

## UCLA Stat 130D Statistical Computing and Visualization in C++

**Instructor: Ivo Dinov**, Asst. Prof. in  
Statistics / Neurology

University of California, Los Angeles, Winter 2007  
[http://www.stat.ucla.edu/~dinov/courses\\_students.html](http://www.stat.ucla.edu/~dinov/courses_students.html)

## ARRAYS

- Introduction to Arrays
  - Declaring and Referencing Arrays
  - Arrays in Memory
  - Initializing Arrays
- Arrays in Functions
  - Indexed Variables as Function Arguments
  - Entire Arrays as Function Arguments
  - The `const` Parameter Modifier
- Programming with Arrays
  - Partially Filled Arrays
- Arrays and Classes
  - Arrays of Classes (arrays of objects)
  - Classes of arrays (classes having arrays as members)

2

## Introduction to Arrays

- The only **composite data types** we have dealt with are the **struct** and the **class**. It is cumbersome at best to write code so program can choose a **particular member** of a **struct** or **class**.
- **Arrays** solve this problem by allowing **random access via indexing**.
- To do this, arrays have two parts: the **name** and the **index**.
- For example:  
If **score** is the name of the array, then **score[0]**, **score[1]**, **score[2]**, **score[3]** and **score[4]** are five consecutive variables.
- To let the program decide which score variable to access, the program computes an `int` variable, say **ind**, which is used as an index for **score**.
- This code stores a score of 89 at a position in the array we have stored in `int` variable **ind**:  
`score[ind] = 89;`
- This fetches the value previously stored in the array at position **ind**:  
`y = score[ind];`

3

## Declaring and Referencing Arrays(1 of 2)

- In C++ an **array** consisting of 5 variables of type `int` is declared:  
`int score[5];`
- The value between the brackets in the definition is called the **declared size** of the array.
- Such a declaration creates 5 `int` type variables which are accessed as `score[0]`, `score[1]`, `score[2]`, `score[3]`, `score[4]`
- These are called **indexed variables**, also called **subscripted variables**.
- The number in the brackets is called an **index** or **subscript**.
- Array subscripts start at 0 and run to *one less than the declared size of the array*.
- The indexed variable type was `int`, but could be almost any type. All indexed variables in a particular array must have the same type.
- This type is called the **base type of the array**.
- Be careful not to confuse the **declared size** in a declaration of an array with the **index value** in an array reference.

```
int array_name[declared_size];  
array_name[index]; // 0 <= index < declared_size
```

## Declaring and Referencing Arrays(2 of 2)

- It is possible to declare arrays along with ordinary variables:  
`int next, score[5], max;`
- I encourage that you declare variables one per line:  
`int next;  
int score[5];  
int max;`
- Array references may be used anywhere an ordinary variable is used.  
`cin >> score[4] >> score[2];  
cout << score[2] << " " << score[4];  
score[0] = 32;  
score[3] = next;`

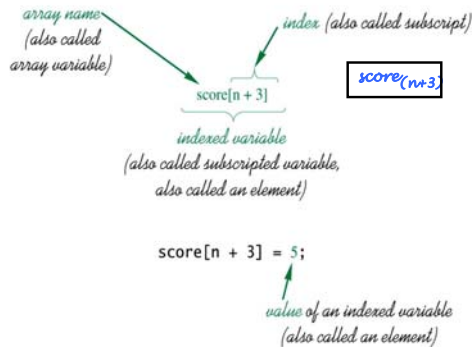
5

## 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;  
}
```

6

### Display 9.2 Array Terminology



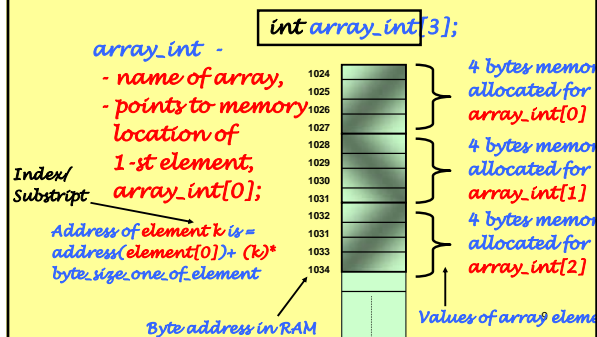
### Arrays in Memory

- A computer's memory is like a long linear list of numbered locations called bytes.
- The locations are called **addresses**, and the information written there is either some program's instructions or data.
- Every variable, whether the type is built-in or constructed by the programmer (**enum, class, struct, or array**), has a **location** where the variable starts and a **number of bytes** necessary to hold the variable that is determined by the type of the variable.
- Hence, every variable has an **address** and a **type**.
- An array variable is represented in this way, but here there is more. Consider:  

