# Arrays as user-defined types

- An array is a data type which holds a collection of variables of the same type
  - We can define an array type which holds 6 ints ← `int thisArray[6];`
  - Or 12 chars ← `char anotherArray[12];`
  - Or 8 strings ← `string nextArray[8];`
  - Or …
- Notice that this is quite a limited way to declare a new data-type
  - We have to define it as a datatype to use *just with the variable we have specified*
  - If there were 2 arrays of 6 ints, the data type 'array of 6 ints' has to be declared for each of them

```
int thisArray[6];

int thatArray[6];
```

an array of 7 `ints` would be a different data type

- It is possible to initialise the array when it is declared
  - `int thisArray[6] = {2, 4, 6, 8, 10, 12};`
  - If we then initialise less than the 6 (in this case)  elements of the array, the other array positions will be initialised to 0
  - But if we try to initialise more, there will be a compiler error
  - And if we don't initialise any, then all the array entries will contain garbage values – don't assume they are 0 in this case !!!!!

# So what is an array data type?

- It is a collection of data items arranged in a contiguous bit of memory
  - That means they come one after the other
  - So if the compiler knows where the first one is, it can find all the others
  - But the compiler has to know *exactly* how many there are, because it will set aside enough space to hold them in memory
  - You cannot decide to make the array larger later – the memory in the next contiguous space is probably being used for something else!
- Here’s what its not!
  - It is not a class … so it doesn’t have any methods

- Once the compiler has set up an array of the correct size for you, *it takes no more responsibility* for the size of the array
  - All that is stored in the array variable is the *address* of the place in memory where the array starts
  - There is no length method to use (because the array is not a class, and does not have any methods!)

# More ways to declare an array

- We could declare and initialise an array of 3 doubles like this

  ```
  double dblArray[] = {3.4, 6.7, 23.8};
  ```

  - In this case, we didn’t say explicitly how big the array of doubles was, but the compiler can work it out
  - Because it assumes all the array positions have been initialised

- We can also use an integer expression to dimension the array
  - But it has to be a constant expression

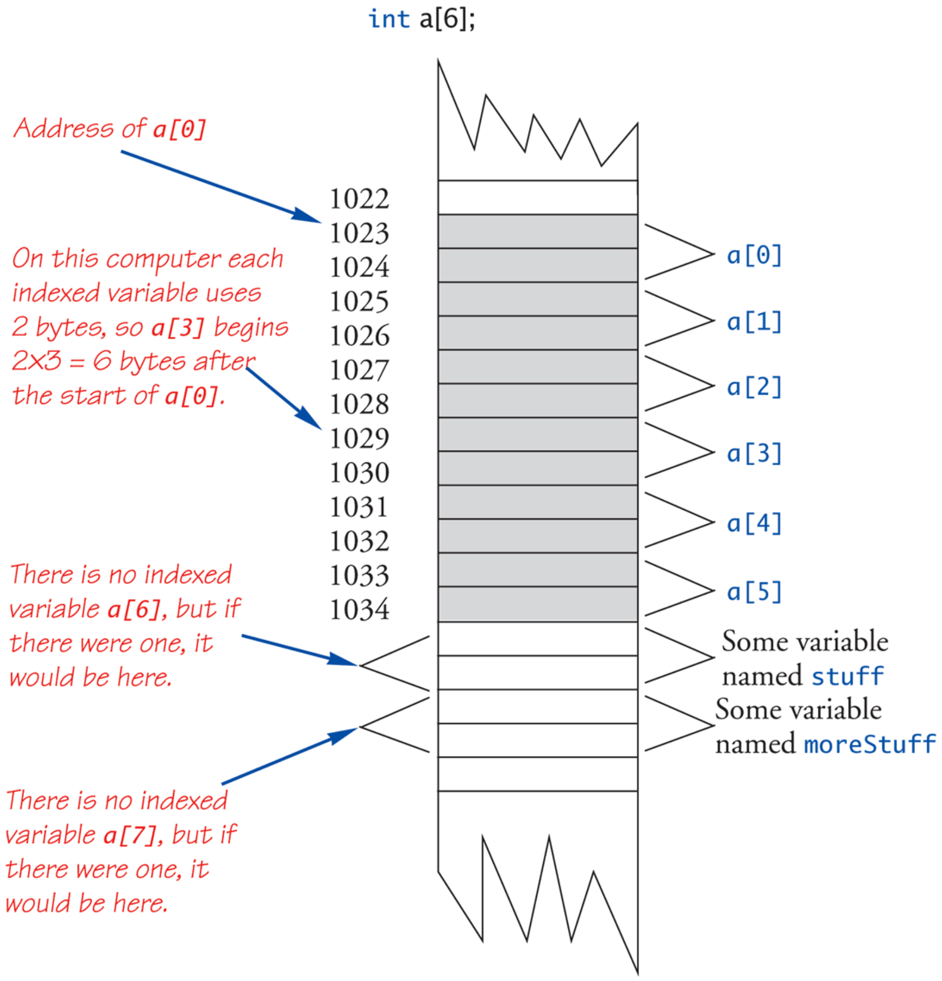
```
float newArray[9 * 5]; //new array is an array of floats of size 45

const int WEEK = 7;
string days[WEEK]; //this is OK, because WEEK is a constant variable

int month = 31;
int days[month] //but this is NOT OK, because month is not a constant
```

