Capitolo 4 - Arrays

Outline

4.1 Introduction

- Arrays
  - Structures of related data items
  - Static entity - same size throughout program
- A few types
  - C-like, pointer-based arrays
  - C++, arrays as objects

4.2 Arrays

- Array
  - Consecutive group of memory locations
  - Same name and type
- To refer to an element, specify
  - Array name and position number
- Format: *arrayname[position number]*
  - First element at position 0
  - **n** element array **c**:

```
c[ 0 ], c[ 1 ]…c[ n - 1 ]
```

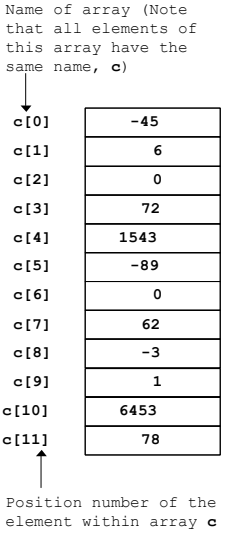
- Array elements are like normal variables

```
c[ 0 ] =  3;
cout << c[ 0 ];
```

- Performing operations in subscript. If **x = 3**,

```
c[ 5 - 2 ] == c[ 3 ] == c[ x ]
```


4.2 Arrays



4.3 Declaring Arrays

- Declaring arrays - specify:
  - Name
  - Type of array
  - Number of elements
  - Examples

```
int c[ 10 ];
float hi[ 3284 ];
```

- Declaring multiple arrays of same type
  - Similar format as other variables
  - Example

```
int b[ 100 ], x[ 27 ];
```


4.4 Examples Using Arrays

- Initializers

```
int n[ 5 ] = { 1, 2, 3, 4, 5 };
```

  - If not enough initializers, rightmost elements become 0
  - If too many initializers, a syntax error is generated

```
int n[ 5 ] = { 0 }
```

  - Sets all the elements to **0**
- If size omitted, the initializers determine it

```
int n[] = { 1, 2, 3, 4, 5 };
```

  - 5 initializers, therefore **n** is a 5 element array

```
1  // Fig. 4.4: fig04_04.cpp
2  // Initializing an array with a declaration
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 int main()
13 {
14    int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
15
16    cout << "Element" << setw( 13 ) << "Value" << endl;
17
18    for ( int i = 0; i < 10; i++ )
19       cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
20
21    return 0;
22 }
```

Notice how they array is declared and elements referenced.

```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```


Outline

1. Initialize array using a declaration
2. Define loop
3. Print out each array element

Program Output

```
1  // Fig. 4.7: fig04_07.cpp
2  // A const object must be initialized
3
4  int main()
5  {
6     const int x;  // Error: x must be initialized
7
8     x = 7;        // Error: cannot modify a const variable
9
10    return 0;
11 }
```

Notice that **`const`** variables must be initialized because they cannot be modified later.

```
Fig04_07.cpp:
Error E2304 Fig04_07.cpp 6: Constant variable 'x' must be
   initialized in function main()
Error E2024 Fig04_07.cpp 8: Cannot modify a const object in
   function main()
*** 2 errors in Compile ***
```


Outline

1. Initialize **`const`** variable
2. Attempt to modify variable

Program Output

4.4 Examples Using Arrays

- Strings
  - Arrays of characters
  - All strings end with **null** (**'\0'**)
  - Examples:

```
char string1[] = "hello";
char string1[] = { 'h', 'e', 'l', 'l', 'o',
                   '\0' };
```

  - Subscripting is the same as for a normal array

String1[0] is **'h'**

string1[2] is **'l'**

- Input from keyboard

```
char string2[ 10 ];
cin >> string2;
```

  - Takes user input
  - Side effect: if too much text entered, data written beyond array

Outline

1. Initialize strings

2. Print strings

2.1 Define loop

2.2 Print characters individually

2.3 Input string

3. Print string

```
1  // Fig. 4 12: fig04 12.cpp
2  // Treating character arrays as strings
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  int main()
10 {
11    char string1[ 20 ], string2[] = "string literal";
12
13    cout << "Enter a string: ";
14    cin >> string1;
15    cout << "string1 is: " << string1
16         << "\nstring2 is: " << string2
17         << "\nstring1 with spaces between characters is:\n";
18
19    for ( int i = 0; string1[ i ] != '\0'; i++ )
20       cout << string1[ i ] << ' ';
21
22    cin >> string1;  // reads "there"
23    cout << "\nstring1 is: " << string1 << endl;
24
25    cout << endl;
26    return 0;
27 }
```

Inputted strings are separated by whitespace characters. **"there"** stayed in the buffer.

Notice how string elements are referenced like arrays.

Program Output