```
int a[6];
```
- Here the compiler decides where in memory to put the array, and the size of the array is  $6 * \text{the size of an int}$ .
- The size of an array is the **Declared\_Size \* sizeof(Array\_Type)**

8

### Memory Representation of Arrays



### Array Index Out of Range (1 of 2)

- The most common programming error when using arrays is the attempt to reference a non-existent array index.
- The array definition  

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

declares **only** the 6 indexed variables `a[0]`, `a[1]`, ..., `a[5]`.
- An index value outside 0 to 5 is an **out of range** error, i.e., **illegal**.
- Consider:  

```
a[7] = 248;
```

C++ treats 7 as if it were legal subscript, and attempts to write to memory where `a[7]` would be!!!! **Serious memory violation**.

10

### Array Index Out of Range (2 of 2)

- Would `a[7]` be written in two chunks of memory the size of an `int` beyond the real end of the array.
- Unfortunately, **compilers do not detect this error**. Your program may seem to work correctly. The effects range from no detectable effects, to abnormal end of program, to operating system crashing, to the next application crashing obscurely when it is started.
- C++ arrays were designed to be a **low level** construct that is normally no array bounds are checked.
- If your application needs the protection, you can write a class to provide bounds checking (sacrificing speed).

11

### Initializing Arrays

- A variable of simple type can hold only one value in the variable
- A **simple variable** can be initialized in one statement:  

```
int sum = 0;
```
- A variable of **array, struct, or class** type is a **compound variable**.
- A **struct** can be **initialized** with one statement:  

```
struct Struct1 { int a; int b;};  
Struct1 x = { 1, 2};
```
- An **array** can be **initialized** like we initialize a struct:  

```
int children[3] = { 2, 12, 1};
```
- The compiler will count for you:  

```
int b[] = { 5, 12, 11};
```
- This is equivalent to  

```
int b[3] = { 5, 12, 11};
```

12

## Arrays in Functions: Indexed Variables as Function Arguments

- An indexed variable is just a variable whose type is the base type of the array, and may be used anywhere any other variable whose type is the base type of the array might be used.

```
int i, n, a[10];
my_function(n);
my_function(a[3]);
```

13

### Indexed Variables as an Argument

```
// Illustrates the use of an indexed variable as an argument. CALL-BY-VALUE
// 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); }
```

14

## Programming Tip

### Use defined constant for the Size of an Array

- Array indices always start at 0 and end with, n-1, a value one less than the size of the array.
- Consider changing a program where the size of an array (say 50) appears several dozen times over thousands of lines.
- You might be looking for 49, 50 or perhaps 51. You must understand every instance where a number in this range was used. You might have to find where 25 (= 50/2) was used. You can't be certain you have completed the job.
- In order to write code that is easily and correctly modifiable, you should use a defined constant for the array size.
- Define a constant SIZE, and use that. Then you have only one point where you need to change the size:
- const int SIZE = 50;

15

## Entire Arrays as Function Arguments (1 of 3)- Call-by-value? Call-by-reference? Or call-by-array?

- It is possible to use an entire array as a formal parameter for a function.
- Remember a formal parameter is a kind of place holder that is filled in by the argument at the time the function is called. Arguments are placed in parentheses after the function name to signal that the function is being called and that this is the list of arguments to be "plugged in" for the parameter.
- A function can have a formal parameter for an entire array that is neither a call-by-value nor a call-by-reference parameter.
- This is a new parameter type called an array parameter.
- This is not call-by-value because we can change the array when passed this way.
- This is not call-by-reference because we cannot change the complete array by a single assignment when passed this way.<sup>46</sup>

## Entire Arrays as Function Arguments(2 of 3)

- While this is not call-by-reference, the behavior is like call-by-reference since we can change individual array elements by assigning them in the called function.
  - The function fill\_up that has an array parameter.
  - Here is an example call to fill\_up:
- ```
int score[5];
const int number_of_scores = 5;
fill_up(score, number_of_scores);
```
- This call is equivalent to
- ```
size = 5; // 5 is the value of number_of_scores
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;

