Review of Important Topics in CS1600

- Functions
- Arrays
- C-strings

Array Basics

Arrays

- An array is used to process a collection of data of the same type
  - Examples: A list of names
    A list of temperatures
- Why do we need arrays?
  - Imagine keeping track of 5 test scores, or 100, or 1000 in memory
    - How would you name all the variables?
    - How would you process each of the variables?

Declaring an Array

- An array, named `score`, containing five variables of type int can be declared as

```
int score[5];
```

- This is like declaring 5 variables of type `int`:

```
score[0], score[1], … , score[4]
```

- The value in brackets is called
  - A subscript
  - An index

The Array Variables

- The variables making up the array are referred to as
  - Indexed variables
  - Subscripted variables
  - Elements of the array
- The number of indexed variables in an array is **the declared size**, or **size**, of the array
  - The largest index is one less than the size
  - The first index value is zero
  - Not all variables are actually being used all the time!

Array Variable Types

- An array can have indexed variables of any type
- All indexed variables in an array are of the same type
  - This is the **base type** of the array
- An **indexed variable** can be used anywhere an ordinary variable of the base type is used

Using [ ] With Arrays

- In an array **declaration**, **[ ]**'s enclose the size of the array such as this array of 5 integers:

  `int score [5];`

- When referring to one of the indexed variables, the **[ ]**'s enclose a number identifying one of the indexed variables
  - E.g.,

    `score[3]=7;`

    `score[3]` is one of the indexed variables
  - The value in the [ ]'s can be any expression that evaluates to one of the integers 0 to (size -1)

Indexed Variable Assignment

- To assign a value to an indexed variable, use the assignment operator:

```
int n = 2;
score[n + 1] = 99;
```

  - In this example, variable score[3] is assigned 99

Loops And Arrays

- for-loops are commonly used to step through arrays

First index is 0

Last index is (size – 1)

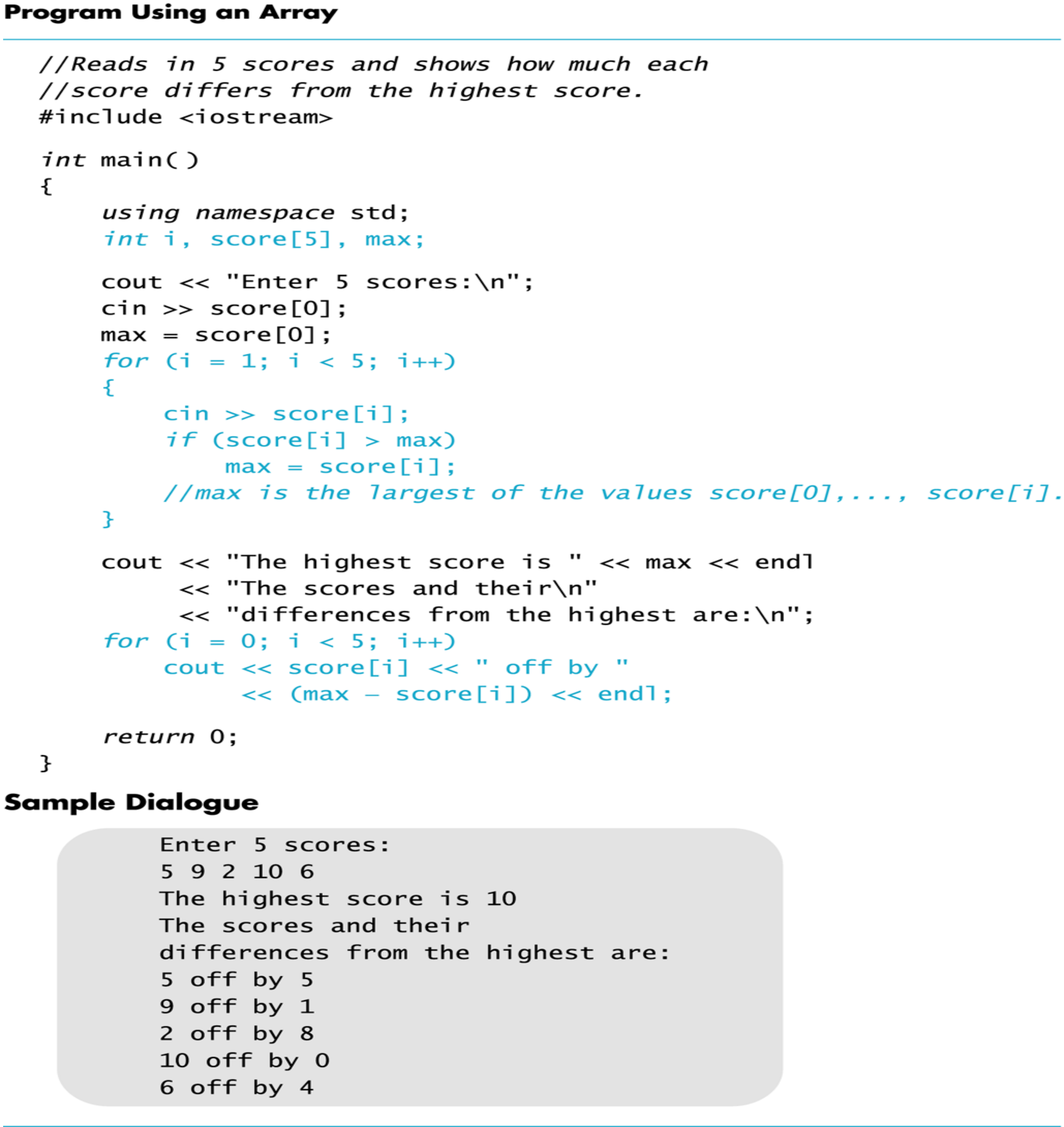
- Example:

```
for (int i = 0; i < 5; i++)
{
 cout << score[i] << " off by “
     << (max – score[i]) << endl;
}
```

could display the difference between each score and the maximum score stored in an array

Display 7.1

Display 7.1



Program Using an Array

```
//Reads in 5 scores and shows how much each
//score differs from the highest score.
#include <iostream>

int main( )
{
    using namespace std;
    int i, score[5], max;

    cout << "Enter 5 scores:\n";
    cin >> score[0];
    max = score[0];
    for (i = 1; i < 5; i++)
    {
        cin >> score[i];
        if (score[i] > max)
            max = score[i];
        //max is the largest of the values score[0],..., score[i].
    }

    cout << "The highest score is " << max << endl
         << "The scores and their\n"
         << "differences from the highest are:\n";
    for (i = 0; i < 5; i++)
        cout << score[i] << " off by "
             << (max - score[i]) << endl;

    return 0;
}
```

Sample Dialogue

```
Enter 5 scores:
5 9 2 10 6
The highest score is 10
The scores and their
differences from the highest are:
5 off by 5
9 off by 1
2 off by 8
10 off by 0
6 off by 4
```

Constants and Arrays

- Use constants to declare the size of an array
  - Using a constant allows your code to be easily altered for use on a smaller or larger set of data
    - Example:

```
const int  NUMBER_OF_STUDENTS = 50;
int score[NUMBER_OF_STUDENTS];
for ( i = 0; i < NUMBER_OF_STUDENTS;  i++)
          cout << score[i] << " off by “ << (max – score[i]) << endl;
```

    - Only the value of the constant must be changed to make this code work for any number of students

Variables and Declarations

- Most compilers do not allow the use of a variable to declare the size of an array

  Example:

  ```
  cout << "Enter number of students: ";
  cin >> number;
  int score[number];
  ```

  - This code is illegal on many compilers

Array Declaration Syntax

- To declare an array, use the syntax:

  ```
  Type_Name     Array_Name[Declared_Size];
  ```

  - Type_Name can be any type
  - Declared_Size can be a constant to make your program more versatile
- Once declared, the array consists of the indexed variables: Array_Name[0] to Array_Name[Declared_Size -1]

Arrays and Memory

- Declaring the array

```
int a[6];
```

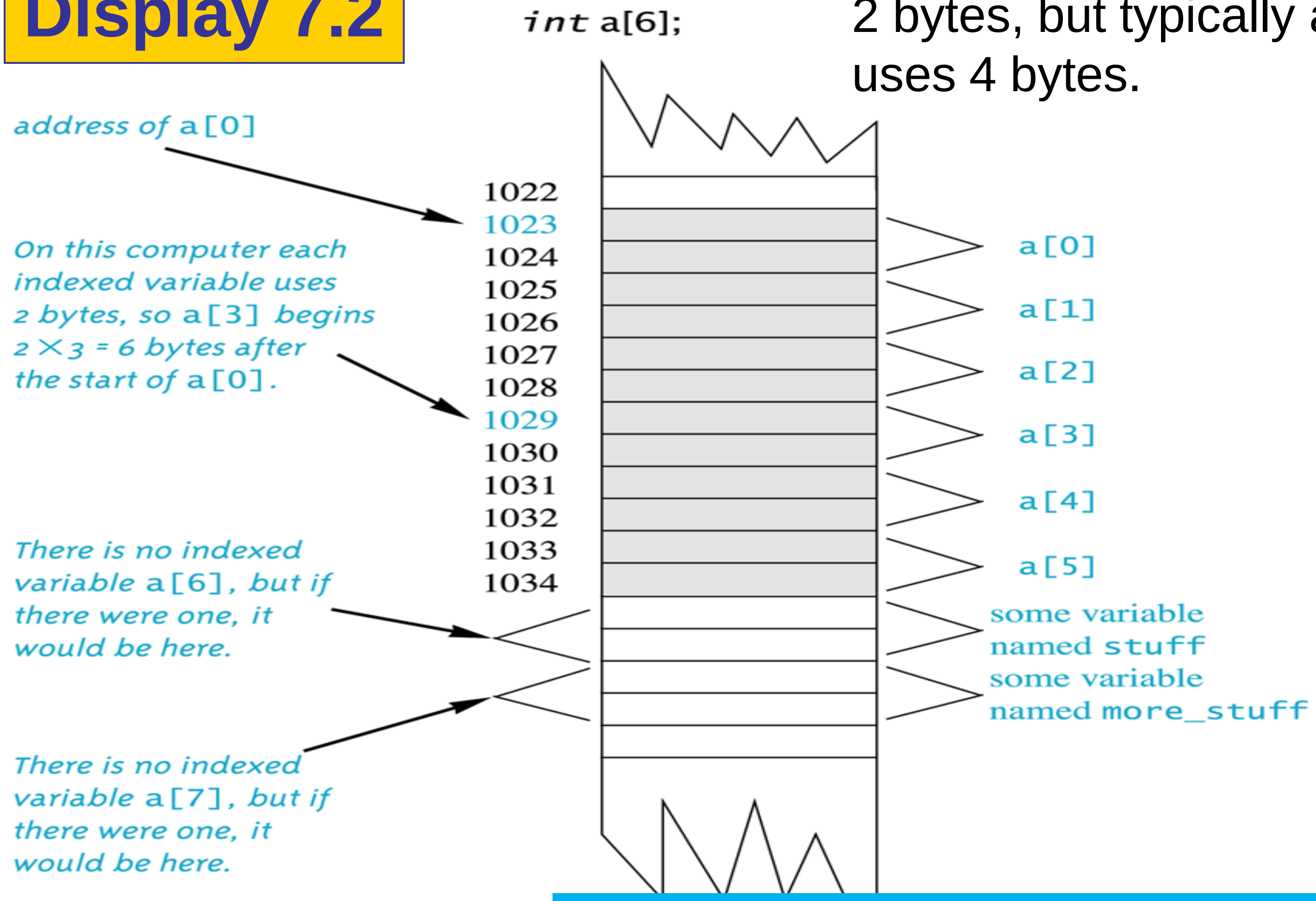
  - Reserves memory for six variables of type int
  - The variables are stored one after another
  - The address of a[0] is remembered by C++
    - The addresses of the other indexed variables is not remembered by C++
  - To determine the address of a[3]
    - C++ starts at a[0]
    - C++ counts past enough memory for three integers to find a[3]

Display 7.2

An Array in Memory

Display 7.2

in this example, each int variable uses 2 bytes, but typically an int variable uses 4 bytes.


int a[6];
address of a[0]
On this computer each indexed variable uses 2 bytes, so a[3] begins 2 × 3 = 6 bytes after the start of a[0].
There is no indexed variable a[6], but if there were one, it would be here.
There is no indexed variable a[7], but if there were one, it would be here.
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
some variable named stuff
some variable named more_stuff

Recall:

- ❑ Computer memory consists of numbered locations called bytes
  - ▪ A byte's number is its address
- ❑ A simple variable is stored in consecutive bytes
  - ▪ The number of bytes depends on the variable's type
- ❑ A variable's address is the address of its first byte

Array Index Out of Range

- A common error is using a nonexistent index
  - Index values for int a[6] are the values 0 through 5
  - An index value not allowed by the array declaration is out of range
  - Using an out of range index value doe not produce an error message!

Out of Range Problems

- If an array is declared as: int a[6];
  and an integer is declared as: int i = 7;
- Executing the statement a[i] = 238; causes…
  - The computer to calculate the address of the illegal a[7]
    (This address could be where some other variable is stored)
  - The value 238 is stored at the address calculated for a[7]
  - No warning is given!

