value1 is maximum
7
8     // determine whether value2 is greater than maximumValue
9     if ( value2 > maximumValue )
10       maximumValue = value2;
11
12    // determine whether value3 is greater than maximumValue
13    if ( value3 > maximumValue )
14       maximumValue = value3;
15
16    return maximumValue;
17 } // end function template maximum
```

Function Template

```
1  // Fig. 6.27: fig06_27.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  #include "maximum.h" // include definition of function template maximum
5  using namespace std;
6
7  int main()
8  {
9     // demonstrate maximum with int values
10    int int1, int2, int3;
11
12    cout << "Input three integer values: ";
13    cin >> int1 >> int2 >> int3;
14
15    // invoke int version of maximum
16    cout << "The maximum integer value is: "
17       << maximum( int1, int2, int3 );
18
19    // demonstrate maximum with double values
20    double double1, double2, double3;
21
22    cout << "\n\nInput three double values: ";
23    cin >> double1 >> double2 >> double3;
```

Function Template

```
25       // invoke double version of maximum
26       cout << "The maximum double value is: "
27          << maximum( double1, double2, double3 );
28
29       // demonstrate maximum with char values
30       char char1, char2, char3;
31
32       cout << "\n\nInput three characters: ";
33       cin >> char1 >> char2 >> char3;
34
35       // invoke char version of maximum
36       cout << "The maximum character value is: "
37          << maximum( char1, char2, char3 ) << endl;
38    } // end main
```



Function Template

```
int maximum( int value1, int value2, int value3 )
{
   int maximumValue = value1; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if ( value2 > maximumValue )
      maximumValue = value2;

   // determine whether value3 is greater than maximumValue
   if ( value3 > maximumValue )
      maximumValue = value3;

   return maximumValue;
} // end function template maximum
```

Recursion: Fibonacci Series

The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, …

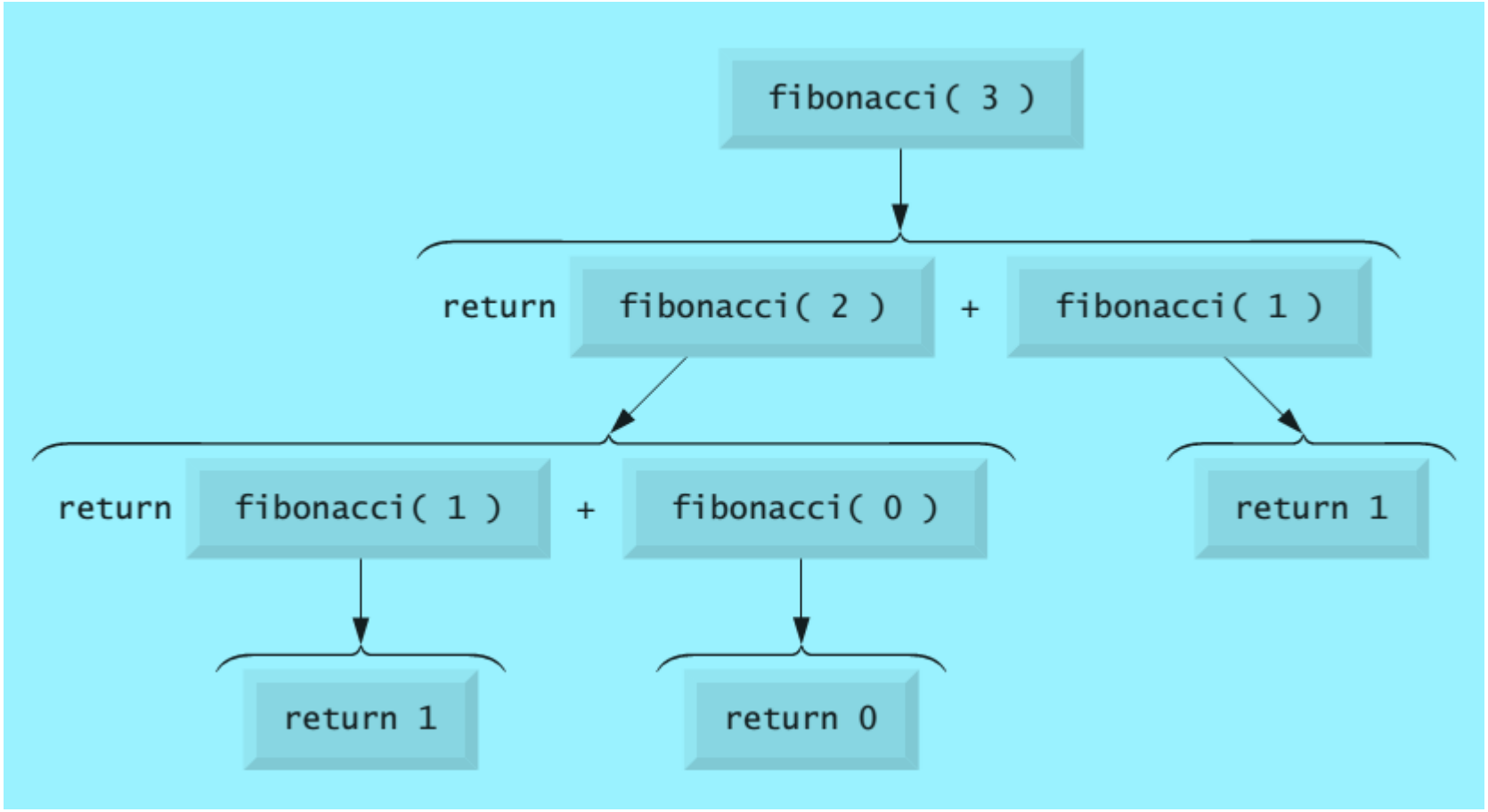
begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

Recursion: Fibonacci Series