17

## Entire Arrays as Function Arguments(3 of 3)

- When an array is used as an argument in a function call, any action that is performed on the array parameter is performed on the array argument, so the values of the indexed variables of the array argument can be changed by the function.
- On declaration, C++ reserves enough memory to hold the indexed variables that make up the array.
- C++ does not remember the addresses of the indexed variables, but it knows how to find them given the array address and the index.
- If score[3] is needed, C++ knows that score[3] is 3 int variables past score[0]. To get score[3], C++ adds 3 \* sizeof(int) + add\_score[0].
- Viewed this way, an array is three things:
  - an address of the start of the array in memory
  - a type, which tells how big each indexed variable is
  - the array size, which tells number of indexed variables.
- The array argument tells the caller only the address and type, but not the size of the array. Hence, the array-function-calls are NOT call by reference

18

### Function with an Array Parameter

```
//      Function Prototype:
void fill_up(int a[ ], int size);
//      Precondition: size is the declared size of the array a.
//      The user will type in "size" number of integers.
//      Postcondition: The array a is filled with "size" integers
//      from the keyboard.
//      Function Definition:
//      Uses iostream.h:
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;
}
```

19

### The `const` Parameter Modifier (1 of 2)

- Array parameters allow the function to change any value stored in the array.
- Frequently, this is the intent of the function.
- Sometimes this is definitely not the case. Sometimes we want to avoid changing an array parameter.

Example: *const prevents inadvertent changing of array a*

```
void show_the_world( const int a[ ], int size_of_a)
{
    cout << "Array values are:\n";
    for ( int i = 0; i < size_of_a; i++)
        cout << a[i] << " ";
    cout << endl;
}
```

20

## Programming with Arrays

### Case Study Production Graph (1 of 10)

- Design a program to display a bar graph showing the production of each of 4 plants during any given week.
- Each plant keeps separate production figures for each of its departments. The various plants have different collections of departments.
- Input is entered plant-by-plant, consisting of a list of numbers specifying the production for each department in that plant.
- Output Form (Production Volume):

```
Plant #1 *****
Plant #2 *****
Plant #3 *****
Plant #4 *****
```

Each asterisk (\*) represents 1000 units output.

21

### A Production Graph (2 of 10)

A precise specification of Input and output:

- Input:
  - Plants are numbered 1, 2, 3, 4
  - Department production is a list terminated with a negative *sentinel*.
- Output:
  - A bar graph showing total production volume in each of the 4 plants.
  - Each asterisk in the graph represents 1000 units. Production is rounded to the nearest 1000 units.

22

### A Production Graph (3 of 10)

#### Design & Analysis of the Problem

- An array named `production` will hold the plant-by-plant totals.
- There are 4 plants. The variable `plant_number` holds the total number of plants, which could change.  
*plant\_number is good variable name because the name communicates meaning, removing need for many comments.*
- The array will be indexed in `[0; plant_number - 1]`. The output for plant `n` will be placed in `production[n - 1]`.
- Output is in terms of 1000s of units, hence we scale array elements.
  - If input is 4040 for plant 3, this value is scaled to 4.
  - Indexed variable `production[2]` is set to 4.
  - Four asterisks are printed on the output graph for Plant #3.

23

### A Production Graph (4 of 10)

#### Design & Analysis of the Problem

- Divide the program into subtasks:
  - 1. Input\_data
  - 2. Scale
  - 3. graph
- 1. Input\_data
  - Read data for each plant, `n`
  - Set the indexed variable `production[n-1]` to the sum of the plant's departmental production numbers.
- 2. Scale:
  - For each plant, `n`, round the values in `production[n-1]`
- 3. Graph:
  - For each plant, `n`, output the bar graph.