# The bounds of an array

- Once the compiler has set up an array of the correct size for you, *it takes no more responsibility* for the size of the array
  - We have said this once, but its worth repeating!
  - All that is stored in the array variable is the *address* of the place in memory where the array starts
- `int myArray[10]` sets up sequential, back-to-back memory locations for 10 ints, and stores the *address* of the first one in the variable `myArray`
- You can refer to the individual entries in the array
  - `myArray[0]` means the first entry in the array
  - `myArray[7]` means the 8th entry in the array
  - The compiler finds the place where each entry is stored by working forwards from what it knows - which is where the array starts
- If you try to change `myArray[12]` the compiler *will not stop you*. But it could have very nasty consequences at runtime, because you have just trashed a piece of memory being used for something else in your program!!!
  - A common mistake is to access or change the entry just one passed the end of the array ( in this case myArray[10] )
  - Remember the array index starts at zero and goes up to one less than the size of the array

# An Array in Memory

**Display 5.2** **An Array in Memory**



`int a[6];`

*Address of **a[0]***

*On this computer each indexed variable uses 2 bytes, so **a[3]** begins 2x3 = 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.*

Some variable named `stuff`

Some variable named `moreStuff`

From Savitch: chapter 5

# Try some yourself (from Savitch self-test exercise)

- Identify any errors in the following array declarations.
  - `int x[4] = { 8, 7, 6, 4, 3 };`
  - `int x[] = { 8, 7, 6, 4 };`
  - 
    ```
    const int SIZE = 4;
    int x[SIZE];
    ```

2. What is the output of the following code?

```
char symbol[3] = {'a', 'b', 'c'};
for (int index = 0; index < 3; index++)
cout << symbol[index];
```

3. What is wrong with the following piece of code?

```
int sampleArray[10];
for (int index = 1; index <= 10; index++)
    sampleArray[index] = 3*index;
```

- Suppose we expect the elements of the array a to be ordered so that
  a[0] <= a[1] <= a[2]<= ...
  However, to be safe we want our program to test the array and issue a warning in case it turns out that some elements are out of order. The following code is supposed to output such a warning, but it contains a bug. What is it?

```
double a[10];

<Some code to fill the array a goes here.>

for (int index = 0; index < 10; index++)

       if (a[index] > a[index + 1])

             cout   << "Array elements " << index
                    << " and ‟ << (index + 1)
                    << " are out of order.";
```

- Suppose you have the following array declaration in your program:

```
int yourArray[7];
```

  - Also, suppose that in your implementation of C++, variables of type int use two bytes of memory. When you run your program, how much memory will this array consume?
  - Suppose that, when you run your program, the system assigns the memory address 1000 to the indexed variable yourArray[0]. What will be the address of the indexed variable yourArray[3]?

# Passing an array element to a function

- A function that expects a single variable as an argument can be passed a single array indexed variable.

- For example if the function expects a double

  ```
  void function1(double myarg);
  ```

- .. and `myArray` is an array of doubles:

  ```
  double myArray[3];
  ```

- Then this call to the function is OK

  ```
      function1(myArray[2]);
  ```

- This call will also compile, but can you see a problem?