```
Enter a string: Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
string1 is: there
```

4.5 Passing Arrays to Functions

- Specify the name without any brackets
  - To pass array **myArray** declared as

    ```
    int myArray[ 24 ];
    ```

    to function **myFunction**, a function call would resemble

    ```
    myFunction( myArray, 24 );
    ```

  - Array size is usually passed to function
- Arrays passed call-by-reference
  - Value of name of array is address of the first element
  - Function knows where the array is stored
    - Modifies original memory locations
- Individual array elements passed by call-by-value
  - pass subscripted name (i.e., **myArray[3]**) to function

4.5 Passing Arrays to Functions

- Function prototype:

  ```
  void modifyArray( int b[], int arraySize );
  ```

  - Parameter names optional in prototype
    - **int b[]** could be simply **int []**
    - **int arraysize** could be simply **int**

```
1  // Fig. 4.14: fig04_14.cpp
2  // Passing arrays and individual array elements to functions
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 void modifyArray( int [], int );  // appears strange
13 void modifyElement( int );
14
15 int main()
16 {
17    const int arraySize = 5;
18    int i, a[ arraySize ] = { 0, 1, 2, 3, 4 };
19
20    cout << "Effects of passing entire array call-by-reference:"
21         << "\n\nThe values of the original array are:\n";
22
23    for ( i = 0; i < arraySize; i++ )
24       cout << setw( 3 ) << a[ i ];
25
26    cout << endl;
27
28    // array a passed call-by-reference
29    modifyArray( a, arraySize );
30
31    cout << "The values of the modified array are:\n";
```

Functions can modify entire arrays. Individual array elements are not modified by default.

No parameter names in function prototype.

```
The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8
```

13

Outline

1. Define function to take in arrays

1.1 Initialize arrays

2. Modify the array (call by reference)

```
32
33    for ( i = 0; i < arraySize; i++ )
34       cout << setw( 3 ) << a[ i ];
35
36    cout << "\n\n\n"
37         << "Effects of passing array element call-by-value:"
38         << "\n\nThe value of a[3] is " << a[ 3 ] << '\n';
39
40    modifyElement( a[ 3 ] );
41
42    cout << "The value of a[3] is " << a[ 3 ] << endl;
43
44    return 0;
45 }
46
47 // In function modifyArray, "b" points to the original
48 // array "a" in memory.
49 void modifyArray( int b[], int sizeOfArray )
50 {
51    for ( int j = 0; j < sizeOfArray; j++ )
52       b[ j ] *= 2;
53 }
54
55 // In function modifyElement, "e" is a loca
56 // array element a[ 3 ] passed from main.
57 void modifyElement( int e )
58 {
59    cout << "Value in modifyElement is "
60         << ( e *= 2 ) << endl;
61 }
```

Parameter names required in function definition

```
Effects of passing array element call-by-value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```

14

Outline

2.1 Modify an element (call by value)

3. Print changes.

3.1 Function

Outline

Program Output

```
Effects of passing entire array call-by-reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8


Effects of passing array element call-by-value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[3] is 6
```


4.6 Sorting Arrays

- Sorting data
  - Important computing application
  - Virtually every organization must sort some data
    - Massive amounts must be sorted
- Bubble sort (sinking sort)
  - Several passes through the array
  - Successive pairs of elements are compared
    - If increasing order (or identical), no change
    - If decreasing order, elements exchanged
  - Repeat these steps for every element

Bubble Sort

Si effettuano una serie di passate sull'array fino a che esso non risulti ordinato.

Ad ogni passata, si confrontano coppie di elementi consecutivi e si scambiano di posto gli elementi che non verificano l'ordinamento.

La versione proposta dell'algoritmo può essere ottimizzata se si tiene conto del numero di scambi effettuati (o meno) in una passata (se durante una passata non sono stati effettuati scambi l'array è già ordinato e l'algoritmo può terminare).

Esercizio: realizzare una versione più efficiente di Bubble Sort.

17

Il programma riportato di seguito, segue il seguente schema

- legge l'array
- ordina l'array utilizzando l'algoritmo bubble sort
- stampa l'array ordinato

In particolare, ad ognuna delle istruzioni indicate corrisponde una funzione.

Inoltre, la funzione che si occupa dell'ordinamento dell'array utilizza un'altra funzione per effettuare lo scambio di posizione tra elementi consecutivi che non rispettano l'ordinamento.