24

### Prototypes for the main function of the Graph Program (1 of 2)

```
// Only the prototypes and prototype comments are listed here. The main function
// is the next slide and the complete program is in Display 09-09.cxx.

// Program reads data and displays a bar graph showing productivity for each plant.

#include <iostream>
const int NUMBER_OF_PLANTS = 4;

void input_data(int a[], int last_plant_number);
// Precondition: last_plant_number is the declared size of the array a.
// Postcondition: For plant_number = 1 through last_plant_number:
//   a[plant_number-1] equals the total production for plant number plant_number.

void scale(int a[], int size);
// Precondition: a[0] through a[size-1] each have a nonnegative value.
// Postcondition: a[i] has been changed to the number of 1000s (rounded to
// an integer) that were originally in a[i], for all i such that 0 <= i <= size-1.

void graph(const int asterisk_count[], int last_plant_number);
// Precondition: asterisk_count[0] through asterisk_count[last_plant_number-1]
// have nonnegative values.
// Postcondition: A bar graph has been displayed saying that plant
// number N has produced asterisk_count[N-1] 1000s of units, for each N
// such that 1 <= N <= last_plant_number
```

25

### Prototypes for the main function of the Graph Program (2 of 2)

```
// This is incomplete. The prototype comments have been omitted so that the main
// function can be seen along with the prototypes.
// Reads data and displays a bar graph showing productivity for each plant.
#include <iostream>
const int NUMBER_OF_PLANTS = 4;
void input_data(int a[], int last_plant_number);
void scale(int a[], int size);
void graph(const int asterisk_count[], int last_plant_number);

int main()
{
    using namespace std;
    int production[NUMBER_OF_PLANTS];

    cout << "This program displays a graph showing\n"
        << "production for each plant in the company.\n";

    input_data(production, NUMBER_OF_PLANTS);
    scale(production, NUMBER_OF_PLANTS);
    graph(production, NUMBER_OF_PLANTS);

    return 0;
}
```

26

## A Production Graph (5 of 10) Algorithm Design for input\_data

- The function prototype and prototype comments:

```
void input_data(int a[], int last_plant_number);
// Precondition: last_plant_number is the declared
// size of the array a.
// Postcondition: For plant_number = 1, 2, ..., last_plant_number:
//   a[plant_number-1] equals the total production for plant
// number plant_number.
```

- The algorithm:

For each plant number starting at 1 through last\_plant\_number  
do the following:

Read in all data for plant numbered plant\_number.  
Sum these numbers.  
Set production[plant\_number - 1] to this total.

27

## A Production Graph (6 of 10) Coding for input\_data

```
void get_total(int& sum);
// Reads nonnegative integers from the keyboard and places their total in
// sum.

//Uses iostream:
void input_data(int a[], int last_plant_number)
{
    using namespace std;
    for (int plant_number = 1;
         plant_number <= last_plant_number; plant_number++)
    {
        cout << endl
            << "Enter production data for plant number " << plant_number <<
            endl;
        get_total(a[plant_number - 1]);
    }
}
```

- The bulk of the work is done by function get\_total.
- get\_total reads production figures for the plant, sums them, and stores them in the indexed variable for that plant. It only needs an ordinary pass by reference parameter to send back the sum.

28

### Test of Function input\_data (2 of 4)

```
// Tests the function input_data. Prototype comments are omitted.
#include <iostream>
const int NUMBER_OF_PLANTS = 4;
void input_data(int a[], int last_plant_number);
void get_total(int& sum);

int main()
{
    using namespace std;
    int production[NUMBER_OF_PLANTS];
    char ans;
    do
    {
        input_data(production, NUMBER_OF_PLANTS);
        cout << endl
            << "Total production for each"
            << " of plants 1 through 4:\n";
        for (int number = 1; number <= NUMBER_OF_PLANTS; number++)
            cout << production[number - 1] << " ";
        cout << endl
            << "Test Again?(Type y or n and return): ";
        cin >> ans;
    } while ( (ans != 'N') && (ans != 'n') );
    cout << endl;
    return 0;
}
```

29

### Test of Function input\_data (3 of 4)

```
// Uses iostream:
void input_data(int a[], int last_plant_number)
{
    using namespace std;
    for (int plant_number = 1; plant_number <= last_plant_number; plant_number++)
    {
        cout << endl
            << "Enter production data for plant number "
            << plant_number << endl;
        get_total(a[plant_number - 1]);
    }
}

// Uses iostream:
void get_total(int& sum)
{
    using namespace std;
    cout << "Enter number of units produced by each department.\n";
    << "When finished - Append a negative number to the end of the list.\n";
    sum = 0;
    int next;
    cin >> next;
    while (next >= 0)
    {
        sum = sum + next;
        cin >> next;
    }
    cout << "Total = " << sum << endl;
}
```