Initializing Arrays

- To initialize an array when it is declared
    - The values for the indexed variables are enclosed in braces and separated by commas
- Example: `int children[3] = { 2, 12, 1 };`

  is equivalent to:

```
int children[3];
children[0] = 2;
children[1] = 12;
children[2] = 1;
```

Default Values

- If too few values are listed in an initialization statement
  - The listed values are used to initialize the first of the indexed variables
  - The remaining indexed variables are initialized to a zero of the base type
  - Example: `int a[10] = {5, 5};`
    initializes a[0] and a[1] to 5 and a[2] through a[9] to 0

Un-initialized Arrays

- If no values are listed in the array declaration, some compilers will initialize each variable to a zero of the base type
    - DO NOT DEPEND ON THIS!

Arrays in Functions

Arrays in Functions

- Indexed variables can be arguments to functions
  - Example: If a program contains these declarations:

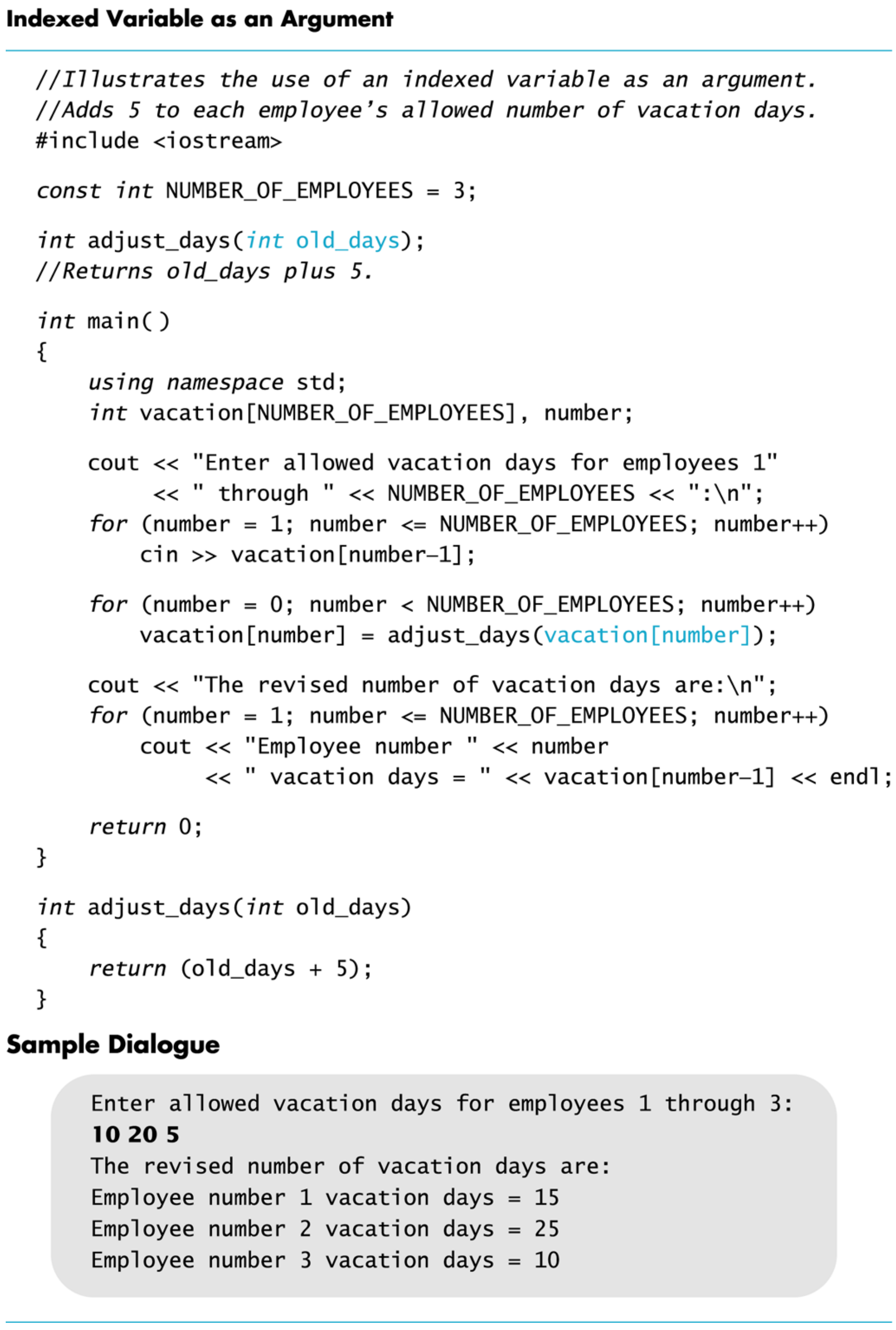
```
int i, n, a[10];
void my_function(int n);
```

  - Variables a[0] through a[9] are of type int, making these calls legal:

```
my_function( a[ 0 ] );
my_function( a[ 3 ] );
my_function( a[ i ] );
```

Display 7.3

Display 7.3



Indexed Variable as an Argument

```
//Illustrates the use of an indexed variable as an argument.
//Adds 5 to each employee's allowed number of vacation days.
#include <iostream>

const int NUMBER_OF_EMPLOYEES = 3;

int adjust_days(int old_days);
//Returns old_days plus 5.

int main( )
{
    using namespace std;
    int vacation[NUMBER_OF_EMPLOYEES], number;

    cout << "Enter allowed vacation days for employees 1"
         << " through " << NUMBER_OF_EMPLOYEES << ":\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cin >> vacation[number-1];

    for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
        vacation[number] = adjust_days(vacation[number]);

    cout << "The revised number of vacation days are:\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cout << "Employee number " << number
             << " vacation days = " << vacation[number-1] << endl;

    return 0;
}

int adjust_days(int old_days)
{
    return (old_days + 5);
}
```

Sample Dialogue

```
Enter allowed vacation days for employees 1 through 3:
10 20 5
The revised number of vacation days are:
Employee number 1 vacation days = 15
Employee number 2 vacation days = 25
Employee number 3 vacation days = 10
```

Arrays as Function Arguments

- A formal parameter can be for an entire array
  - Such a parameter is called **an array parameter**
    - It is not a call-by-value parameter
    - It is not a call-by-reference parameter
    - Array parameters behave much like call-by-reference parameters

Array Parameter Declaration

- An array parameter is indicated using empty brackets in the parameter list such as

```
void fill_up(int a[ ], int size);
```

Function Calls With Arrays

- If function fill_up is **declared** in this way:

  ```
  void fill_up( int a[ ] , int size);
  ```

- and array score is declared this way:

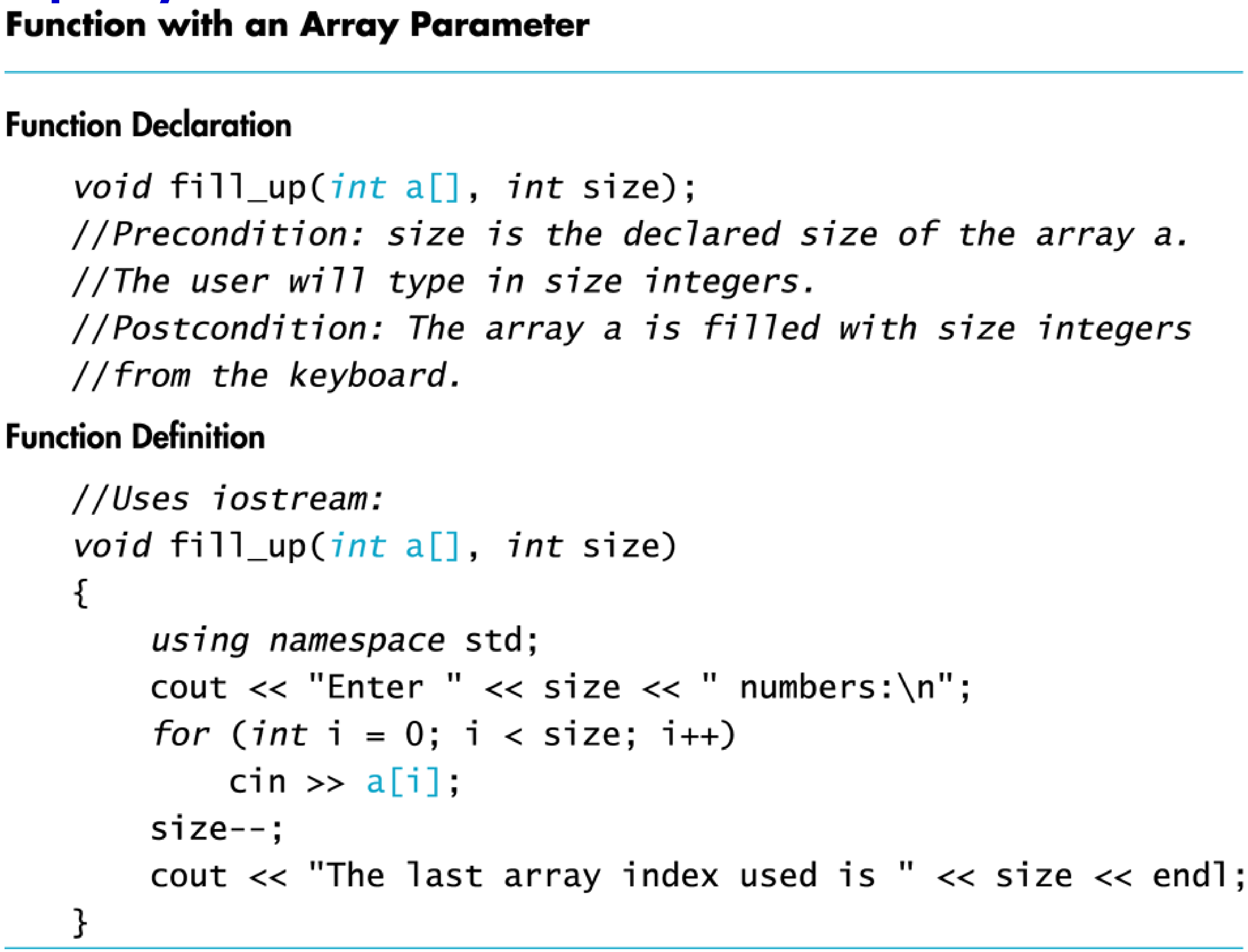
  ```
  int score[5], number_of_scores;
  ```

- fill_up is **called** in this way:

  ```
  fill_up(score, number_of_scores);
  ```

Display 7.4

Display 7.4



Function with an Array Parameter

Function Declaration

```
void fill_up(int a[], int size);
//Precondition: size is the declared size of the array a.
//The user will type in size integers.
//Postcondition: The array a is filled with size integers
//from the keyboard.
```

Function Definition

```
//Uses iostream:
void fill_up(int a[], int size)
{
    using namespace std;
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    size--;
    cout << "The last array index used is " << size << endl;
}
```

Function Call Details

- A formal parameter is identified as an array parameter by the [ ]'s with no index expression

```
void fill_up(int a[ ], int size);
```

- An array argument does not use the [ ]'s

```
fill_up(score, number_of_scores);
```

Array Formal Parameters

- An array formal parameter is a placeholder for the argument
  - When an array is an argument in a function call, an action performed on the array parameter is performed on the array argument
  - The values of the indexed variables (i.e., the array argument) can be changed by the function

Array Argument Details

- What does the computer know about an **array** once it is declared?
  - The base type
  - The address of the first indexed variable
  - The number of indexed variables
- What does a function know about an **array argument** during a function call?
  - The base type
  - The address of the first indexed variable

Array Parameter Considerations

- Because a function does not know the size of an array argument…
  - The programmer should include a formal parameter that specifies the size of the array
  - The function can process arrays of various sizes
    - Function fill_up from Display 7.4 can be used to fill an array of any size:

```
int score[5];
int time[10];
fill_up(score, 5);
fill_up(time, 10);
```

const Modifier

- Recall: array parameters allow a function to change the values stored in the array argument
- If a function should not change the values of the array argument, use the modifier **const**
- An array parameter modified with **const** is a constant array parameter
  - Example:

    ```
    void display_array(const int a[ ], int size);
    ```

Using const With Arrays

- If const is used to modify an array parameter:
  - const is used in both the function declaration and definition to modify the array parameter
  - The compiler will issue an error if you write code that changes the values stored in the array parameter

Function calls and const

- If a function with a constant array parameter calls another function using the constant array parameter as an argument…
  - The called function must use a constant array parameter as a placeholder for the array
  - The compiler will issue an error if a function is called that does not have a const array parameter to accept the array argument

const Parameters Example

```
double compute_average(int a[ ], int size);

 void show_difference(const int a[ ], int size)
 {
        double average = compute_average(a, size);
         …
 }
```

- compute_average has no constant array parameter
- This code generates an error message because compute_average could change the array parameter

Returning An Array

- Recall that functions can return (via return-statement) a value of type int, double, char, …