  ```
  function1(myArray[3]);
  ```

  - If `myArray[3]` is not in the array bounds, there will be a problem when the program is run

# Passing an array to a function

- Write a function which is passed an array of doubles, and displays the values in the array: FIRST ATTEMPT (POOR!!)

```
void displayDblAry(double[] );
```

The formal parameter in the declaration tells the compiler that this function will have an array of doubles as an argument

```
void displayDblAry(double theAry[] );
```

We could name the formal parameter in the declaration to help self-documentation

```
void displayDblAry(double theAry[] )
{
     for (int i = 0; i < ???; i++)
              cout << theAry[i] << ‘ ’;
}
```

We have a problem here because the function does not know the size of the array!

# Passing an array to a function

- We must pass the size of the array to the function as a second argument

- The function declaration

```
void displayDblAry(double[], int );
// OR
void displayDblAry(double theAry[], int arySize );
```

- The function implementation (aka function definition )

```
void displayDblAry(double theAry[], int arySize )
{
    for (int i = 0; i < arySize; i++)
          cout << theAry[i] << ‘ ’;
}
```

- A call to the function

```
double gpas[20];  //gpas is an array of 20 double values
//some code to assign values to the array goes here
displayDblAry(gpas, 20);
```

Just the array name is plugged in as an actual argument – no square brackets

# Passing an array to a function: by value or by reference?

```
void displayDblAry(double theAry[], int arySize )
{
      for (int i = 0; i < arySize; i++)
            cout << theAry[i] << ‘ ’;
}
```

- Has the array been passed by value or by reference?
  - In other words, is the array local inside the function
  - Or will a change to it change an array outside the function?

- Remember that the information contained in the array variable is just a memory address
  - It says where the array starts in memory
  - This cant be changed inside the function, so is it passed by value?
  - But if we change one of the array elements, we have gone to that address and changed a value in the array we were given
  - In other words, a change to an array element inside the function is changing the value outside the function too, so is it passed by reference?

# Passing by array-reference

- In fact, passing an array to a function is a special case, not quite passing by reference, and certainly not exactly passing by value
  - We call it passing by ***array-reference.***
- This means that the function can change elements of the array whether we like it or not! So we need the following rule for robust code:

  if a function with an array parameter is not meant to make changes to the values held in the array, then always pass it as a CONSTANT array

```
 void displayDblAry(const double theAry[], int arySize );
```

- Remember that the const keyword must be repeated in the implementation

```
void displayDblAry(const double theAry[], int arySize )
{
  for (int i = 0; i < arySize; i++)
        cout << theAry[i] << ' ';
  theAry[0] = 0;
}
```

This would be caught as a compiler error, because `theAry` has been passed as a **constant** array reference

# Rule: Use const parameters consistently for robust code!

- What is wrong with the following code?

```
   double computeAverage(const int a[], int numberUsed);
// double computeAverage(int a[], int numberUsed);
//Returns the average of the elements in the first numberUsed
//elements of the array a. The array a is unchanged.