```
1  // Fig. 6.30: fig06_30.cpp
2  // Testing the recursive fibonacci function.
3  #include <iostream>
4  using namespace std;
5
6  unsigned long fibonacci( unsigned long ); // function prototype
7
8  int main()
9  {
10    // calculate the fibonacci values of 0 through 10
11    for ( int counter = 0; counter <= 10; counter++ )
12       cout << "fibonacci( " << counter << " ) = "
13          << fibonacci( counter ) << endl;
14
15    // display higher fibonacci values
16    cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
17    cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
18    cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
19 } // end main
20
21 // recursive function fibonacci
22 unsigned long fibonacci( unsigned long number )
23 {
24    if ( ( number == 0 ) || ( number == 1 ) ) // base cases
25       return number;
26    else // recursion step
27       return fibonacci( number - 1 ) + fibonacci( number - 2 );
28 } // end function fibonacci
```

Recursion: Fibonacci Series



the order in which operators are applied to their operands, namely, the order dictated by the rules of operator precedence and associativity.

Recursion vs. Iteration

```
1  // Fig. 6.32: fig06_32.cpp
2  // Testing the iterative factorial function.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned long ); // function prototype
8
9  int main()
10 {
11    // calculate the factorials of 0 through 10
12    for ( int counter = 0; counter <= 10; counter++ )
13       cout << setw( 2 ) << counter << "! = " << factorial( counter )
14          << endl;
15 } // end main
16
17 // iterative function factorial
18 unsigned long factorial( unsigned long number )
19 {
20    unsigned long result = 1;
21
22    // iterative factorial calculation
23    for ( unsigned long i = number; i >= 1; i-- )
24       result *= i;
25
26    return result;
27 } // end function factorial
```

Recursion vs. Iteration

- Iteration uses a repetition structure; recursion uses a selection structure.
- Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls.

- Recursion has many negatives

- It repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space.

- Each recursive call causes another copy of the function (actually only the function's variables) to be created; this can consume considerable memory.

- Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.

Functions and Recursion: What We Learned

Function Prototypes and Argument Coercion

- ✓ The portion of a function prototype that includes the name of the function and the types of its arguments is called the function signature or simply the signature.
- ✓ An important feature of function prototypes is argument coercion—i.e., forcing arguments to the appropriate types specified by the parameter declarations.
- ✓ Arguments can be promoted by the compiler to the parameter types as specified by C++'s promotion rules. The promotion rules indicate how to convert between types without losing data.

Functions and Recursion: What We Learned

Storage Classes

- ✓ An identifier's storage class determines the period during which that identifier exists in memory. An identifier's scope is where the identifier can be referenced in a program.

- ✓ Keywords auto and register declare variables of the automatic storage class. Such variables are created when program execution enters the block in which they're defined, they exist while the block is active and they're destroyed when the program exits the block.

- ✓ Keywords extern and static declare identifiers for variables of the static storage class and for functions. Static-storage-class variables exist from the point at which the program begins execution and last for the duration of the program.

- ✓ A static-storage-class variable's storage is allocated when the program begins execution. Such a variable is initialized once when its declaration is encountered. For functions, the name of the function exists when the program begins execution as for all other functions.

Functions and Recursion: What We Learned

Scope Rules

- ✓ Unlike automatic variables, static local variables retain their values when the function in which they're declared returns to its caller.
- ✓ An identifier declared outside any function or class has global namespace scope.
- ✓ Labels are the only identifiers with function scope. Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body.
- ✓ Identifiers declared inside a block have local scope. Local scope begins at the identifier's declaration and ends at the terminating right brace (}) of the block in which the identifier is declared.
- ✓ Identifiers in the parameter list of a function prototype have function-prototype scope

Functions and Recursion: What We Learned

Inline Functions

- ✓ C++ provides inline functions to help reduce function call overhead—especially for small functions.

- ✓ Placing the qualifier ***inline*** before a function's return type in the function definition "advises" the compiler to generate a copy of the function's code in place to avoid a function call.

Unary Scope Resolution Operator

- ✓ C++ provides the unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope.

Functions and Recursion: What We Learned

Function Overloading

- ✓ C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters. This capability is called function overloading.

- ✓ When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call.

- ✓ Overloaded functions are distinguished by their signatures

Functions and Recursion: What We Learned

Function Template

- ✓ If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently using function templates.
- ✓ Given the argument types provided in calls to a function template, C++ automatically generates separate function template specializations to handle each type of call appropriately.

- ✓ All function template definitions begin with the template keyword followed by a template parameter list to the function template enclosed in angle brackets.

- ✓ The formal type parameters are placeholders for fundamental types or user-defined types. These placeholders are used to specify the types of the function's parameters, to specify the function's return type and to declare variables within the body of the function definition.

Functions and Recursion: What We Learned

Recursion vs. Iteration

- ✓ Iteration and recursion have many similarities: both are based on a control statement, involve repetition, involve a termination test, gradually approach termination and can occur infinitely.

- ✓ Recursion repeatedly invokes the mechanism, and consequently the overhead, of function calls.

- ✓ This can be expensive in both processor time and memory space. Each recursive call causes another copy of the function's variables to be created; this can consume considerable memory.

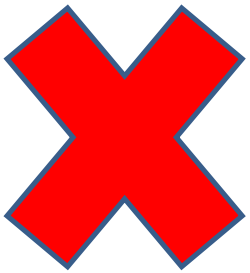
Case Study: Class GradeBook Using an Array to Store Grades

What We need to do:

- ✓ This class represents a grade book used by a professor to store and analyze student grades.
- ✓ Previous versions of the class process grades entered by the user, but do not maintain the individual grade values in the class's data members. Thus, repeat calculations require the user to reenter the grades.
- ✓ One way to solve this problem would be to store each grade entered in an individual data member of the class.
- ✓ For example, we could create data members grade1, grade2, …, grade10 in class GradeBook to store 10 student grades.
- ✓ However, the code to total the grades and determine the class average would be cumbersome.
- ✓ In this section, we solve this problem by storing grades in an array.

O/P

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87

Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```