- **Functions cannot return arrays**

- We learn later how to return a pointer to an array

Programming with Arrays

Programming With Arrays

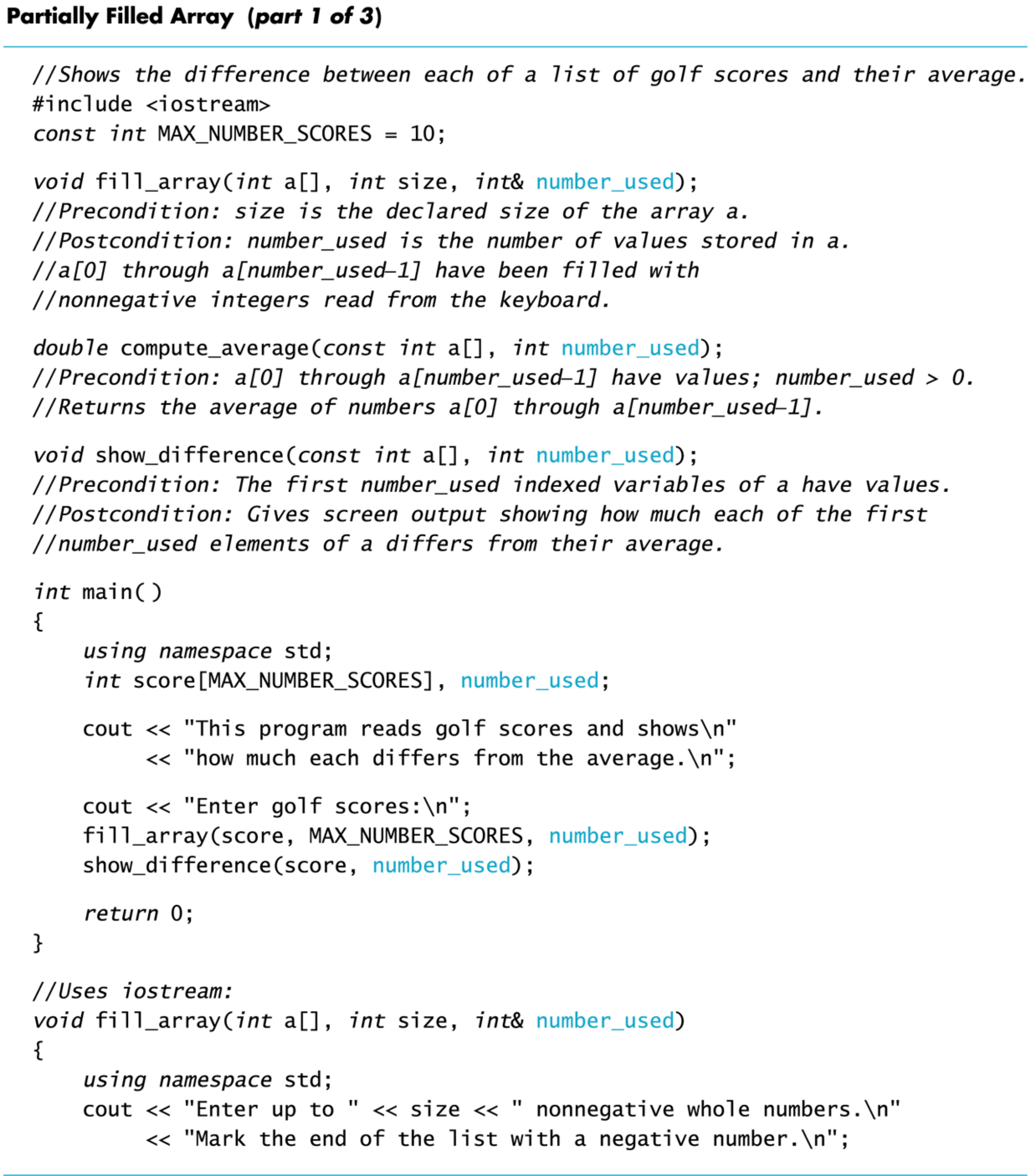
- The size needed for an array is changeable
  - Often varies from one run of a program to another
  - Is often not known when the program is written

- A common solution to the size problem
  - Declare the array size to be the largest that could be needed
  - Decide how to deal with partially filled arrays

Partially Filled Arrays

- When using arrays that are partially filled
  - A parameter, **number_used**, may be sufficient to ensure that referenced index values are legal
  - Functions dealing with the array may not need to know the declared size of the array, only how many elements are stored in the array
  - A function such as **fill_array** in Display 7.9 needs to know the declared size of the array

Display 7.9 (1) Display 7.9 (2) Display 7.9 (3)

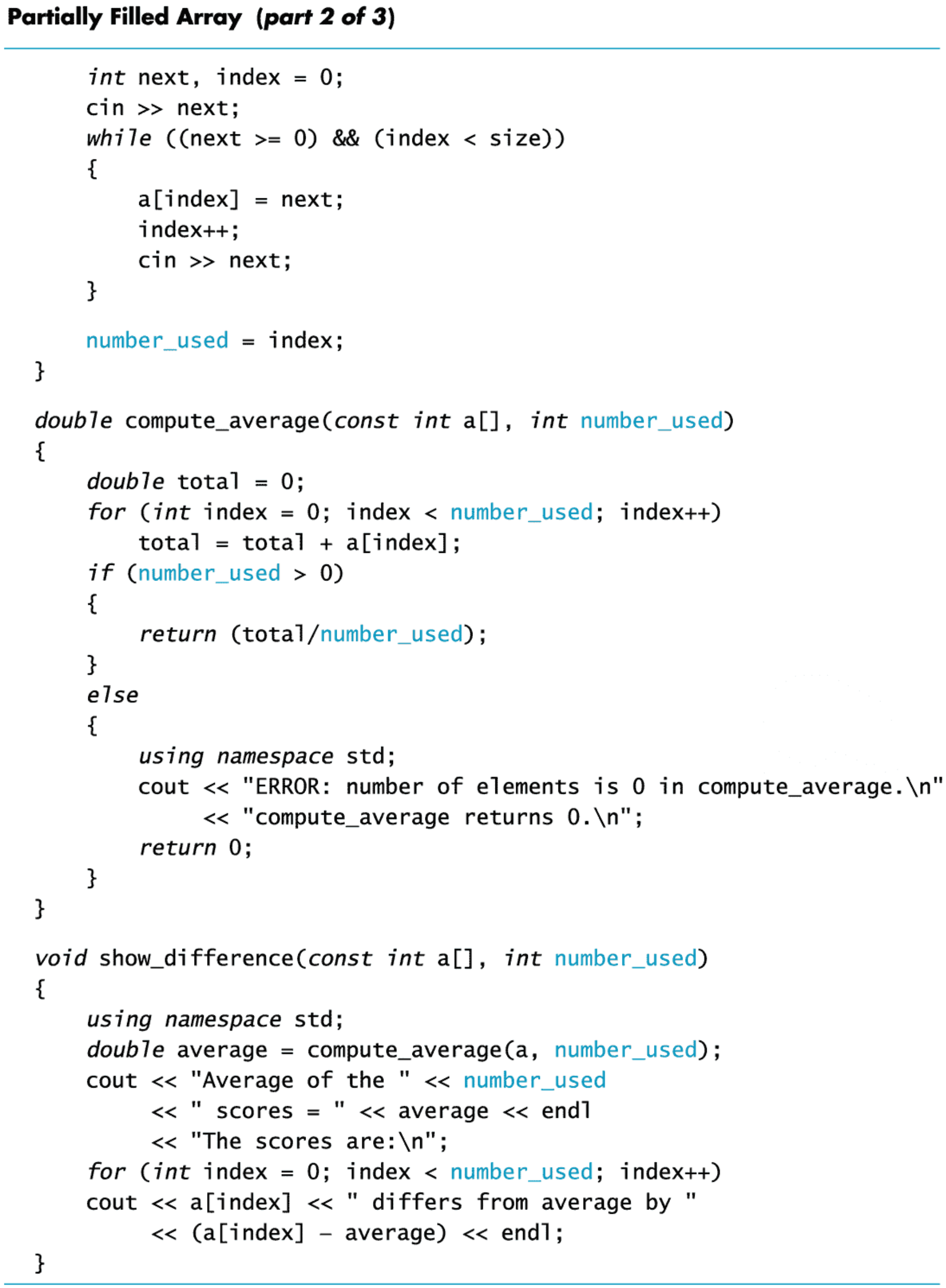


Partially Filled Array ***(part 1 of 3)***

```
//Shows the difference between each of a list of golf scores and their average.
#include <iostream>
const int MAX_NUMBER_SCORES = 10;

void fill_array(int a[], int size, int& number_used);
//Precondition: size is the declared size of the array a.
//Postcondition: number_used is the number of values stored in a.
//a[0] through a[number_used-1] have been filled with
//nonnegative integers read from the keyboard.

double compute_average(const int a[], int number_used);
//Precondition: a[0] through a[number_used-1] have values; number_used > 0.
//Returns the average of numbers a[0] through a[number_used-1].

void show_difference(const int a[], int number_used);
//Precondition: The first number_used indexed variables of a have values.
//Postcondition: Gives screen output showing how much each of the first
//number_used elements of a differs from their average.

int main( )
{
    using namespace std;
    int score[MAX_NUMBER_SCORES], number_used;

    cout << "This program reads golf scores and shows\n"
         << "how much each differs from the average.\n";

    cout << "Enter golf scores:\n";
    fill_array(score, MAX_NUMBER_SCORES, number_used);
    show_difference(score, number_used);

    return 0;
}

//Uses iostream:
void fill_array(int a[], int size, int& number_used)
{
    using namespace std;
    cout << "Enter up to " << size << " nonnegative whole numbers.\n"
         << "Mark the end of the list with a negative number.\n";
```

Display 7.9 (1/3)

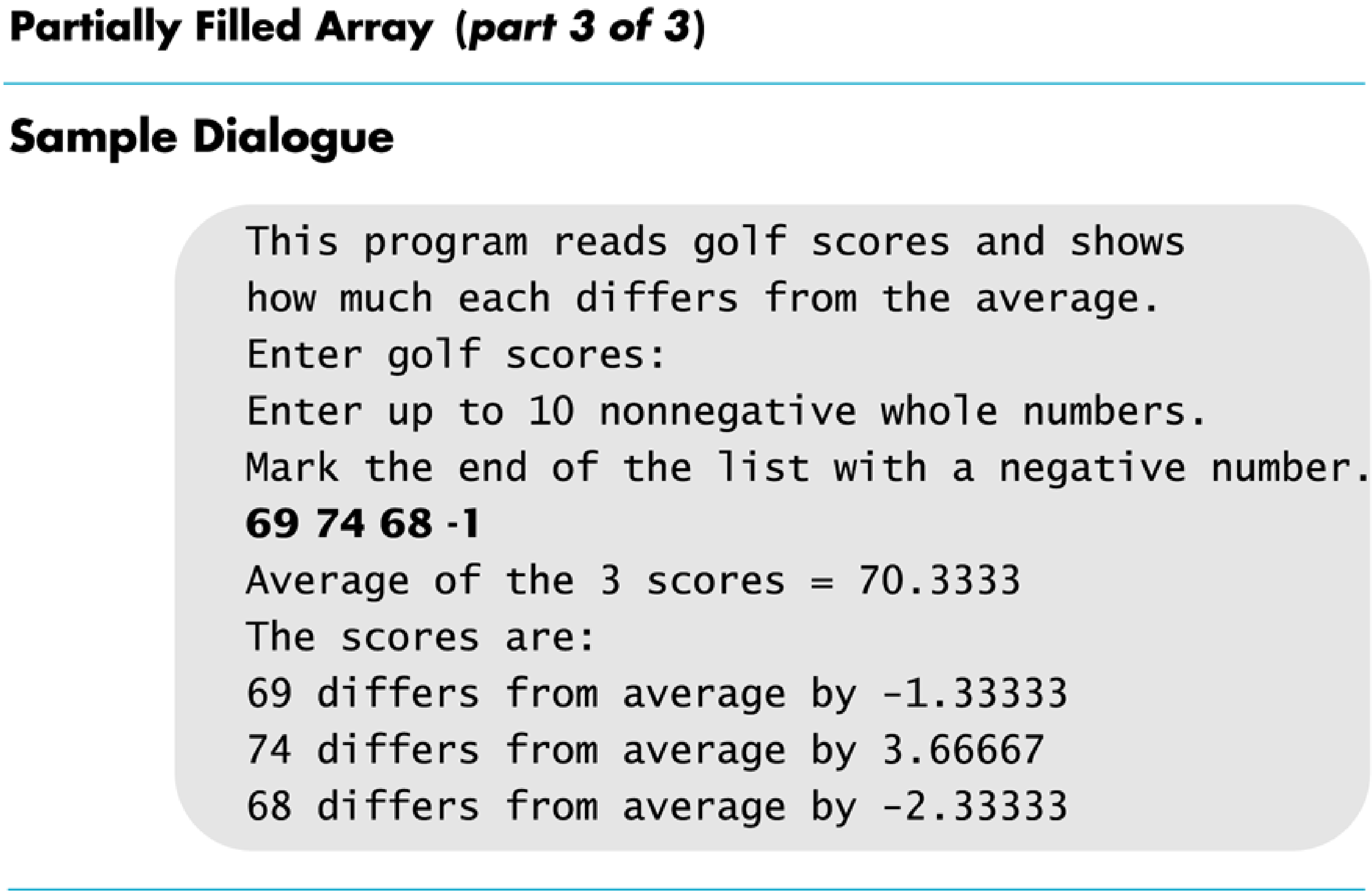
Display 7.9 (2/3)