18

4.6 Sorting Arrays

- Example:
  - Original: 3 4 2 6 7
  - Pass 1: 3 (2 4) 6 7
  - Pass 2: (2 3) 4 6 7
  - Small elements "bubble" to the top

```
#include <iostream>
#include <iomanip.h>
using std::cout;  // le seguenti istruzioni non sono necessarie perché già presenti in iomanip.h
using std::cin;
using std::endl;
using std::setw;
void bubbleSort(int [], int);
void scambia(int&, int&);
void leggiArray(int [], int);
void stampaArray(int[], int);

int main()
{       const int size = 20;
        int a[size];
        leggiArray(a, size);
        bubbleSort(a, size);
        stampaArray(a, size);
        return 0;
}

void leggiArray(int a[], int size)
{
        cout<<"Introduci gli elementi dell'array"<<endl;
        for (int i = 0; i < size; i++)
                cin>>a[i];
}
```

```
void bubbleSort(int a[], int size)
{
    for ( int pass = 1; pass < size; pass++ )
      for ( int j = 0; j < size - 1; j++ )
        if ( a[ j ] > a[ j + 1 ] )
          scambia(a[j], a[j+1]);
}

void scambia(int& a, int& b)
{
    int temp = a;
    a = b;
    b= temp;
}

void stampaArray(int a[], int size)
{
  for (int i = 0; i < size; i++)
  {
     if (i%20 == 0)
        cout<<endl;
        cout<<setw(4)<<a[i];
  }
}
```

21

22

4.7 Case Study: Computing Mean, Median and Mode Using Arrays

- Mean
  - Average
- Median
  - Number in middle of sorted list
  - 1, 2, 3, 4, 5 (3 is median)
- Mode
  - Number that occurs most often
  - 1, 1, 1, 2, 3, 3, 4, 5 (1 is mode)

```
1  // Fig. 4.17: fig04 17.cpp
2  // This program introduces the topic of survey data analysis.
3  // It computes the mean, median, and mode of the data.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8  using std::ios;
9
10 #include <iomanip>
11
12 using std::setw;
13 using std::setiosflags;
14 using std::setprecision;
15
16 void mean( const int [], int );
17 void median( int [], int );
18 void mode( int [], int [], int );
19 void bubbleSort( int[], int );
20 void printArray( const int[], int );
21
22 int main()
23 {
24    const int responseSize = 99;
25    int frequency[ 10 ] = { 0 },
26        response[ responseSize ] =
27           { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
28             7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
29             6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
30             7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
31             6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
32             7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
33             5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
```

Outline

23

1. Function prototypes

1.1 Initialize array

2. Call functions `mean`, `median`, and `mode`

```
34             7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
35             7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
36             4, 5, 6, 1, 6, 5, 7, 8, 7 };
37
38    mean( response, responseSize );
39    median( response, responseSize );
40    mode( frequency, response, responseSize );
41
42    return 0;
43 }
44
45 void mean( const int answer[], int arraySize )
46 {
47    int total = 0;
48
49    cout << "********\n  Mean\n********\n";
50
51    for ( int j = 0; j < arraySize; j++ )
52       total += answer[ j ];
53
54    cout << "The mean is the average value of the data\n"
55         << "items. The mean is equal to the total of\n"
56         << "all the data items divided by the number\n"
57         << "of data items (" << arraySize
58         << "). The mean value for\nthis run is: "
59         << total << " / " << arraySize << " = "
60         << setiosflags( ios::fixed | ios::showpoint )
61         << setprecision( 4 )
62         << static cast< double >( total ) / arraySize << "\n\n";
63 }
64
65 void median( int answer[], int size )
66 {
67    cout << "\n********\n Median\n********\n"
```

Outline

24

3. Define function `mean`

3.1 Define function `median`

```
68          << "The unsorted array of responses is";
69
70       printArray( answer, size );
71       bubbleSort( answer, size );
72       cout << "\n\nThe sorted array is";
73       printArray( answer, size );
74       cout << "\n\nThe median is element " << size / 2
75            << " of\nthe sorted " << size
76            << " element array.\nFor this run the median is "
77            << answer[ size / 2 ] << "\n\n";
78    }
79
80    void mode( int freq[], int answer[], int size )
81    {
82       int rating, largest = 0, modeValue = 0;
83
84       cout << "\n********\n  Mode\n********\n";
85
86       for ( rating = 1; rating <= 9; rating++ )
87          freq[ rating ] = 0;
88
89       for ( int j = 0; j < size; j++ )
90          ++freq[ answer[ j ] ];
91
92       cout << "Response"<< setw( 11 ) << "Frequency"
93            << setw( 19 ) << "Histogram\n\n" << setw( 55 )
94            << "1    1    2    2\n" << setw( 56 )
95            << "5    0    5    0    5\n\n";
```

Notice how the subscript in **`frequency[]`** is the value of an element in **`response[]`** (**`answer[]`**).