void showDifference(const int a[], int numberUsed)
{
      double average = computeAverage(a, numberUsed);
      cout << "Average of the " << numberUsed << " numbers = "
           << average << endl << "The numbers are:\n";
      for (int index = 0; index < numberUsed; index++)
            cout << a[index] << " differs from average by "
                 << (a[index] - average) << endl;
}
```

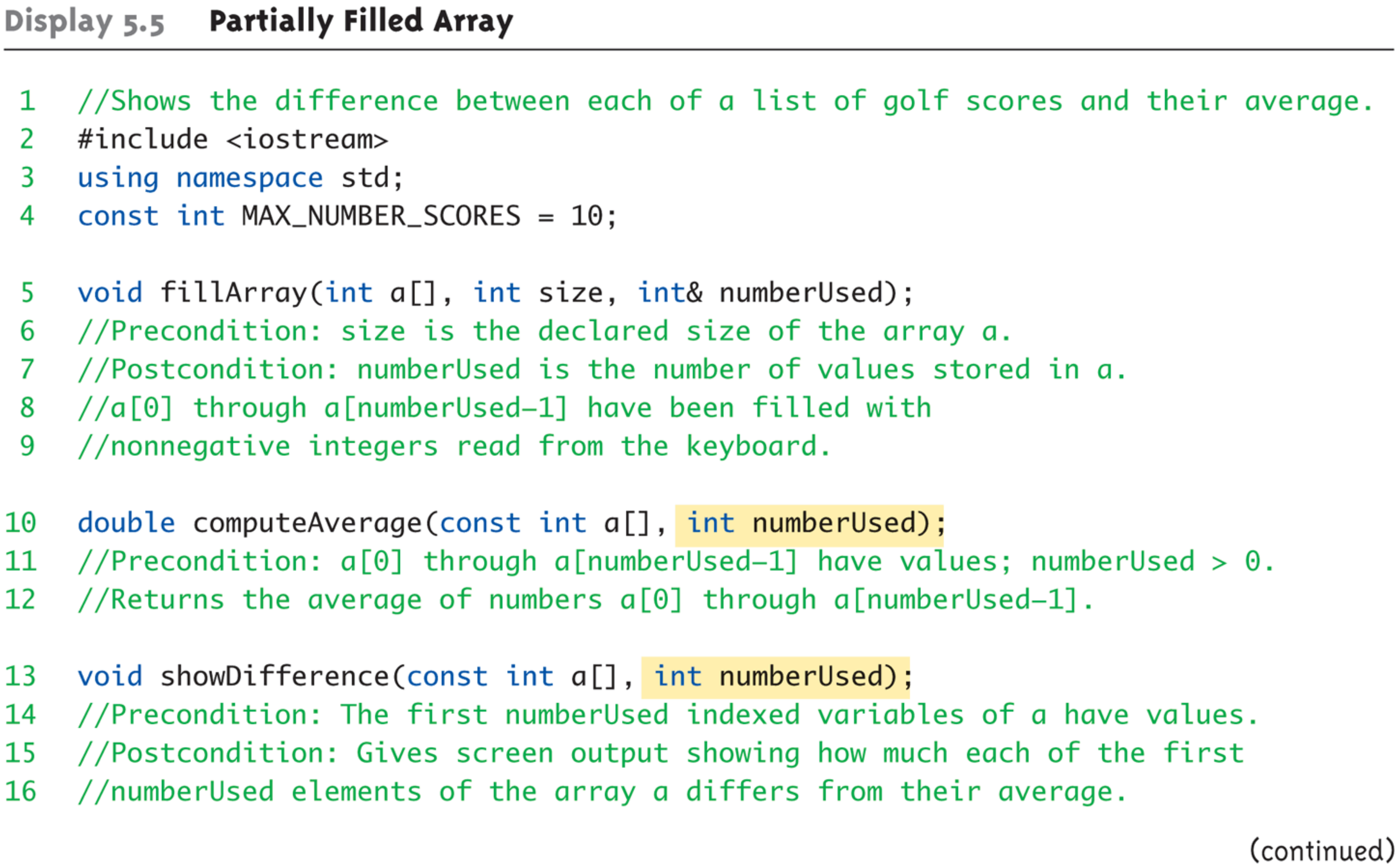
# Partially-filled arrays

- We must know ahead of time what size to specify for the array
  - It must be available at compile time
- But often difficult to know the exact array size needed
  - Maybe it will depend on the result of a calculation involving variables
  - Or maybe it will depend on user input
  - Or on how many entries are found in a file
- Solution: declare it to be largest size we might need
  - Some spaces in the array will probably not be needed
  - To keep "track" of valid data in array, an additional "tracking" variable needed, such as