30

#### Sample Dialog Testing Function `input_data` (4 of 4)

```
Enter production data for plant number 1
Enter number of units produced by each department.
Append a negative number to the end of the list.
1 2 3 -1
Total = 6

Enter production data for plant number 2
Enter number of units produced by each department.
Append a negative number to the end of the list.
0 2 3 -1
Total = 5

Enter production data for plant number 3
Enter number of units produced by each department.
Append a negative number to the end of the list.
2 -1
Total = 2

Enter production data for plant number 4
Enter number of units produced by each department.
Append a negative number to the end of the list.
-1
Total = 0

Total production for each of plants 1 through 4:
6 5 2 0
Test Again?(Type y or n and return): n
```

31

#### A Production Graph (8 of 10) Coding for `scale`

- The algorithm for `scale` divides each indexed variable from the array in the parameter list by 1000 then calls the function `round`, then assigns this back to the indexed variable.
- Code:

```
void scale(int a[], int size)
{
    for (int index = 0; index < size; index++)
        a[index] = round((float)a[index] / 1000.0);
}
```
- We used `float` division then rounding because `int` division drops the fractional part, it does not round:  
 $3/2$  gives 1, not 1.5
- Round adds 0.5 then uses library function `floor` to carry out the rounding.

32

#### The Function `scale` (1 of 2)

```
// Driver program for the function scale.
#include <iostream>
#include <cmath>

void scale(int a[], int size);
// Precondition: a[0] through a[size-1] each has a nonnegative value.
// Postcondition: a[i] has been changed to the number of 1000s (rounded to
// an integer) that were originally in a[i], for all i such that 0 <= i <= size-1.

int round(double number);
// Precondition: number >= 0.
// Returns number rounded to the nearest integer.

int main()
{
    using namespace std;
    int some_array[4], index;

    cout << "Enter 4 numbers to scale: ";
    for (index = 0; index < 4; index++)
        cin >> some_array[index];
    scale(some_array, 4);

    cout << "Values scaled to the number of 1000s are: ";
    for (index = 0; index < 4; index++)
        cout << some_array[index] << " ";
    cout << endl;
    return 0;
}
```

33

#### The Function `scale` (2 of 2)

```
// Function Prototypes and header definitions

void scale(int a[], int size);
// Precondition: a[0] through a[size-1] each has a nonnegative value.
// Postcondition: a[i] has been changed to the number of 1000s (rounded to
// an integer) that were originally in a[i], for all i such that 0 <= i <= size-1.

void scale(int a[], int size)
{
    for (int index = 0; index < size; index++)
        { a[index] = round(a[index]/1000.0); }
}

int round(double number);
// Precondition: number >= 0.
// Postcondition: Returns number rounded to the nearest integer.
// Uses cmath:

int round(double number)
{
    using namespace std;
    return floor(number + 0.5);
}
```

34

#### A Production Graph (10 of 10) Partially Filled Arrays

- The [size of the array that is needed may not be known at program writing time](#), or the size may vary from one run to another.
- One common and easy way to handle this is to [declare the largest array](#) that could ever be needed.
- The program uses as much of the array as needed.
- Access requires care not to access array positions that have not been assigned a value.
- This program uses array elements for index values 0 to `number_used - 1`. The details are similar to what they would be if exactly the required number of elements were allocated.
- One can also use memory allocation (`malloc`) to define arrays of diff size on the fly.