Outline

3.1 Define function `median`

3.1.1 Sort Array

3.1.2 Print middle element

3.2 Define function `mode`

3.2.1 Increase `frequency[]` depending on `response[]`

```
96
97       for ( rating = 1; rating <= 9; rating++ ) {
98          cout << setw( 8 ) << rating << setw( 11 )
99               << freq[ rating ] << "          ";
100
101         if ( freq[ rating ] > largest ) {
102            largest = freq[ rating ];
103            modeValue = rating;
104         }
105
106         for ( int h = 1; h <= freq[ rating ]; h++ )
107            cout << '*';
108
109         cout << '\n';
110      }
111
112      cout << "The mode is the most frequent value.\n"
113           << "For this run the mode is " << modeValue
114           << " which occurred " << largest << " times." << endl;
115   }
116
117   void printArray( const int a[], int size )
118   {
119      for ( int j = 0; j < size; j++ ) {
```

Print stars depending on value of **`frequency[]`**

Outline

3.3 Define `bubbleSort`

```
120
121      if ( j % 20 == 0 )
122         cout << endl;
123
124      cout << setw( 2 ) << a[ j ];
125   }
126 }
```

Outline

27

3.3 Define `bubbleSort`

3.3 Define `printArray`

```
********
  Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items (99). The mean value for
this run is: 681 / 99 = 6.8788


********
 Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7
```

Outline

28

4. Program Output

Outline

Program Output

```
********
  Mode
********
Response  Frequency        Histogram

                                   1    1    2    2
                              5    0    5    0    5

      1          1          *
      2          3          ***
      3          4          ****
      4          5          *****
      5          8          ********
      6          9          *********
      7         23          ***********************
      8         27          ***************************
      9         19          *******************
The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```


4.8 Searching Arrays: Linear Search and Binary Search

- Search array for a key value
- Linear search
  - Compare each element of array with key value
  - Useful for small and unsorted arrays
- Binary search
  - Can only be used on sorted arrays
  - Compares middle element with key
    - If equal, match found
    - If key < middle, repeat search through the first half of the array
    - If key > middle, repeat search through the last half of the array
  - Very fast; at most **n** steps, where 2^**n** > # of elements
    - 30 element array takes at most 5 steps
      - $2^5 > 30$

Ricerca lineare e binaria

```
#include <iostream>

using namespace std;

int ricercaLineare(int, int[], int);
int ricercaBinaria(int, int[], int);

int main ()
{
   int vector[10]= { 2 , 4 , 5, 11, 15, 23, 44, 50, 90, 700 };        int elem=90;
   int posizione=ricercaLineare(elem,vector,10);
   cout<<"L'elemento "<<elem<<" si trova in posizione"<<posizione<<endl;

   posizione=ricercaBinaria(elem,vector,10);
   cout<<"L'elemento "<<elem<<" si trova in posizione"<<posizione<<endl;

   return 0;
}
```


Ricerca lineare

```
int ricercaLineare(int elem, int vector[], int vectorSize)
{
   int nPassi=0; //per contare il numero di passi eseguiti dalla funzione
   bool trovato=false;
   int i=0;
   while(!trovato && i<vectorSize)
   {
        nPassi++;
        if(vector[i]==elem)
                trovato=true;
        i++;
   }
   cout<<"numero di passi eseguiti dalla ricerca lineare: "<<nPassi<<endl;
   if(trovato)
        return i-1;
   return -1;
}
```


Ricerca binaria

```
int ricercaBinaria(int elem, int vector[], int vectorSize)
{
   int nPassi=0; // per contare il numero di passi eseguiti dalla funzione
   int min=0;
   int max=vectorSize-1;
   int mezzo;
   bool trovato=false;
   while(!trovato && min<=max)
   {
        nPassi++;
        mezzo=(max+min)/2;
        if(vector[mezzo]==elem)
                trovato=true;
        if(vector[mezzo]<elem)
                min=mezzo+1;
        else
                max=mezzo-1;
   }
   cout<<"numero di passi eseguiti dalla ricerca binaria: "<<nPassi<<endl;
   if(trovato) return mezzo;
   return -1;
}
```


Confronto ricerca lineare e binaria