```
int numberUsed;
```

Which tracks the number of *valid* elements in the array at any stage of execution

# Partially-filled Arrays Example: (1 of 3)



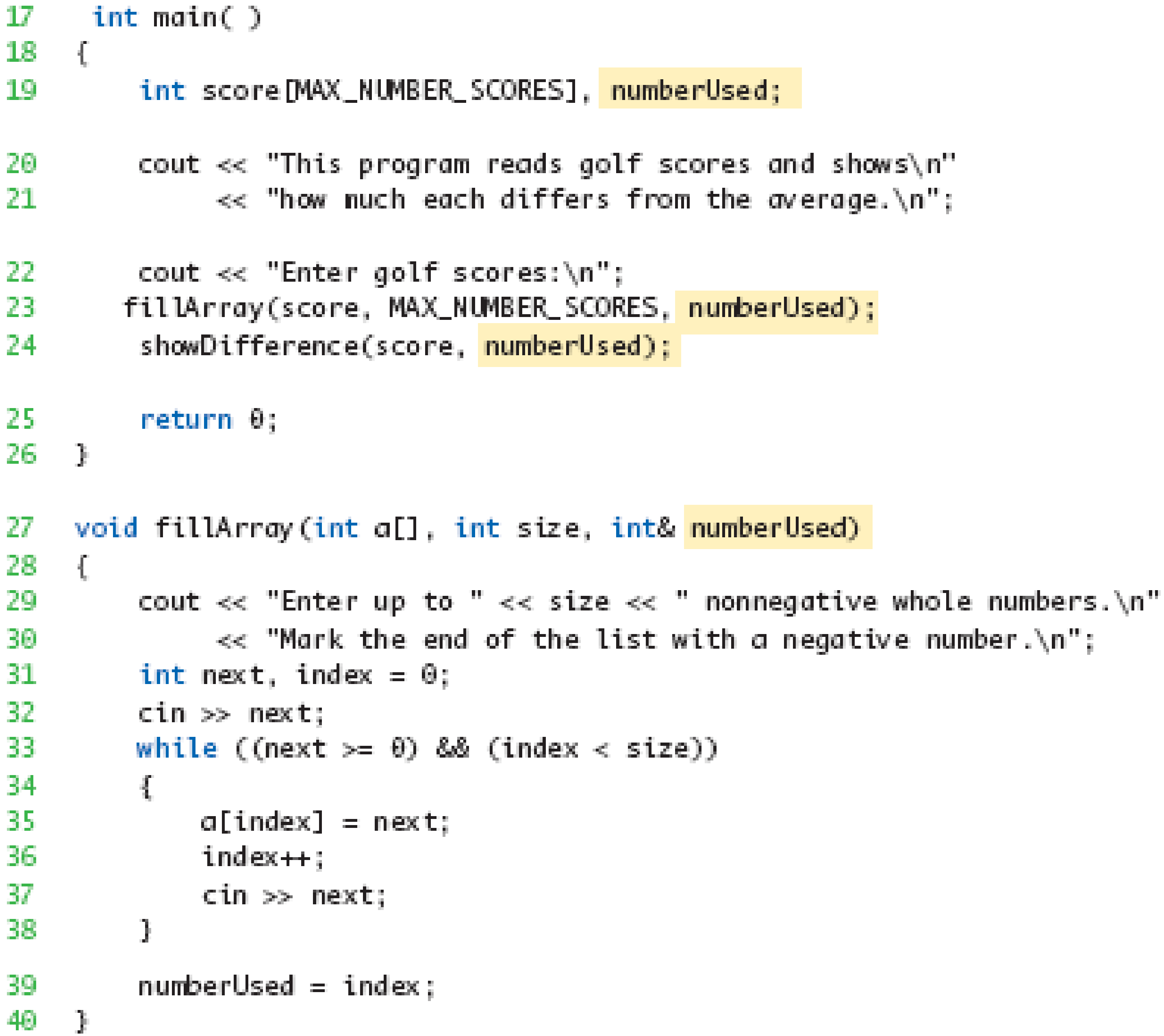
**Display 5.5 Partially Filled Array**

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