Partially Filled Array *(part 2 of 3)*

```
    int next, index = 0;
    cin >> next;
    while ((next >= 0) && (index < size))
    {
        a[index] = next;
        index++;
        cin >> next;
    }

    number_used = index;
}

double compute_average(const int a[], int number_used)
{
    double total = 0;
    for (int index = 0; index < number_used; index++)
        total = total + a[index];
    if (number_used > 0)
    {
        return (total/number_used);
    }
    else
    {
        using namespace std;
        cout << "ERROR: number of elements is 0 in compute_average.\n"
             << "compute_average returns 0.\n";
        return 0;
    }
}

void show_difference(const int a[], int number_used)
{
    using namespace std;
    double average = compute_average(a, number_used);
    cout << "Average of the " << number_used
         << " scores = " << average << endl
         << "The scores are:\n";
    for (int index = 0; index < number_used; index++)
    cout << a[index] << " differs from average by "
         << (a[index] – average) << endl;
}
```

Display 7.9 (3/3)



Partially Filled Array *(part 3 of 3)*

Sample Dialogue

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
69 74 68 -1
Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333
```

Searching Arrays

- A sequential search is one way to search an array for a given value
  - Look at each element from first to last to see if the target value is equal to any of the array elements
  - The index of the target value can be returned to indicate where the value was found in the array
  - A value of -1 can be returned if the value was not found

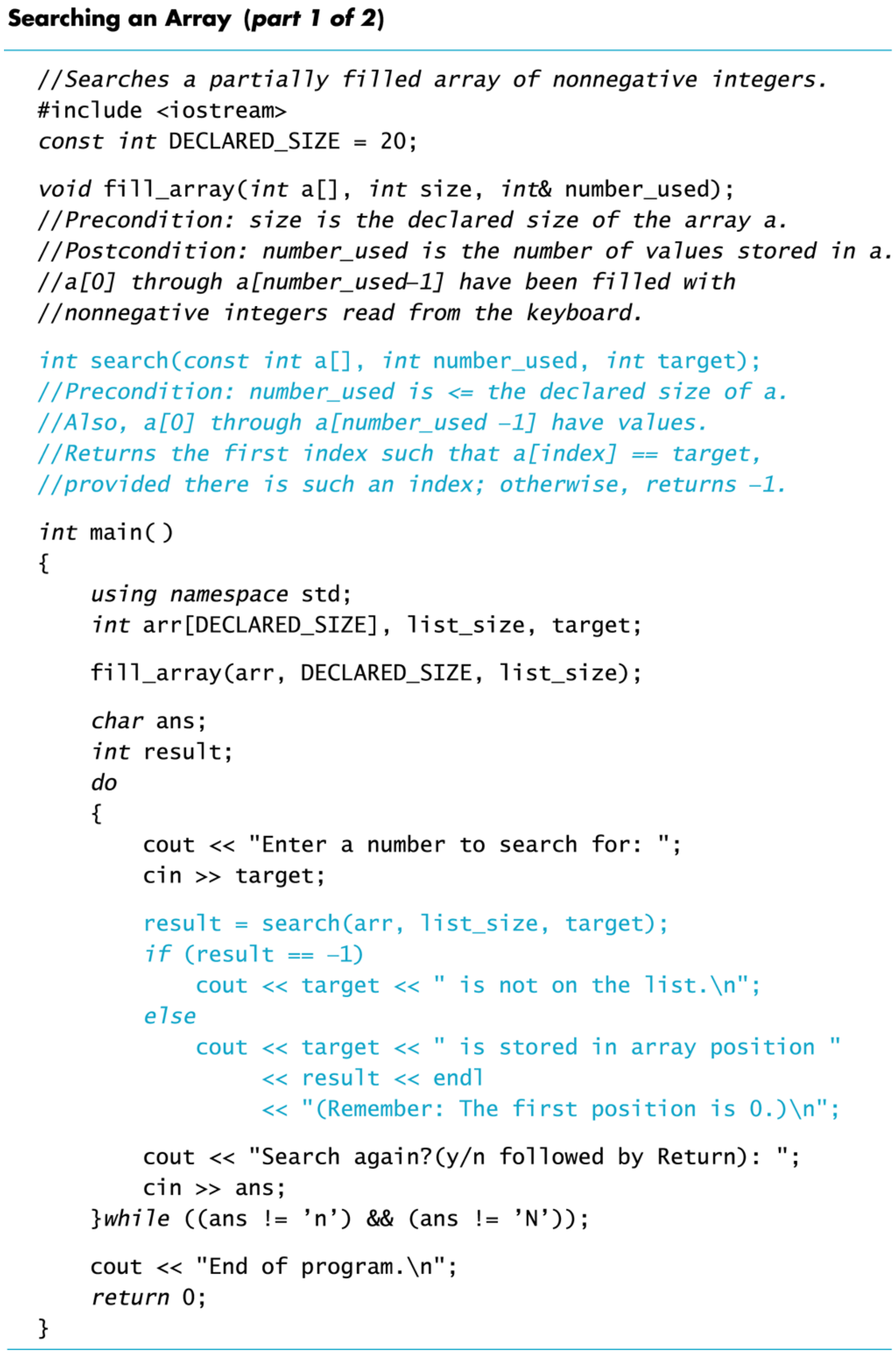
The search Function

- The search function of Display 7.10…
  - Uses a while loop to compare array elements to the target value
  - Sets a variable of type `bool` to true if the target value is found, ending the loop
  - Checks the boolean variable when the loop ends to see if the target value was found
  - Returns the index of the target value if found, otherwise returns -1

Display 7.10 (1)

Display 7.10 (2)

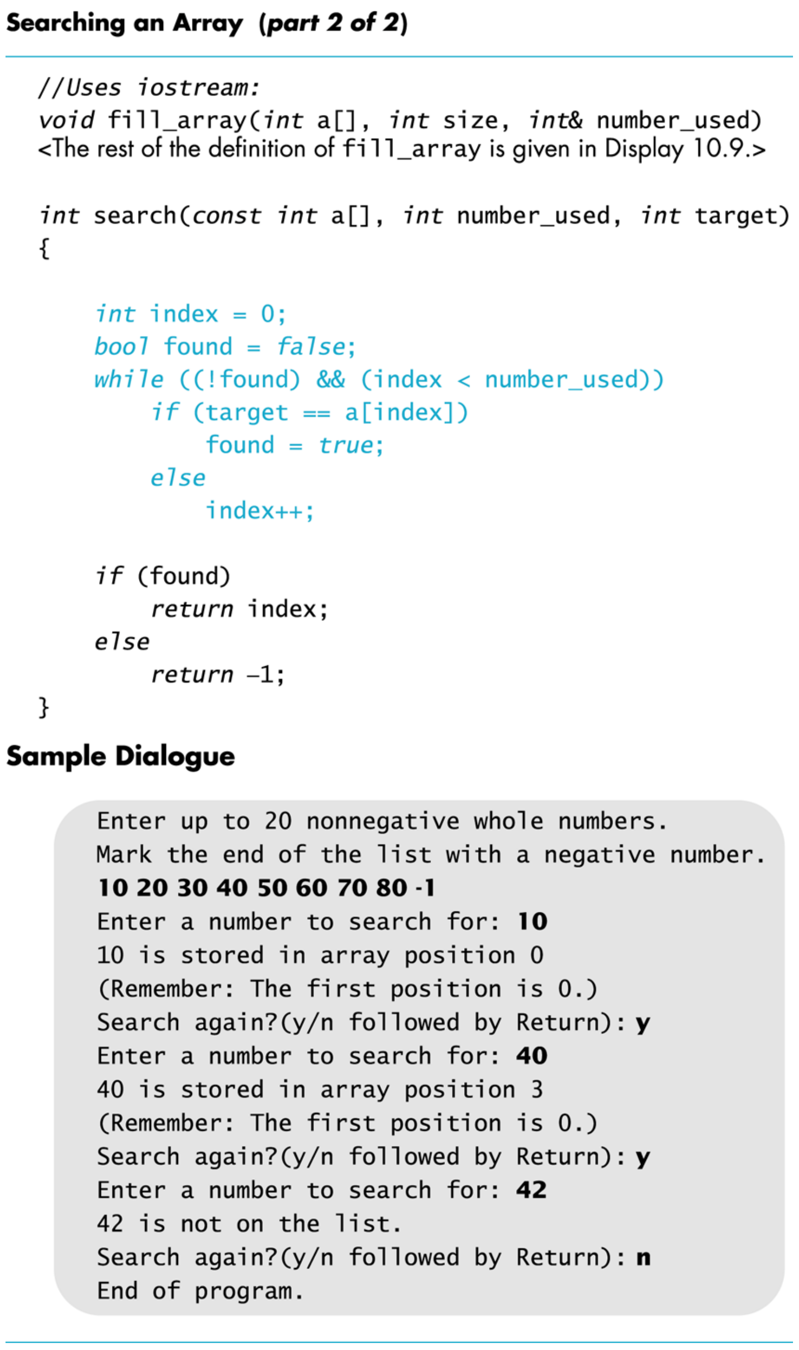
Display 7.10 (1/2)



Searching an Array ***(part 1 of 2)***

```
//Searches a partially filled array of nonnegative integers.
#include <iostream>
const int DECLARED_SIZE = 20;

void fill_array(int a[], int size, int& number_used);
//Precondition: size is the declared size of the array a.
//Postcondition: number_used is the number of values stored in a.
//a[0] through a[number_used–1] have been filled with
//nonnegative integers read from the keyboard.

int search(const int a[], int number_used, int target);
//Precondition: number_used is <= the declared size of a.
//Also, a[0] through a[number_used –1] have values.
//Returns the first index such that a[index] == target,
//provided there is such an index; otherwise, returns –1.

int main( )
{
    using namespace std;
    int arr[DECLARED_SIZE], list_size, target;

    fill_array(arr, DECLARED_SIZE, list_size);

    char ans;
    int result;
    do
    {
        cout << "Enter a number to search for: ";
        cin >> target;

        result = search(arr, list_size, target);
        if (result == –1)
            cout << target << " is not on the list.\n";
        else
            cout << target << " is stored in array position "
                 << result << endl
                 << "(Remember: The first position is 0.)\n";

        cout << "Search again?(y/n followed by Return): ";
        cin >> ans;
    }while ((ans != ’n’) && (ans != ’N’));

    cout << "End of program.\n";
    return 0;
}
```

Display 7.10 (2/2)



Searching an Array ***(part 2 of 2)***

```
//Uses iostream:
void fill_array(int a[], int size, int& number_used)
<The rest of the definition of fill_array is given in Display 10.9.>

int search(const int a[], int number_used, int target)
{

    int index = 0;
    bool found = false;
    while ((!found) && (index < number_used))
        if (target == a[index])
            found = true;
        else
            index++;

    if (found)
        return index;
    else
        return -1;
}
```

Sample Dialogue

```
Enter up to 20 nonnegative whole numbers.
Mark the end of the list with a negative number.
10 20 30 40 50 60 70 80 -1
Enter a number to search for: 10
10 is stored in array position 0
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 40
40 is stored in array position 3
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Return): n
End of program.
```

Go over this page:
http://storm.cis.fordham.edu/~zhang/cs2000/grading.html

Also documentation for function declaration, definition.

Program Example: Sorting an Array

- Sorting a list of values is very common task
  - Create an alphabetical listing
  - Create a list of values in ascending order
  - Create a list of values in descending order
- Many sorting algorithms exist
  - Some are very efficient
  - Some are easier to understand

Program Example: The Selection Sort Algorithm

- When the sort is complete, the elements of the array are ordered such that

  $a[0] < a[1] < \ldots < a[\text{number\_used} - 1]$

  Outline of the algorithm