35

#### Production Graph Program (1 of 5)

```
#include <iostream>
#include <cmath>
const int NUMBER_OF_PLANTS = 4;

void input_data(int a[], int last_plant_number);
// Precondition: last_plant_number is the declared size of the array a.
// Postcondition: For plant_number = 1 through last_plant_number:
// a[plant_number-1] equals the total production for plant number plant_number.

void scale(int a[], int size);
// Precondition: a[0] through a[size-1] each have a nonnegative value.
// Postcondition: a[i] has been changed to the number of 1000s (rounded to
// an integer) that were originally in a[i], for all i such that 0 <= i <= size-1.

void graph(const int asterisk_count[], int last_plant_number);
// Precondition: asterisk_count[0] through asterisk_count[last_plant_number-1]
// have nonnegative values.
// Postcondition: A bar graph has been displayed saying that plant number N
// has produced asterisk_count[N-1] 1000s of units, for each N such that
// 1 <= N <= last_plant_number

void get_total(int& sum);
// Reads nonnegative integers from the keyboard and places their total in sum.

int round(double number);
// Precondition: number >= 0.
// Returns number rounded to the nearest integer.

void print_asterisks(int n);
// Prints n asterisks to the screen.
```

36

### Production Graph Program (2 of 5)

```
// Prototype comments omitted so main and prototypes can be displayed together.
#include <iostream>
#include <cmath>
const int NUMBER_OF_PLANTS = 4;
void input_data(int a[], int last_plant_number);
void scale(int a[], int size);
void graph(const int asterisk_count[], int last_plant_number);
void get_total(int& sum);
int round(double number);
void print_asterisks(int n);
int main ()
{
    using namespace std;
    int production[NUMBER_OF_PLANTS];
    cout << "This program displays a graph showing\n"
         << "production for each plant in the company.\n";
    input_data(production, NUMBER_OF_PLANTS);
    scale(production, NUMBER_OF_PLANTS);
    graph(production, NUMBER_OF_PLANTS);
    return 0;
}
```

37

### Production Graph Program (3 of 5)

```
void input_data(int a[], int last_plant_number)
{
    using namespace std;
    for (int plant_number = 1; plant_number <= last_plant_number; plant_number++)
    {
        cout << endl
             << "Enter production data for plant number "
             << plant_number << endl;
        get_total(a[plant_number - 1]);
    }
}

void get_total(int& sum)
{
    using namespace std;
    cout << "Enter number of units produced by each department.\n"
         << "Append a negative number to the end of the list.\n";
    sum = 0;
    int next;
    cin >> next;
    while (next >= 0)
    {
        sum = sum + next;
        cin >> next;
    }
    cout << "Total = " << sum << endl;
}
```

38

### Production Graph Program (4 of 5)

```
void scale(int a[], int size)
{
    for (int index = 0; index < size; index++)
        a[index] = round(a[index]/1000.0);
}

int round(double number)
{
    using namespace std;
    return floor(number + 0.5); //floor is from cmath:
}

// Uses iostream:
void graph(const int asterisk_count[], int last_plant_number)
{
    using namespace std;
    cout << "\nUnits produced in thousands of units:\n";
    for (int plant_number = 1;
         plant_number <= last_plant_number; plant_number++)
    {
        cout << "Plant #" << plant_number << " ";
        print_asterisks(asterisk_count[plant_number - 1]);
        cout << endl;
    }
}
```

39

### Production Graph Program (5 of 5)

```
// Uses iostream:
void print_asterisks(int n)
// more general prototype definition:
// void print_asterisks(ostream fout=cout, int n)
{
    using namespace std;
    for (int count = 1; count <= n; count++)
        cout << "*";
}
```

40

## PROGRAMMING EXAMPLE: **Sorting** an Array

- **Sorting an array** means rearranging data in a data structure to obey some **order** relation. An array that contains objects that can be compared is sorted if for every pair of indices **i** and **j**, if **i < j** then **array[i] <= array[j]**
- In other words:  
 $a[0] \leq a[1] \leq a[2] \leq \dots \leq a[\text{number\_used} - 1]$
- We illustrate sorting using the **selection sort** algorithm.
- The array is partially filled so we must pass an additional array parameter that specifies how many array elements are used.

41

## *Why sorting arrays? How to sort arrays?*

```
int myArray[] = {12, 0, 23, -19, 79, 34, 109, 23, 1, 15, -32, -45, 108, 17};
```

- monotone sequence:  
 $\{-45, -32, -19, 0, 1, 12, 15, 17, 23, 23, 34, 79, 108, 109\}$
- median (15 or 17, middle number)
- mode (most frequently used number: 23)
- ordered sequences are easier to interpret.
- sometimes throwing out the **outliers** (smallest, largest elements) is needed.
- Efficiency of sorting (minimizing computation time), best algorithms could sort an array of size **N** in  **$N^2 \log(N)$**  operations. This is only 3,000 operations for array of size 1,000.