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

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

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

(continued)

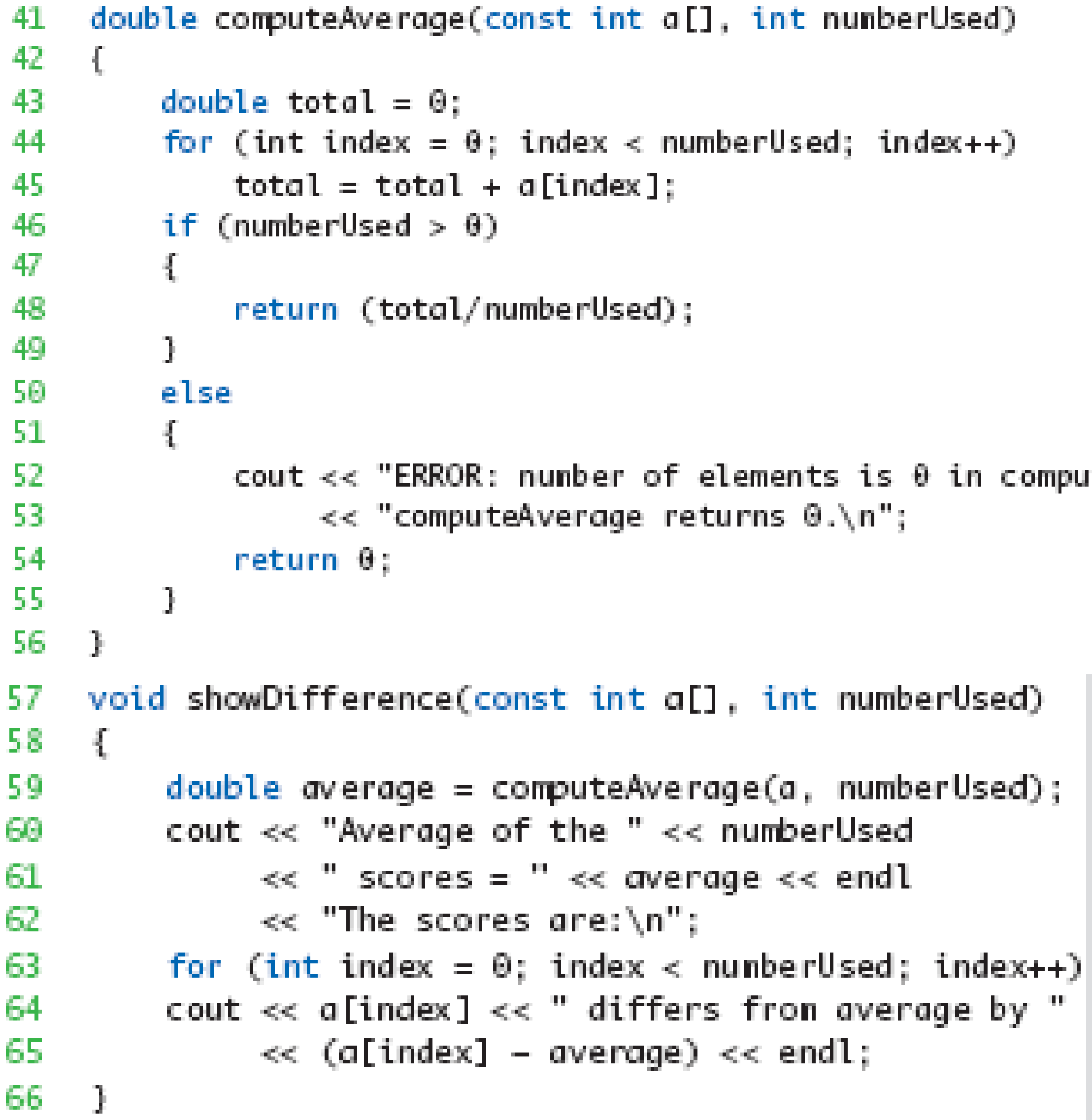
From Savitch chapter 5

# Partially-filled Arrays Example: (2 of 3)

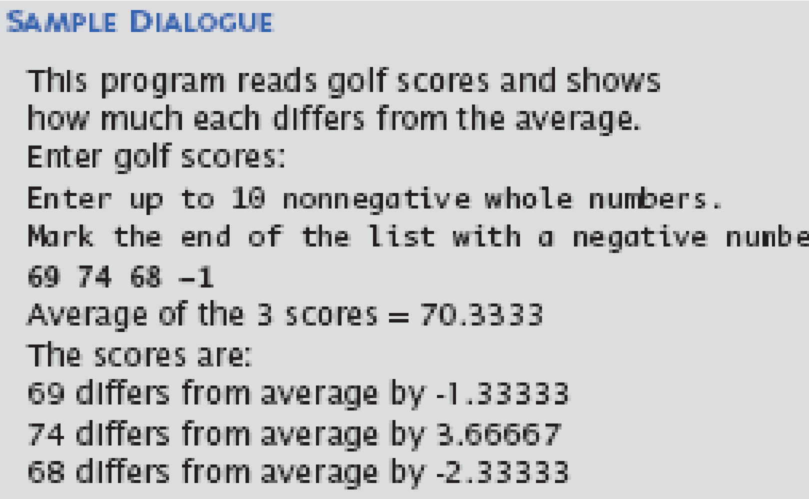