```
for (int index = 0; index < number_used; index++)
    place the index-th smallest element in a[index]
```

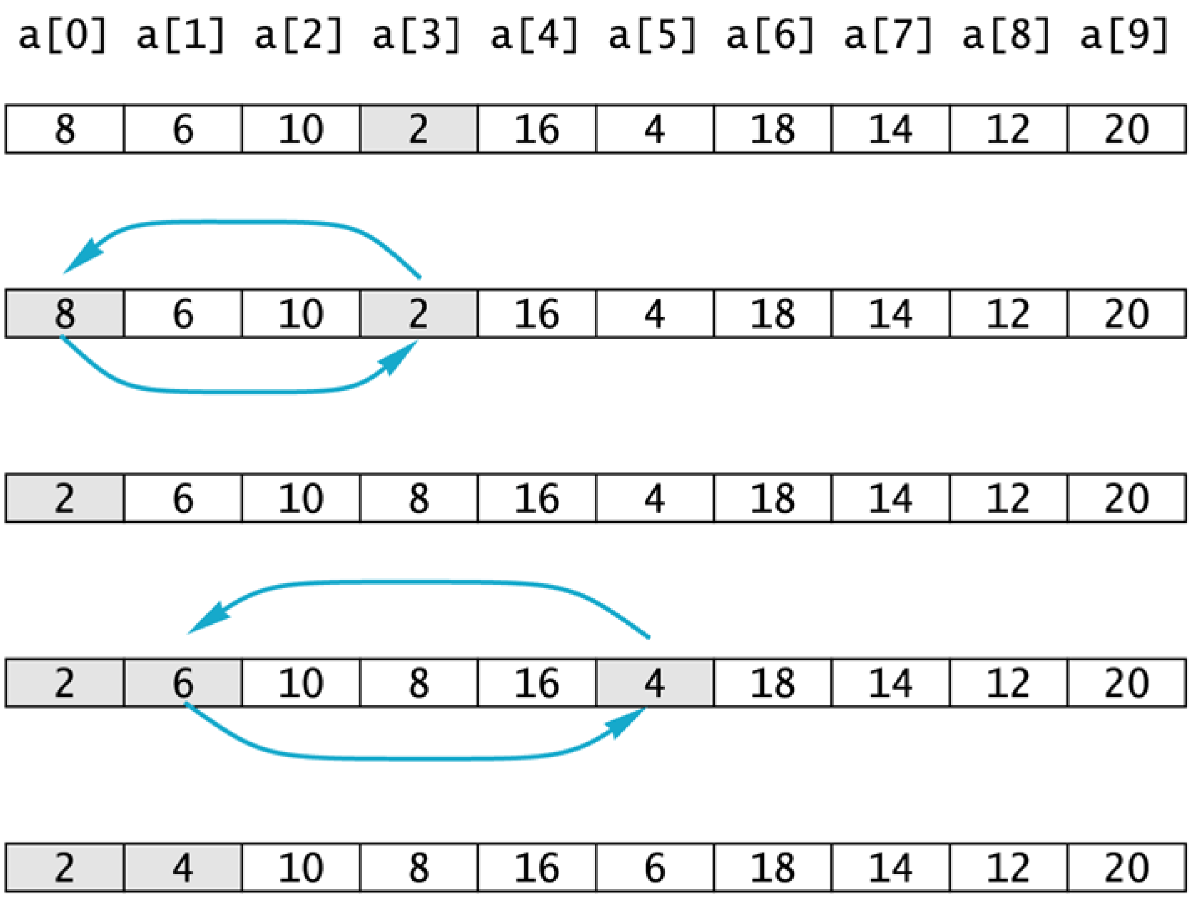
Program Example:
Sort Algorithm Development

- One array is sufficient to do our sorting
  - Search for the smallest value in the array
  - Place this value in a[0], and place the value that was in a[0] in the location where the smallest was found
  - Starting at a[1], find the smallest remaining value swap it with the value currently in a[1]
  - Starting at a[2], continue the process until the array is sorted

Display 7.11 **Display 7.12 (1-2)**

Display 7.11

Selection Sort

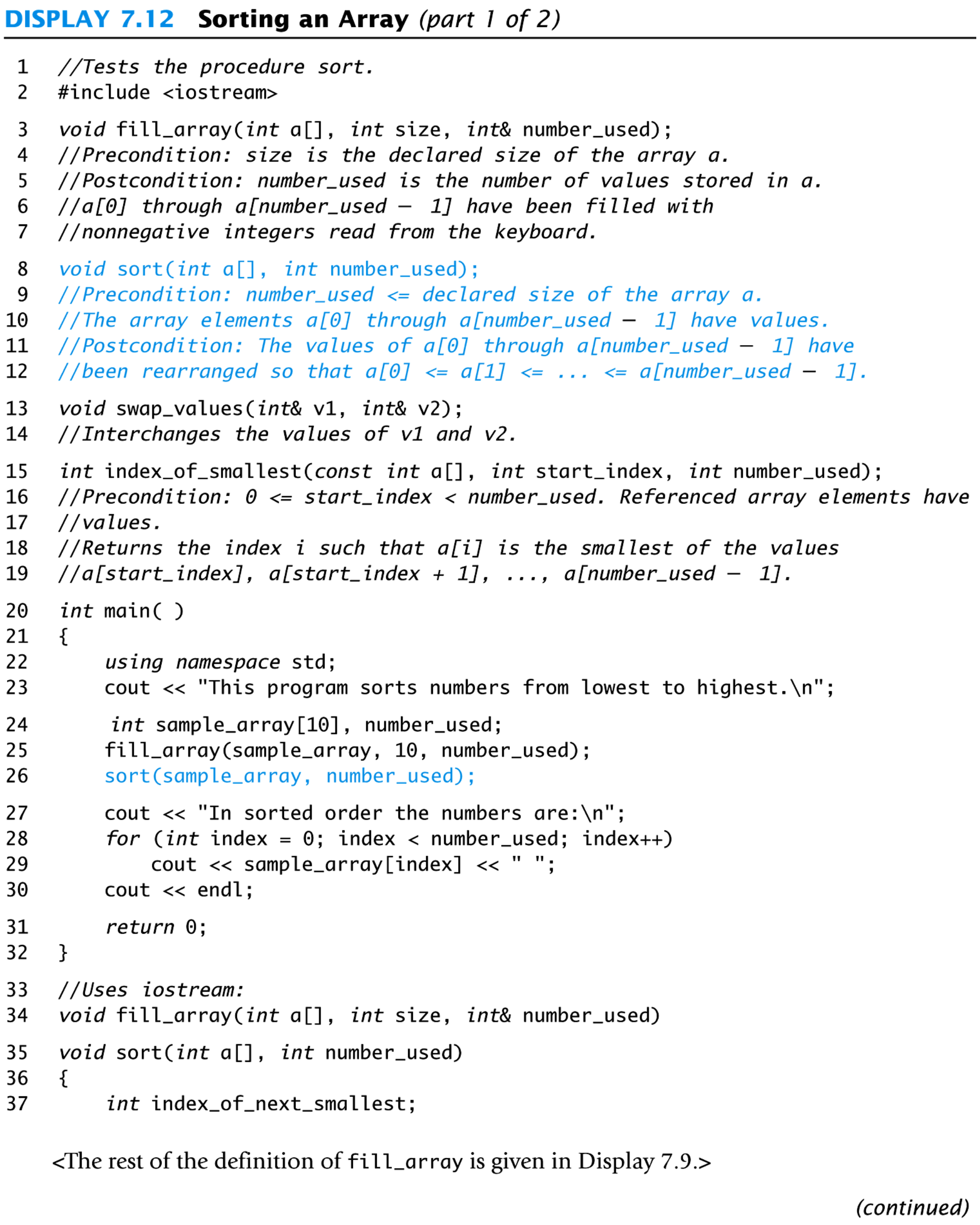


a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	20
8	6	10	2	16	4	18	14	12	20
2	6	10	8	16	4	18	14	12	20
2	6	10	8	16	4	18	14	12	20
2	4	10	8	16	6	18	14	12	20

go over the source code

http://storm.cis.fordham.edu/~zhang/cs2000/CodeExample_Savitch/Chapter07/07-12.cpp

Display 7.12 (1/2)



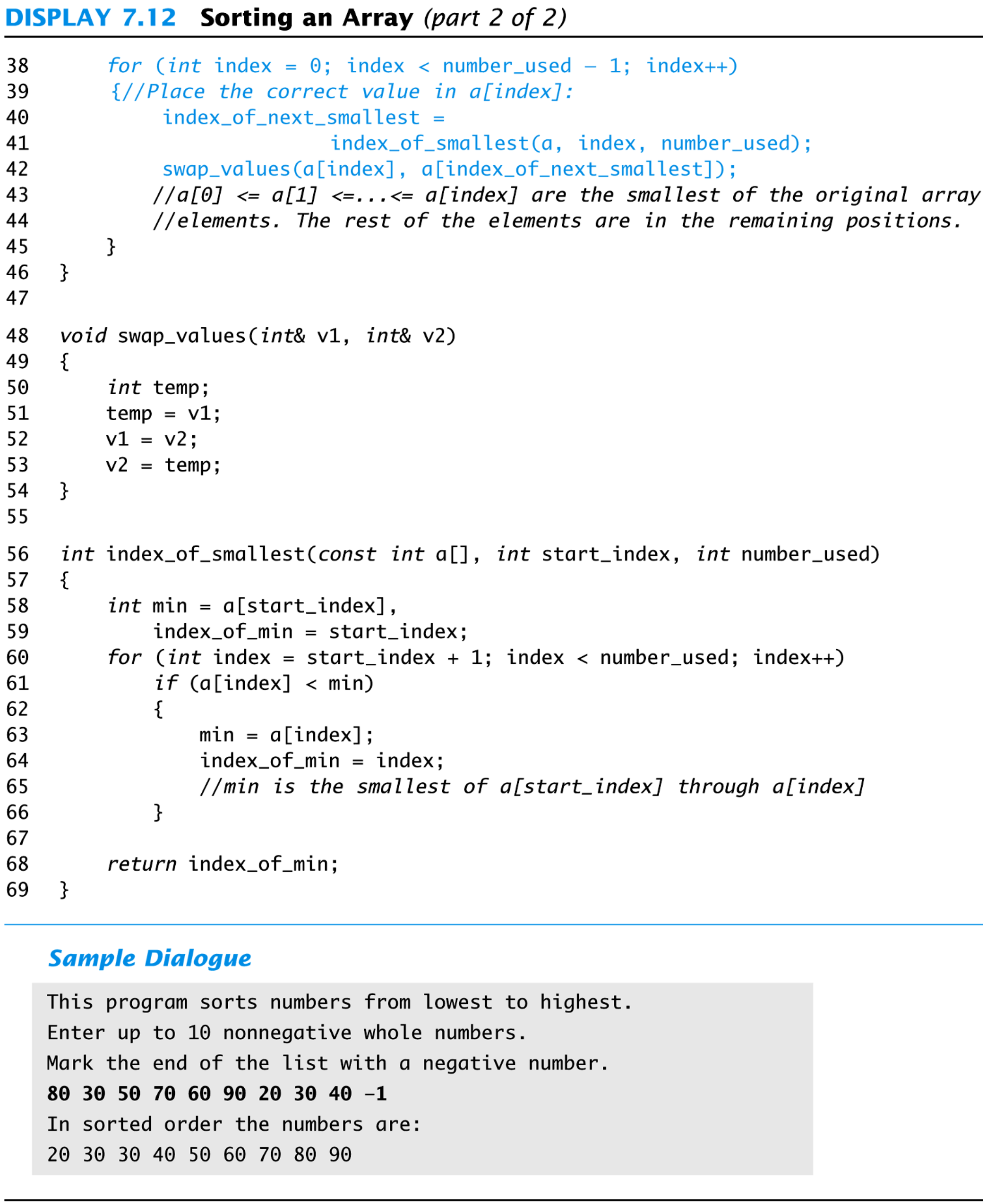
DISPLAY 7.12 Sorting an Array *(part 1 of 2)*

```
//Tests the procedure sort.
#include <iostream>

void fill_array(int a[], int size, int& number_used);
//Precondition: size is the declared size of the array a.
//Postcondition: number_used is the number of values stored in a.
//a[0] through a[number_used – 1] have been filled with
//nonnegative integers read from the keyboard.

void sort(int a[], int number_used);
//Precondition: number_used <= declared size of the array a.
//The array elements a[0] through a[number_used – 1] have values.
//Postcondition: The values of a[0] through a[number_used – 1] have
//been rearranged so that a[0] <= a[1] <= ... <= a[number_used – 1].

void swap_values(int& v1, int& v2);
//Interchanges the values of v1 and v2.

int index_of_smallest(const int a[], int start_index, int number_used);
//Precondition: 0 <= start_index < number_used. Referenced array elements have
//values.
//Returns the index i such that a[i] is the smallest of the values
//a[start_index], a[start_index + 1], ..., a[number_used – 1].

int main( )
{
    using namespace std;
    cout << "This program sorts numbers from lowest to highest.\n";

    int sample_array[10], number_used;
    fill_array(sample_array, 10, number_used);
    sort(sample_array, number_used);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < number_used; index++)
        cout << sample_array[index] << " ";
    cout << endl;

    return 0;
}

//Uses iostream:
void fill_array(int a[], int size, int& number_used)

void sort(int a[], int number_used)
{
    int index_of_next_smallest;
```

<The rest of the definition of fill_array is given in Display 7.9.>

(continued)

Display 7.12 (2/2)



DISPLAY 7.12 Sorting an Array *(part 2 of 2)*