42



## Array Sorting

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

## Sorting an Array (1 of 5)

```
// 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. References 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].
```

44

## Sorting an Array (2 of 5)

```
// Tests the procedure sort.
#include <iostream>
void fill_array(int a[], int size, int& number_used);
void sort(int a[], int number_used);
void swap_values(int& v1, int& v2);
int index_of_smallest(const int a[], int start_index, int number_used);
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;
}
```

45

## Sorting an Array (3 of 5)

```
// 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";

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

46

## Sorting an Array (4 of 5)

```
void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    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);
        if (a[index] > a[index_of_next_smallest])
            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;
    }
}
```

47

## Sorting an Array (5 of 5)

```
int index_of_smallest(const int a[], int start_index, int number_used)
{
    // Finds the index of the smallest array element from the
    // remaining [start_index + 1; number_used] elements
    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;
}
```

48



## Arrays of Classes

- An **array** with **struct** elements:

```
struct WindInfo
{
    double velocity; // in miles per hour
    char direction; // 'N', 'S', 'E', 'W'
};
WindInfo data_point[10];
```

- To fill the arrays `data_point`:

```
for (int i = 0; i < 10; i++)
{
    cout<<"Enter velocity for data point numbered: "<< i;
    cin >> data_point[i].velocity;
    cout<<"Enter direction for data point numbered: "<< i;
    cin >> data_point[i].direction;
}
```

49

## Arrays of Classes

- The way to read an expression like this *left to right* and *very carefully*.

```
data_point[i].direction
```

*an array having  
base type struct*

*an indexed variable of type struct*

*a member of the struct*

- The type of this expression is the type of the `direction` member of the struct that is the base type of the array `data_point`. In this case, the type is `char`.

50

### Header File for the class Money (1 of 4)

```
// File: money.h INTERFACE for the class Money.
// Values of this type are amounts of money in U.S. currency.
#ifndef MONEY_H
#define MONEY_H
#include <iostream>
using namespace std;
namespace money
{
    class Money
    {
    public:
        // Friend operator functions:
        friend Money operator +(const Money& amount1, const Money& amount2);
        // Returns the sum of the values of amount 1 and amount2.
        friend Money operator -(const Money& amount1, const Money& amount2);
        // Returns amount 1 minus amount2.
        friend Money operator -(const Money& amount);
        // Returns the negative of the value of amount.
        friend bool operator ==(const Money& amount1, const Money& amount2);
        // Returns true if amount1 and amount2 have the same value, false otherwise.
        friend bool operator < (const Money& amount1, const Money& amount2);
        // Returns true if amount1 is less than amount2, false otherwise.
    };
};
```

51

### Header File for the class Money (2 of 4)

```
// File: money.h INTERFACE for the class Money.
// class Money friend functions for i/o

friend istream& operator >>(istream& ins, Money& amount);
// Overloads the >> operator so it can be used to input values of
// type Money. For inputting negative amounts is as in -$100.00.
// Precondition: If ins is a file input stream, then ins has already
// been connected to a file.

friend ostream& operator <<(ostream& outs, const Money& amount);
// Overloads the << operator so it can be used to output values of
// type Money. Precedes each output value of type Money
// with a dollar sign.
// Precondition: If outs is a file output stream, then outs has
// already been connected to a file.
```

52

### Header File for the class Money (3 of 4)

```
// File: money.h INTERFACE for the class Money.
// class Money Constructors:

Money(long dollars, int cents);
// Initializes the object so its value represents an amount with
// the dollars and cents given by the arguments. If the amount
// is negative, then both dollars and cents should be negative.

Money(long dollars);
// Initializes the object so its value represents $dollars.00.