```
// Fig. 7.15: GradeBook.h
// Definition of class GradeBook that uses an array to store test grades
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ Standard Library string class
using namespace std;

// GradeBook class definition
class GradeBook
{
public:
   // constant -- number of students who took the test
   static const int students = 10; // note public data

   // constructor initializes course name and array of grades
   GradeBook( string, const int [] );

   void setCourseName( string ); // function to set the course name
   string getCourseName(); // function to retrieve the course name
   void displayMessage(); // display a welcome message
   void processGrades(); // perform various operations on the grade data
   int getMinimum(); // find the minimum grade for the test
   int getMaximum(); // find the maximum grade for the test
   double getAverage(); // determine the average grade for the test
   void outputBarChart(); // output bar chart of grade distribution
   void outputGrades(); // output the contents of the grades array
private:
   string courseName; // course name for this grade book
   int grades[ students ]; // array of student grades
}; // end class GradeBook
```

Case Study: Class GradeBook Using an Array to Store Grades

- ✓ Keyword static: the data member is shared by all objects of the class—all GradeBook objects store grades for the same number of students
- ✓ What we knew: each object (instance) of the class has a separate copy of the variable in memory
- ✓ There are variables for which each object of a class does not have a separate copy. That is the case with static data members, which are also known as class variables.
- ✓ When objects of a class containing static data members are created, all the objects share one copy of the class's static data members.
- ✓ A static data member can be accessed within the class definition and the member-function definitions like any other data member.
- ✓ A public static data member can also be accessed outside of the class, even when no objects of the class exist, using the class name followed by the binary scope resolution operator (::) and the name of the data member.

Case Study: Class GradeBook Using an Array to Store Grades

```
// Fig. 7.16: GradeBook.cpp
// Member-function definitions for class GradeBook that
// uses an array to store test grades.
#include <iostream>
#include <iomanip>
#include "GradeBook.h" // GradeBook class definition
using namespace std;

// constructor initializes courseName and grades array
GradeBook::GradeBook( string name, const int gradesArray[] )
{
   setCourseName( name ); // initialize courseName

   // copy grades from gradesArray to grades data member
   for ( int grade = 0; grade < students; grade++ )
      grades[ grade ] = gradesArray[ grade ];
} // end GradeBook constructor
```

Case Study: Class GradeBook Using an Array to Store Grades