```
    for (int index = 0; index < number_used – 1; index++)
    {//Place the correct value in a[index]:
         index_of_next_smallest =
                       index_of_smallest(a, index, number_used);
         swap_values(a[index], a[index_of_next_smallest]);
        //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
        //elements. The rest of the elements are in the remaining positions.
    }
}

void swap_values(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}

int index_of_smallest(const int a[], int start_index, int number_used)
{
    int min = a[start_index],
        index_of_min = start_index;
    for (int index = start_index + 1; index < number_used; index++)
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
            //min is the smallest of a[start_index] through a[index]
        }

    return index_of_min;
}
```

Sample Dialogue

```
This program sorts numbers from lowest to highest.
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
80 30 50 70 60 90 20 30 40 -1
In sorted order the numbers are:
20 30 30 40 50 60 70 80 90
```

Exercise

- Write a program that will read up to 10 letters into an array and write the letters back to the screen in the reverse order?

  abcd should be output as dcba

  Use a period as a sentinel value to mark the end of input

A side note:

Recall variables and memory

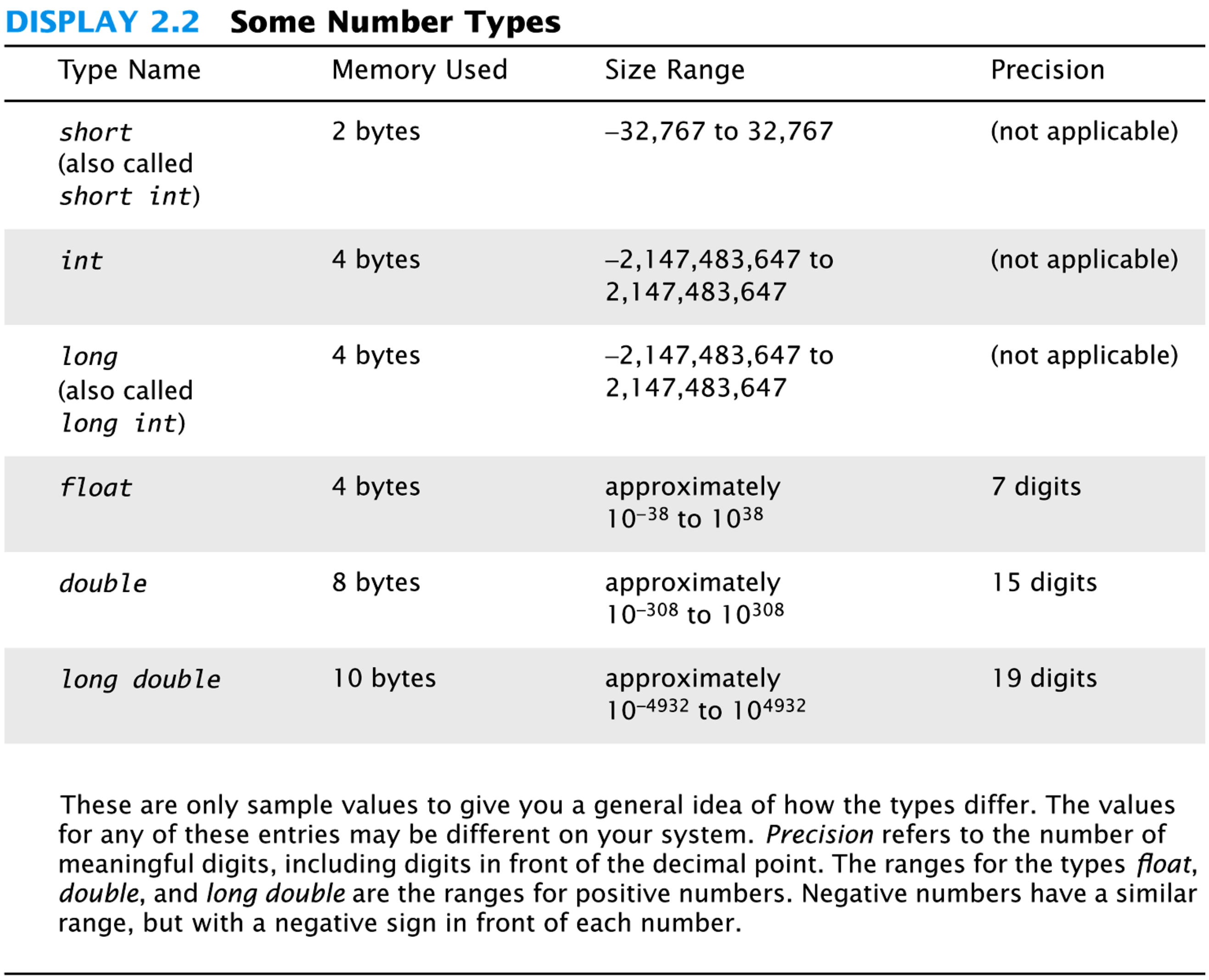
Computer Memory

- Computer memory consists of numbered locations called bytes
  - A byte's number is its address

- A simple variable is stored in consecutive bytes
  - The number of bytes depends on the variable's type

- A variable's address is the address of its first byte

Recall ...



DISPLAY 2.2 Some Number Types

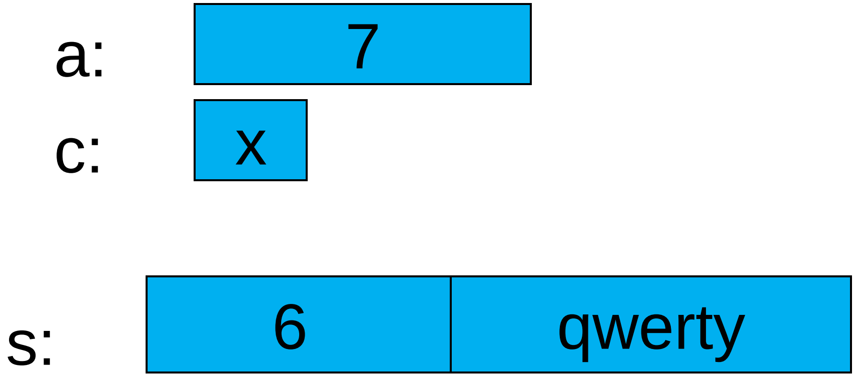
Type Name	Memory Used	Size Range	Precision
short (also called *short int*)	2 bytes	–32,767 to 32,767	(not applicable)
int	4 bytes	–2,147,483,647 to 2,147,483,647	(not applicable)
long (also called *long int*)	4 bytes	–2,147,483,647 to 2,147,483,647	(not applicable)
float	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
double	8 bytes	approximately 10^{-308} to 10^{308}	15 digits
long double	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits

These are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types *float*, *double*, and *long double* are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

Recall: types and Objects

- A type defines a set of possible values and a set of operations
- A value is a sequence of bits in memory, interpreted according to its type
- An object is a piece of memory that holds a value of a given type

```
int a = 7;
char c = 'x';
string s = "qwerty";
```



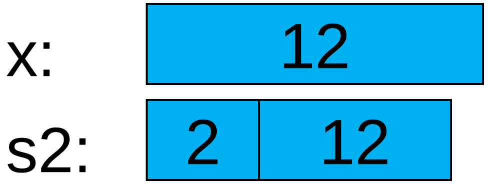
String object keeps the # of chars in the string, and the chars ..
We will learn how to access each char, s[0], s[1], …

More example

❑ What's the difference?

double x=12;

string s2="12";



1. x stores the value of number 12

s2 stores the two characters, '1','2'

2. applicable operations are different

x: arithmetic operations, numerical comparison,

s2: string concatenation, string comparison

value: a sequence of bits in memory

- interpreted according to a type
- E,g, int x=8;

x:
8

represented in memory as a seq. of binary digits (i.e., bits):

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

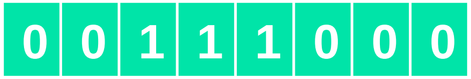
- An integer value is stored using the value's binary representation
  - In everyday life, we use decimal representation

value: a sequence of bits in memory (cont’d)

- interpreted according to a type
- E,g, char x=‘8’;

x: ‘8’

- is represented in memory as a seq. of binary digits (i.e., bits)

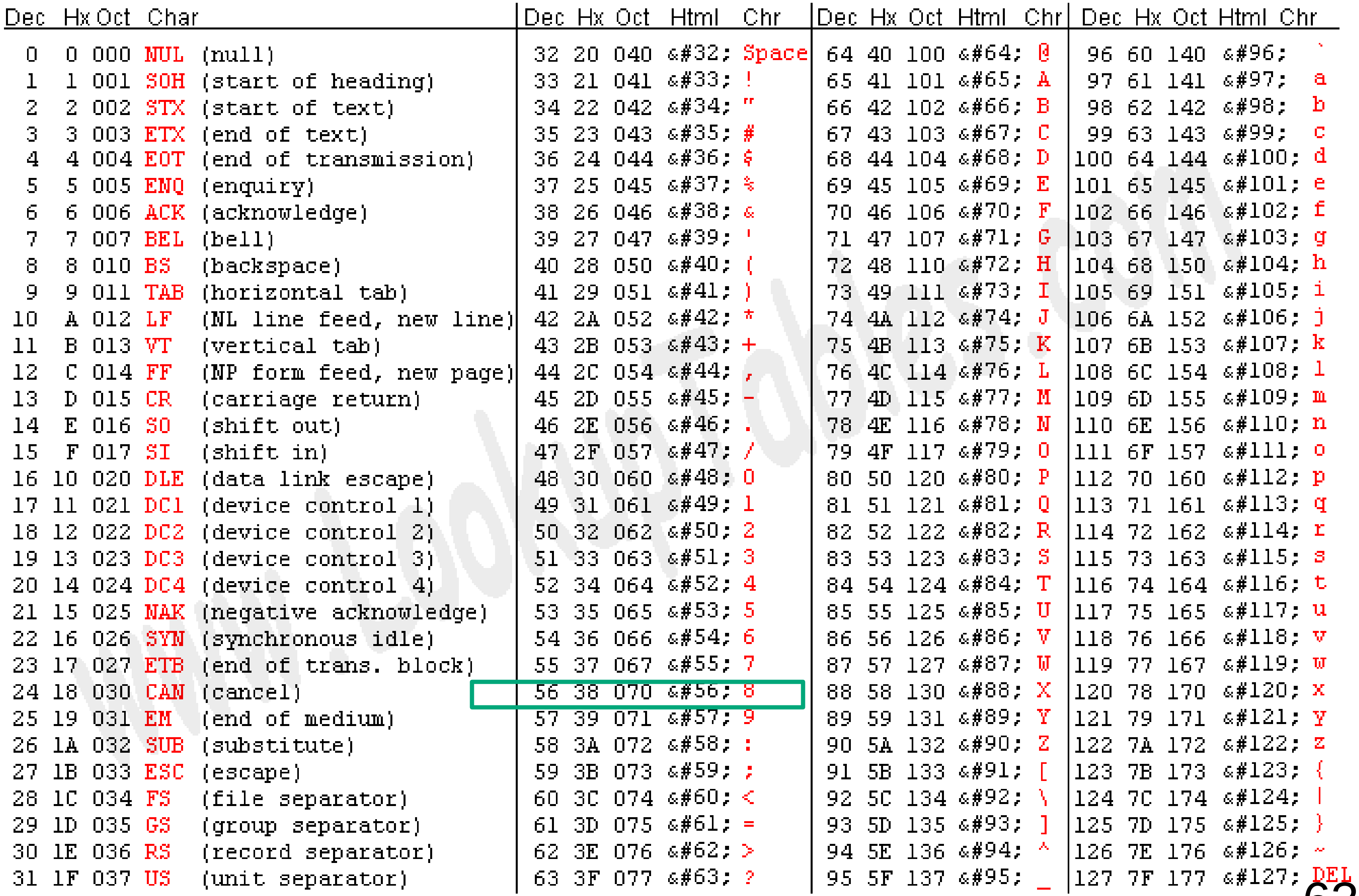


0	0	1	1	1	0	0	0

- A char value is stored using char’s ASCII code

(American Standard Code for Information Interchange)

ASCII Code

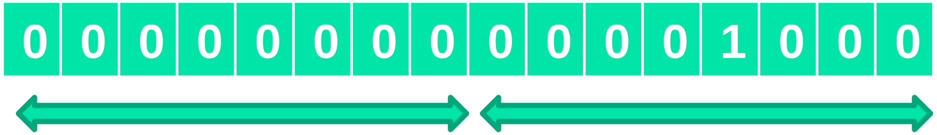


Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Interpretation of a bit sequence

- Given a bit sequence in memory



0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

- If it's interpreted as integer, then it represents value 8
  - $1*2^3=8$

- If interpreted as char, there are two chars, a **NULL** char, and a **BACKSPACE** char

A technical detail

- In computer memory, everything is just bits; type is what gives meaning to the bits

```
char c = 'a';
cout << c;    // print the value of character variable c, which is a
int i = c;
cout << i;    // print the integer value of the character c, which is 97
```

int i = c;

Left-hand-side (LHS) is an **int** type variable

Right-hand-side (RHS) is a value of **char** type

- *Assign a **char** value to a **int** type variable ?!*
- *A safe type conversion.*

Sizeof operator

Yields size of its operand
Measured by the size of
type **char**, i.e., a byte

```
cout <<"sizeof bool is " << sizeof (bool)  << "\n"
     <<"sizeof char is " << sizeof (char)  << "\n"
     <<"sizeof int is " << sizeof (int)  << "\n"
     <<"sizeof short is " << sizeof (short)  << "\n"
     <<"sizeof long is " << sizeof (long)  << "\n"
     <<"sizeof double is " << sizeof (double)  << "\n"
     <<"sizeof float is " << sizeof (float)  << "\n";
```

```
sizeof bool is 1
sizeof char is 1
sizeof int is 4
sizeof short is 2
sizeof long is 8
sizeof double is 8
sizeof float is 4
```

Char-to-int conversion

c: 01100001