```
numero di passi eseguiti dalla ricerca lineare: 9
L'elemento 90 si trova in posizione 8
numero di passi eseguiti dalla ricerca binaria: 3
L'elemento 90 si trova in posizione 8
```

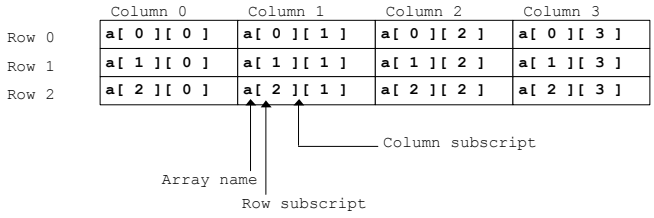
La ricerca binaria termina al più in **$\log_2$(dimensioneVettore)** passi
nel nostro caso $\log_2(10)$=**3**.

La ricerca lineare termina in al più in **dimensioneVettore** passi
nel nostro caso **10**.

Benché la ricerca binaria sia più efficiente di quella lineare non può essere utilizzata in tutti i casi. La ricerca binaria richiede, a differenza di quella lineare, che il vettore in input sia ordinato.

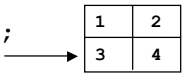
4.9 Multiple-Subscripted Arrays

- Multiple subscripts - tables with rows, columns
  - Like matrices: specify row, then column.



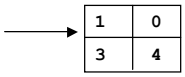
- Initialize

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```



  - Initializers grouped by row in braces

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```



4.9 Multiple-Subscripted Arrays

- Referenced like normal

```
cout << b[ 0 ][ 1 ];
```

  - Will output the value of **0**
  - Cannot reference with commas

```
cout << b( 0, 1 );
```

    - Will try to call function **b**, causing a syntax error

```
1  // Fig. 4.23: fig04 23.cpp
2  // Double-subscripted array example
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7  using std::ios;
8
9  #include <iomanip>
10
11 using std::setw;
12 using std::setiosflags;
13 using std::setprecision;
14
15 const int students = 3;   // number of students
16 const int exams = 4;      // number of exams
17
18 int minimum( int [][ exams ], int, int );
19 int maximum( int [][ exams ], int, int );
20 double average( int [], int );
21 void printArray( int [][ exams ], int, int );
22
23 int main()
24 {
25    int studentGrades[ students ][ exams ] =
26          { { 77, 68, 86, 73 },
27            { 96, 87, 89, 78 },
28            { 70, 90, 86, 81 } };
29
30    cout << "The array is:\n";
31    printArray( studentGrades, students, exams );
32    cout << "\n\nLowest grade: "
33         << minimum( studentGrades, students, exams )
```

Each row is a particular student, each column is the grades on the exam.

Outline

1. Initialize variables

1.1 Define functions to take double scripted arrays

1.2 Initialize `studentgrades[][]`

2. Call functions `minimum`, `maximum`, and `average`

```
34         << "\nHighest grade: "
35         << maximum( studentGrades, students, exams ) << '\n';
36
37    for ( int person = 0; person < students; person++ )
38       cout << "The average grade for student " << person << " is "
39            << setiosflags( ios::fixed | ios::showpoint )
40            << setprecision( 2 )
41            << average( studentGrades[ person ], exams ) << endl;
42
43    return 0;
44 }
45
46 // Find the minimum grade
47 int minimum( int grades[][ exams ], int pupils, int tests )
48 {
49    int lowGrade = 100;
50
51    for ( int i = 0; i < pupils; i++ )
52
53       for ( int j = 0; j < tests; j++ )
54
55          if ( grades[ i ][ j ] < lowGrade )
56             lowGrade = grades[ i ][ j ];
57
58    return lowGrade;
59 }
60
61 // Find the maximum grade
62 int maximum( int grades[][ exams ], int pupils, int tests )
63 {
64    int highGrade = 0;
65
66    for ( int i = 0; i < pupils; i++ )
```


Outline

2. Call functions `minimum`, `maximum`, and `average`

3. Define functions