```
 int main( )
{
    int score[MAX_NUMBER_SCORES], numberUsed;

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

    cout << "Enter golf scores:\n";
    fillArray(score, MAX_NUMBER_SCORES, numberUsed);
    showDifference(score, numberUsed);

    return 0;
}

void fillArray(int a[], int size, int& numberUsed)
{
    cout << "Enter up to " << size << " nonnegative whole numbers.\n"
         << "Mark the end of the list with a negative number.\n";
    int next, index = 0;
    cin >> next;
    while ((next >= 0) && (index < size))
    {
        a[index] = next;
        index++;
        cin >> next;
    }

    numberUsed = index;
}
```

# Partially-filled Arrays Example: (3 of 3)



```
double computeAverage(const int a[], int numberUsed)
{
    double total = 0;
    for (int index = 0; index < numberUsed; index++)
        total = total + a[index];
    if (numberUsed > 0)
    {
        return (total/numberUsed);
    }
    else
    {
        cout << "ERROR: number of elements is 0 in computeAverage.\n"
             << "computeAverage returns 0.\n";
        return 0;
    }
}
void showDifference(const int a[], int numberUsed)
{
    double average = computeAverage(a, numberUsed);
    cout << "Average of the " << numberUsed
         << " scores = " << average << endl
         << "The scores are:\n";
    for (int index = 0; index < numberUsed; index++)
    cout << a[index] << " differs from average by "
         << (a[index] - average) << endl;
}
```



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 numbe
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
```

# 2-dimensional arrays

- A 2-dimensional array is considered by the compiler as a one dimensional array where each element in the array is a 1-dimensional array of a specified length in other words “AN ARRAY OF ARRAYS”
  - If we think of a 2-dim array as representing an arrangement of rows and columns, then each *row* represents another 1-dimensional array element

exampleAry

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 10 | 11 | 12 |
| 20 | 21 | 22 |
| 30 | 31 | 32 |

exampleAry is a 2-dimensional array with 4 rows and 3 columns

```
int exampleAry[4][3];
```

exampleAry is a 1-dimensional array of size 4 where each entry is an array of 3 elements

exampleAry[0]  is  {1, 2, 3}

exampleAry[3] is {30, 31, 32}

- A 2-dimensional array is set up as a contiguous bit of memory, where all the data elements of the array are stored in row-order
- The compiler can still find any element in the array by counting forward, but it has to know how many columns there are in each row to do this.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 10 | 11 | 12 | 20 | 21 | 22 | 30 | 31 | 32 |

# Passing a 2-dimensional array to a function

- Thinking of it as an ‘array of arrays’ helps us to understand.
- Still have to pass in the size of the array
  - that is the number of rows
- Also have to give a description of the base type
  - the type of each element of the array is an array of a specified length
  - where the length tells the number of columns in each row

```
void myArray(int[][3], int numRows);
```

  - So the rule is:

the number of rows as a second parameter, but the number of columns as part of the description of the base type

# Arrays of more dimensions

- We could have even more dimensions to an array
  - But after 3 it gets a bit hard to visualise what’s going on
  - We probably would need to be a pure mathematician to do it justice
- The programming rule is always the same:
  - The compiler doesn’t remember the first dimension, but all the others are part of the description of the base type.
  - To pass a multi-dimensional array to a function, the first dimension has to be given as a separate variable
  - All the others are specified as part of the base type being passed

```
void multiArrayFunction(double theArray[][6][3][2], int firstDim);

double mArray[5][6][3][2];
   …
multiArrayFunction(mArray, 5);
```

- We’ll probably not need more than a 3-dim array for any problem!

# Practise

- Write a program that will read up to 10 letters into an array, and write the letters back to the screen in reverse order.
- Add functions to your program, one to read in the letters, and one to write out in reverse order.
- Write code that will fill the array a (declared below) with numbers typed in at the keyboard. The numbers will be input 5 per line, although your solution need not depend on how the input is divided into lines.

  ```
  int a[4][5];
  ```
- Write a function definition for a void function called echo such that the following function call will echo the input in 3, and will echo it as 4 lines of 5 numbers per line.

  ```
  echo(a, 4);
  ```