```
char c = 'a';
cout << c;        // print the value of character variable c, which is a
int i = c;
cout << i;        // print the integer value of the character c, which is 97
```

i: 00000000000000000000000001100001

- ❑ No information is lost in the conversion

  char c2=i; //c2 has same value as c

  - Can convert int back to char type, and get the original value
- ❑ Safe conversion:
  - **bool** to char, int, double
  - **char** to int, double
  - **int** to double

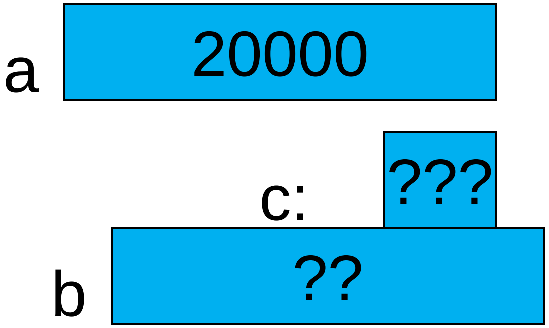
```
#include <iostream>
using namespace std;
int main()
{
   int pennies = 8;                    //what if change 8 to "eight"?
   int dimes = 4;
   int quarters = 3;

   double total = pennies * 0.01 + dimes * 0.10
       + quarters * 0.25;  // Total value of the coins
   cout << "Total value = " << total << "\n";
   return 0;
}
```

Implicit type conversion
  int to double

A type-safety violation ("implicit narrowing")

Beware: C++ does not prevent you from trying to **put a large value into a small variable** (though a compiler may warn)

```
int main()
{
    int a = 20000;
    char c = a;
    int b = c;
    if (a != b)                    //  != means "not equal"
            cout << "oops!: " << a << "!=" << b << '\n';
    else
            cout << "Wow! We have large characters\n";
}
```



"narrowing" conversion

```
int main()
{
    double d =0;
    while (cin>>d) {   // repeat the statements below
                    // as long as we type in numbers
      int i = d;      // try to squeeze a double into an int
      char c = i;     // try to squeeze an int into a char
      int i2 = c;     // get the integer value of the character
      cout << "d==" << d              // the original double
           << " i=="<< i              // converted to int
           << " i2==" << i2           // int value of char
           << " char(" << c << ")\n"; // the char
  }
```

A type-safety violation

(Uninitialized variables)

// Beware: C++ does not prevent you from trying to use a variable before you have initialized it (though a compiler typically warns)

```
int main()
{
    int x;              // x gets a "random" initial value
    char c;             // c gets a "random" initial value
    double d;           // d gets a "random" initial value
                        //      – not every bit pattern is a valid floating-point value
    double dd = d;              // potential error: some implementations
                                // can't copy invalid floating-point values
    cout << " x: " << x << " c: " << c << " d: " << d << '\n';
}
```

- Always initialize your variables
  - valid exception to this rule: input variable

Multi-dimensional Array

Read Section 7.4

Multi-Dimensional Arrays

- C++ allows arrays with multiple index values
  - char page [30] [100];
    declares an array of characters named page
    - page has two index values:
      The first ranges from 0 to 29
      The second ranges from 0 to 99
  - Each index in enclosed in its own brackets
  - Page can be visualized as an array of
    30 rows and 100 columns

Index Values of page

- The **indexed variables** for array page are
  page[0][0], page[0][1], ..., page[0][99]
  page[1][0], page[1][1], ..., page[1][99]
  …
  page[29][0], page[29][1], … , page[29][99]

- page is actually an array of size 30
  - page's base type is an array of 100 characters

Multidimensional Array Parameters

- Recall that the size of an array is not needed when declaring a formal parameter:
  void display_line(const char a[ ], int size);
- The base type of a multi-dimensional array must be completely specified in the parameter declaration
- C++ treats **a** as an array of arrays
  - void display_page(const char page[ ] [100],
    int size_dimension_1);

Program Example: Grading Program

- Grade records for a class can be stored in a two-dimensional array
  - For a class with 4 students and 3 quizzes the array could be declared as

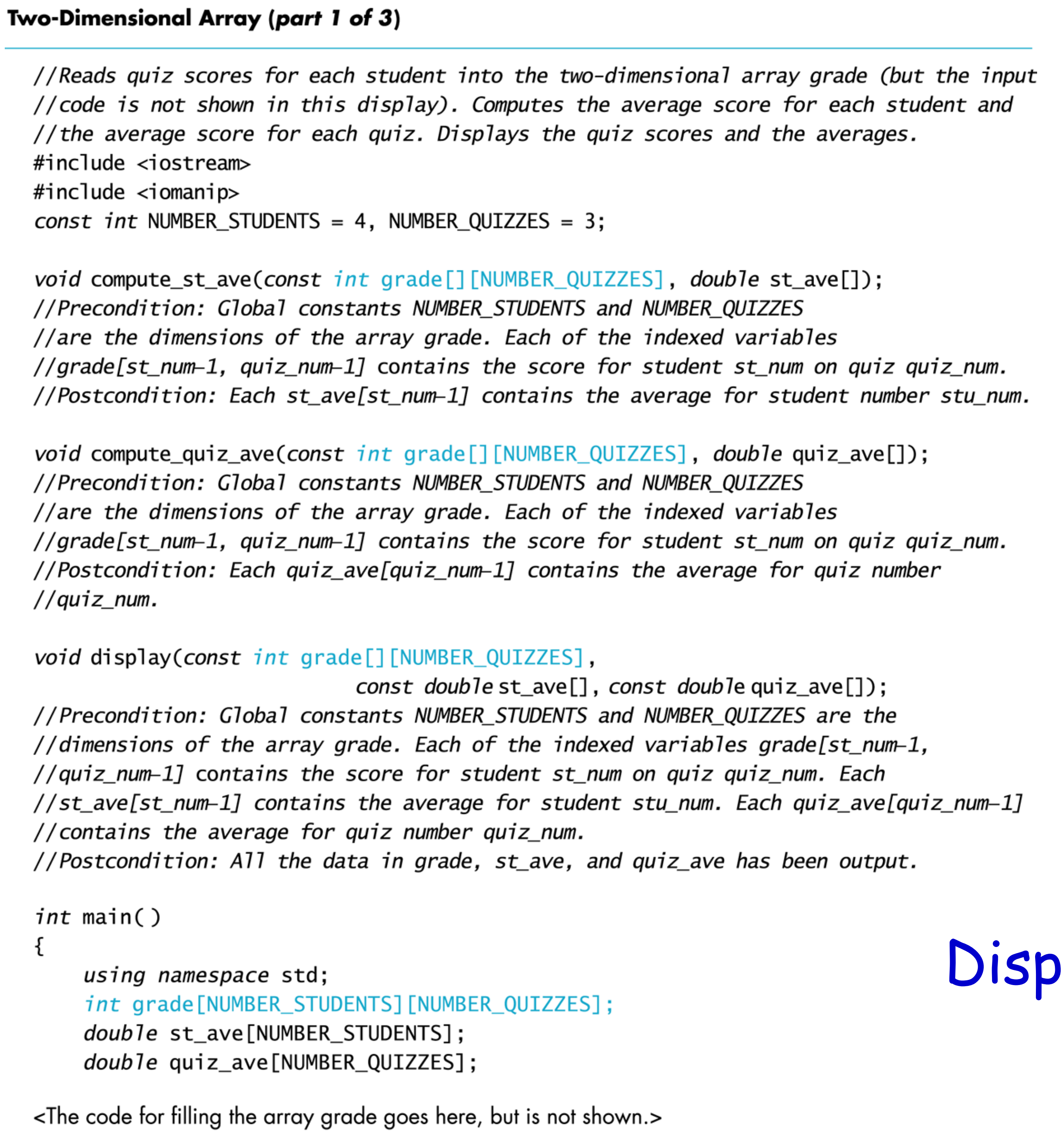
```
int grade[4][3];
```

    - The first array index refers to the number of a student
    - The second array index refers to a quiz number
- Since student and quiz numbers start with one, we subtract one to obtain the correct index

Grading Program: average scores

- The grading program uses one-dimensional arrays to store…
  - Each student's average score
  - Each quiz's average score
- The functions that calculate these averages use global constants for the size of the arrays
  - This was done because the functions seem to be particular to this program

Display 7.13 (1-3)

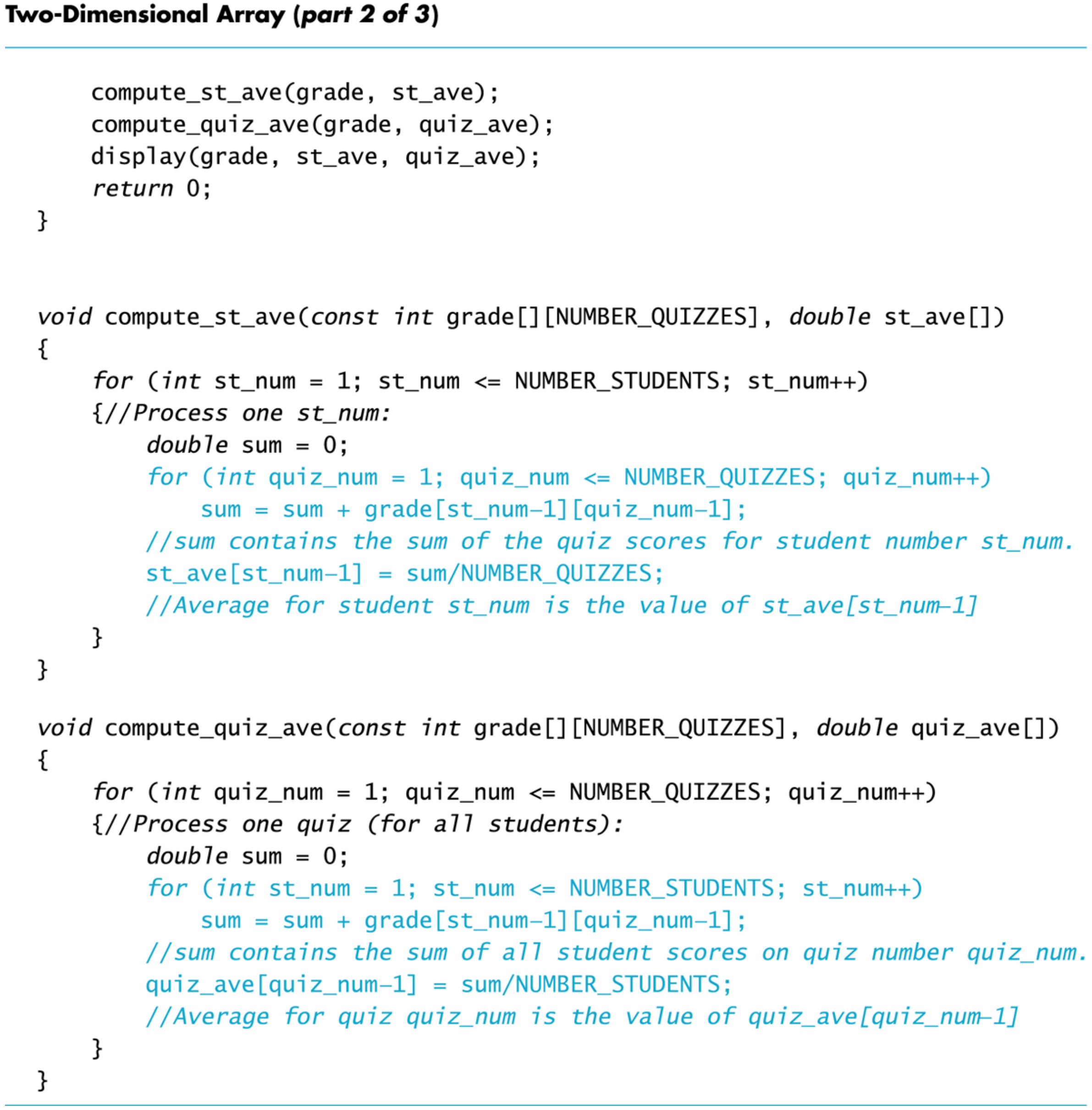


Two-Dimensional Array (*part 1 of 3*)

```
//Reads quiz scores for each student into the two-dimensional array grade (but the input
//code is not shown in this display). Computes the average score for each student and
//the average score for each quiz. Displays the quiz scores and the averages.
#include <iostream>
#include <iomanip>
const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[]);
//Precondition: Global constants 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 st_ave[st_num-1] contains the average for student number stu_num.

void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[]);
//Precondition: Global constants 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 number
//quiz_num.

void display(const int grade[][NUMBER_QUIZZES],
                              const double st_ave[], const double quiz_ave[]);
//Precondition: Global constants 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 number quiz_num.
//Postcondition: All the data in grade, st_ave, and quiz_ave has been output.

int main( )
{
    using namespace std;
    int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
    double st_ave[NUMBER_STUDENTS];
    double quiz_ave[NUMBER_QUIZZES];