```
67
68       for ( int j = 0; j < tests; j++ )
69
70          if ( grades[ i ][ j ] > highGrade )
71             highGrade = grades[ i ][ j ];
72
73    return highGrade;
74 }
75
76 // Determine the average grade for a particular student
77 double average( int setOfGrades[], int tests )
78 {
79    int total = 0;
80
81    for ( int i = 0; i < tests; i++ )
82       total += setOfGrades[ i ];
83
84    return static cast< double >( total ) / tests;
85 }
86
87 // Print the array
88 void printArray( int grades[][ exams ], int pupils, int tests )
89 {
90    cout << "                  [0]  [1]  [2]  [3]";
91
92    for ( int i = 0; i < pupils; i++ ) {
93       cout << "\nstudentGrades[" << i << "] ";
94
95       for ( int j = 0; j < tests; j++ )
96          cout << setiosflags( ios::left ) << setw( 5 )
97               << grades[ i ][ j ];
98    }
99 }
```

Outline

39

3. Define functions

```
The array is:
                  [0]  [1]  [2]  [3]
studentGrades[0] 77   68   86   73
studentGrades[1] 96   87   89   78
studentGrades[2] 70   90   86   81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 2 is 87.50
The average grade for student 2 is 81.75
```

Outline

40

Program Output

3.12 Recursion

- Recursive functions
  - Are functions that calls themselves
  - Can only solve a base case
  - If not base case, the function breaks the problem into a slightly smaller, slightly simpler, problem that resembles the original problem and
    - Launches a new copy of itself to work on the smaller problem, slowly converging towards the base case
    - Makes a call to itself inside the **`return`** statement
  - Eventually the base case gets solved and then that value works its way back up to solve the whole problem

3.12 Recursion

- Example: factorial

  $n! = n * (n – 1) * (n – 2) * ... * 1$

  - Recursive relationship ($n! = n * (n – 1)!$)

    $5! = 5 * 4!$

    $4! = 4 * 3!...$

  - Base case ($1! = 0! = 1$)

Example Using Recursion: The Factorial

C++ code for **`factorial`** function

- Recursive version

```
long factorial( long n )
 {
    if ( n == 0 || n == 1 )  // base case
       return 1;
    else return n*factorial(n-1);
 }
```


Example Using Recursion: The Factorial

- Iterative version

```
long factorial( long n )
 {
    long fact=1;
    int i;
    for (i=1; i<=n; i++)
    {
     fact=fact*i;
    }
    return fact;
 }
```

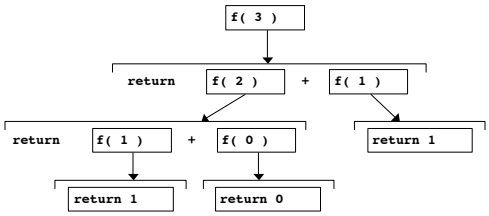

3.13 Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number sum of two previous ones
  - Example of a recursive formula:
    fib(n) = fib(n-1) + fib(n-2)
- C++ code for **fibonacci** function

```
long fibonacci( long n )
{
  if ( n == 0 || n == 1 )  // base case
  return n;
else return fibonacci( n - 1 ) +
  fibonacci( n – 2 );
}
```


3.13 Example Using Recursion: The Fibonacci Series

- Diagram of Fibonacci function



```
1  // Fig. 3.15: fig03_15.cpp
2  // Recursive fibonacci function
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  unsigned long fibonacci( unsigned long );
10
11 int main()
12 {
13     unsigned long result, number;
14
15     cout << "Enter an integer: ";
16     cin >> number;
17     result = fibonacci( number );
18     cout << "Fibonacci(" << number << ") = " << result << endl;
19     return 0;
20 }
21
22 // Recursive definition of function fibonacci
23 unsigned long fibonacci( unsigned long n )
24 {
25    if ( n == 0 || n == 1 )  // base case
26       return n;
27    else                     // recursive case
28       return fibonacci( n - 1 ) + fibonacci( n - 2 );
29 }
```


Outline

1. Function prototype

1.1 Initialize variables

2. Input an integer

2.1 Call function `fibonacci`

2.2 Output results.

3. Define `fibonacci` recursively

Only the base cases return values. All other cases call the **`fibonacci`** function again.

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```


Outline

Program Output

3.14 Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between performance (iteration) and good software engineering (recursion)

Binary Search: Recursive Version

```
int binarySearch(int v[], int elem, int from, int to)
{ int middle;
  if (from > to ) // elem is not present in v[]
      return -1;
  middle = (from+to)/2;
  if (v[middle]== elem) // elem is in v[middle]
      return middle;
  if (v[middle] > elem)  // search in the first half of the array
      return binarySearch(v,elem,from,middle-1);
  else  // search in the second half of the array
      return binarySearch(v,elem,middle+1,to);
}
```