```
// function to set the course name
void GradeBook::setCourseName( string name )
{
   courseName = name; // store the course name
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName()
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
   // this statement calls getCourseName to get the
   // name of the course this GradeBook represents
   cout << "Welcome to the grade book for\n" << getCourseName() << "!"
      << endl;
} // end function displayMessage
```

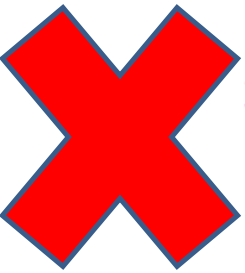
Case Study: Class GradeBook Using an Array to Store Grades

```
// perform various operations on the data
void GradeBook::processGrades()
{
   outputGrades(); // output grades array

   // display average of all grades and minimum and maximum grades
   cout << "\nClass average is " << setprecision( 2 ) << fixed <<
      getAverage() << "\nLowest grade is " << getMinimum() <<
      "\nHighest grade is " << getMaximum() << endl;

   outputBarChart(); // print grade distribution chart
} // end function processGrades

// find minimum grade
int GradeBook::getMinimum()
{
   int lowGrade = 100; // assume lowest grade is 100

   // loop through grades array
   for ( int grade = 0; grade < students; grade++ )
   {
      // if current grade lower than lowGrade, assign it to lowGrade
      if ( grades[ grade ] < lowGrade )
         lowGrade = grades[ grade ]; // new lowest grade
   } // end for

   return lowGrade; // return lowest grade
} // end function getMinimum
```

Case Study: Class GradeBook Using an Array to Store Grades

```
// find maximum grade
int GradeBook::getMaximum()
{
    int highGrade = 0; // assume highest grade is 0

    // loop through grades array
    for ( int grade = 0; grade < students; grade++ )
    {
       // if current grade higher than highGrade, assign it to highGrade
       if ( grades[ grade ] > highGrade )
          highGrade = grades[ grade ]; // new highest grade
    } // end for

    return highGrade; // return highest grade
 } // end function getMaximum

 // determine average grade for test
 double GradeBook::getAverage()
 {
    int total = 0; // initialize total

    // sum grades in array
    for ( int grade = 0; grade < students; grade++ )
       total += grades[ grade ];

    // return average of grades
    return static_cast< double >( total ) / students;
 } // end function getAverage
```

Case Study: Class GradeBook Using an Array to Store Grades



```
// output bar chart displaying grade distribution
void GradeBook::outputBarChart()
{
   cout << "\nGrade distribution:" << endl;

   // stores frequency of grades in each range of 10 grades
   const int frequencySize = 11;
   int frequency[ frequencySize ] = {}; // initialize elements to 0

   // for each grade, increment the appropriate frequency
   for ( int grade = 0; grade < students; grade++ )
      frequency[ grades[ grade ] / students ]++;

   // for each grade frequency, print bar in chart
   for ( int count = 0; count < frequencySize; count++ )
   {
      // output bar labels ("0-9:", ..., "90-99:", "100:" )
      if ( count == 0 )
         cout << "  0-9: ";
      else if ( count == 10 )
         cout << "  100: ";
      else
         cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";

      // print bar of asterisks
      for ( int stars = 0; stars < frequency[ count ]; stars++ )
         cout << '*';
```

Case Study: Class GradeBook Using an Array to Store Grades

```
      cout << endl; // start a new line of output
   } // end outer for
} // end function outputBarChart

// output the contents of the grades array
void GradeBook::outputGrades()
{
   cout << "\nThe grades are:\n\n";

   // output each student's grade
   for ( int student = 0; student < students; student++ )
      cout << "Student " << setw( 2 ) << student + 1 << ": " << setw( 3 )
         << grades[ student ] << endl;
} // end function outputGrades
```

Case Study: Class GradeBook Using an Array to Store Grades

```
1  // Fig. 7.17: fig07_17.cpp
2  // Creates GradeBook object using an array of grades.
3  #include "GradeBook.h" // GradeBook class definition
4
5  // function main begins program execution
6  int main()
7  {
8     // array of student grades
9     int gradesArray[ GradeBook::students ] =
10       { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11
12    GradeBook myGradeBook(
13       "CS101 Introduction to C++ Programming", gradesArray );
14    myGradeBook.displayMessage();
15    myGradeBook.processGrades();
16 } // end main
```

Case Study: Case Study: Class GradeBook Using a Two Dimensional Array

What We need to do:

- ✓ Our program will contain a version of class GradeBook that uses a two-dimensional array grades to store the grades of a number of students on multiple exams.

- ✓ Each row of the array represents a single student's grades for the entire course, and each column represents all the grades the students earned for one particular exam.

- ✓ A client program, will pass the array as an argument to the GradeBook constructor. In this example, we use a ten-by-three array containing ten students' grades on three exams

Do It Yourself

VERY IMPORTANT

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```