<The code for filling the array grade goes here, but is not shown.>
```

Display 7.13 (1/3)

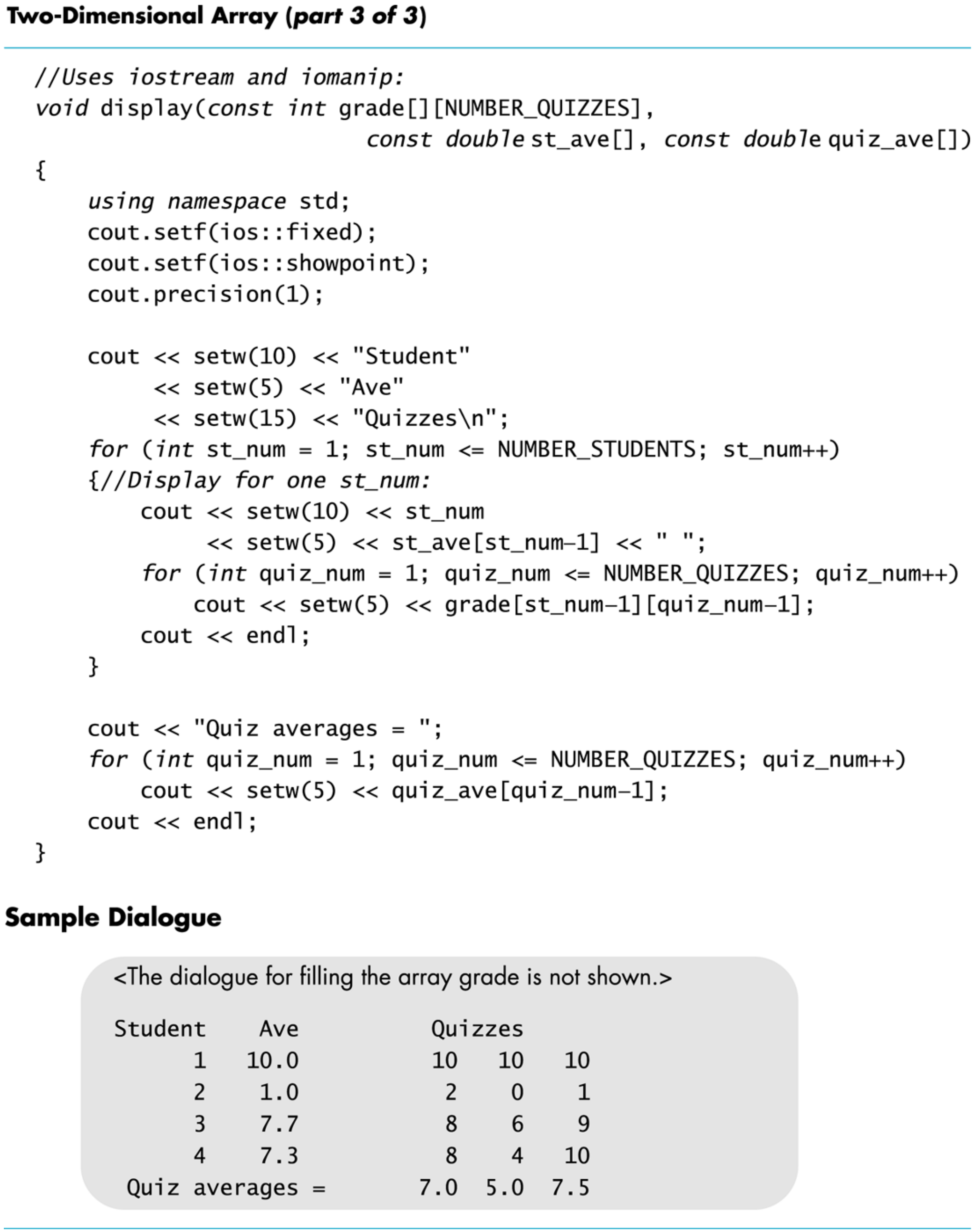
Display 7.13 (2/3)



Two-Dimensional Array (*part 2 of 3*)

```
    compute_st_ave(grade, st_ave);
    compute_quiz_ave(grade, quiz_ave);
    display(grade, st_ave, quiz_ave);
    return 0;
}


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 = sum + grade[st_num-1][quiz_num-1];
        //sum contains the sum of the quiz scores for student number st_num.
        st_ave[st_num-1] = sum/NUMBER_QUIZZES;
        //Average for student st_num is the value of st_ave[st_num-1]
    }
}

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 = sum + grade[st_num-1][quiz_num-1];
        //sum contains the sum of all student scores on quiz number quiz_num.
        quiz_ave[quiz_num-1] = sum/NUMBER_STUDENTS;
        //Average for quiz quiz_num is the value of quiz_ave[quiz_num-1]
    }
}
```

Display 7.13 (3/3)



Two-Dimensional Array (*part 3 of 3*)

```
//Uses iostream and iomanip:
void display(const int grade[][NUMBER_QUIZZES],
                         const double st_ave[], const double quiz_ave[])
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);

    cout << setw(10) << "Student"
         << setw(5) << "Ave"
         << setw(15) << "Quizzes\n";
    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;
}
```

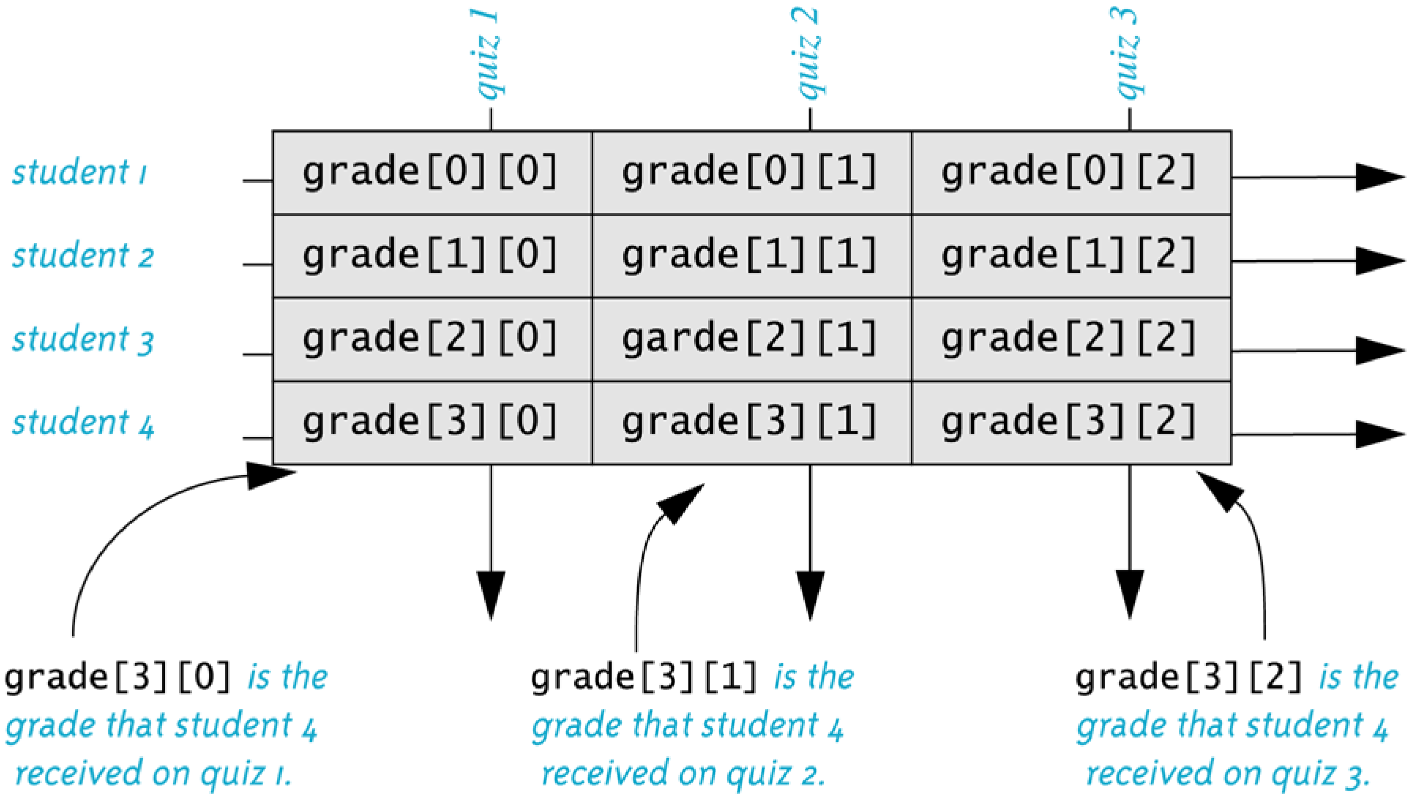
Sample Dialogue

<The dialogue for filling the array grade is not shown.>

```
Student   Ave         Quizzes
      1  10.0        10   10   10
      2   1.0         2    0    1
      3   7.7         8    6    9
      4   7.3         8    4   10
 Quiz averages =    7.0  5.0  7.5
```

Display 7.14

The Two-Dimensional Array grade



	quiz 1	*quiz 2*	*quiz 3*
student 1	`grade[0][0]`	`grade[0][1]`	`grade[0][2]`
student 2	`grade[1][0]`	`grade[1][1]`	`grade[1][2]`
student 3	`grade[2][0]`	`garde[2][1]`	`grade[2][2]`
student 4	`grade[3][0]`	`grade[3][1]`	`grade[3][2]`

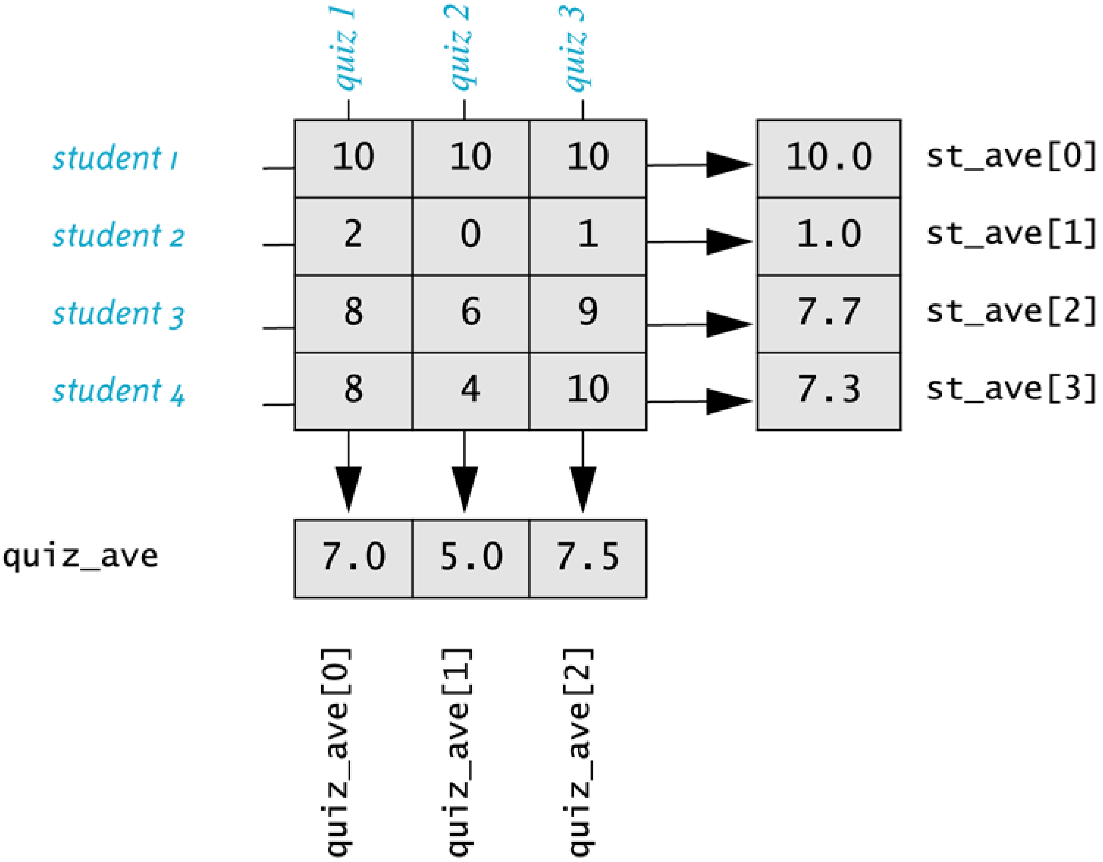
`grade[3][0]` *is the grade that student 4 received on quiz 1.*

`grade[3][1]` *is the grade that student 4 received on quiz 2.*

`grade[3][2]` *is the grade that student 4 received on quiz 3.*

Display 7.15

The Two-Dimensional Array grade (Another View)



	quiz 1	quiz 2	quiz 3		
student 1	10	10	10	10.0	st_ave[0]
student 2	2	0	1	1.0	st_ave[1]
student 3	8	6	9	7.7	st_ave[2]
student 4	8	4	10	7.3	st_ave[3]
quiz_ave	7.0	5.0	7.5		
	quiz_ave[0]	quiz_ave[1]	quiz_ave[2]		

Showing Decimal Places

- To specify fixed point notation
  - `setf(ios::fixed)`
- To specify that the decimal point will always be shown
  - `setf(ios::showpoint)`
- To specify that two decimal places will always be shown
  - `precision(2)`

- Example:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout   << "The price is "
       << price << endl;
```