Money();
// Initializes the object so its value represents $0.00.

double get_value() const;
// Returns the amount of money recorded in the data portion of the calling
// object.
```

53

### Header File for the class Money (4 of 4)

```
// File: money.h INTERFACE for the class Money.
// class Money: Private members and wrap up

private:
    long all_cents;
};

// money

#endif //MONEY_H
```

54

## Arrays as Class Members

- A structure or a class can have an array as a member.
- If you have to keep track of practice times for swimming a particular distance, you might use this struct

```
struct Data
{
    double time[10];
    int distance;
};
```

- You can use this to set the distance:  
`my_best.distance = 20;`
- You can use this to set the time array:  
`cout<< "Enter ten times in seconds: \n";`  
`for (int i = 0; i < 10; i++)`  
 `cin >> my_best.time[i];`

55

## Program Using an Array of Objects (1 of 2)

```
// Reads in 5 amounts of money and shows how much each
// amount differs from the largest amount.
#include <iostream>
#include "money.h"

int main()
{
    using namespace std;
    using namespace savitchmoney;
    Money amount[5], max;
    int i;

    cout << "Enter 5 amounts of money:\n";
    cin >> amount[0];
    max = amount[0];

    for (i = 1; i < 5; i++)
    {
        cin >> amount[i];
        if (max < amount[i])
            max = amount[i];
        //max is the largest of amount[0],..., amount[i].
    }
}
```

56

## Program Using an Array of Objects (2 of 2)

```
Money difference[5];
for (i = 0; i < 5; i++)
    difference[i] = max - amount[i];
cout << "The highest amount is " << max << endl;
cout << "The amounts and their differences from the largest are:\n";
for (i = 0; i < 5; i++)
{
    cout << amount[i] << " off by "
        << difference[i] << endl;
}

return 0;
}
```

57

## Interface for a Class with an Array Member (1 of 2)

```
// FILE: templist.h. This is the INTERFACE for the class TemperatureList.
// Values of this type are lists of Fahrenheit temperatures.
#ifndef TEMPLIST_H
#define TEMPLIST_H
#include <iostream>
using namespace std;

namespace tlist
{
    const int MAX_LIST_SIZE = 50;
    class TemperatureList
    {
    public:
        TemperatureList();
        // Initializes the object to an empty list.
        void add_temperature(double temperature);
        // Precondition: The list is not full.
        // Postcondition: The temperature has been added to the list.
        bool full() const;
        // Returns true if the list is full, false otherwise.
        friend ostream& operator <<(ostream& outs,
                                   const TemperatureList& the_object);
        // Overloads the << operator to output TemperatureList values, one per line.
        // Precondition: If outs is a file output stream, then outs
        // has already been connected to a file.
    };
}
```

58

## Interface for a Class with an Array Member (2 of 2)

```
private:
    double list[MAX_LIST_SIZE]; // of temperatures in Fahrenheit
    int size; // number of array positions filled
};
// end namespace list
#endif
```

59

## Implementation for a Class with an Array Member (1 of 2)

```
// FILE: templist.cxx.. IMPLEMENTATION of class TemperatureList.
#include <iostream>
#include <cstdlib>
#include "templist.h"

namespace list
{
    TemperatureList::TemperatureList()
    {
        size = 0;
    }
    // Uses iostream.h and stdlib.h:
    void TemperatureList::add_temperature(double temperature)
    {
        if (full())
        {
            cout << "Error: adding to a full list.\n";
            exit(1);
        }
        else
        {
            list[size] = temperature;
            size = size + 1;
        }
    }
}
```

60

### Implementation for a Class with an Array Member (2 of 2)

```
bool TemperatureList::full() const
{ return (size == MAX_LIST_SIZE); }

// Uses iostream.h:
ostream& operator <<(ostream& outs, const TemperatureList&
the_object)
{
for (int i = 0; i < the_object.size; i++)
outs << the_object.list[i] << " F\n";
return outs;
}
} // namespace list
```

61

## Address Operator

- `&`
- all variables have an address in memory
- `&` placed before a variable produces the address of the variable
- examples:

```
void MyFn(int &i) ... // pass-by-reference, only
// the address is copied

int a = 3;
cout << &a << " " << a; // outputs the address of
// a and 3 (the value of a)

example output: 0x241ff5c 3
```

62

## Pointer

- a variable that can contains the address of another variable; variable that points to a variable
- cannot contain a value other than an address
- does not contain an address when declared
- a pointer must be declared to be of the type of the variable it is to point to

63

## Dereferencing Operator

- `*`
- also called *indirection* operator
- used to declare pointers and allow pointers to access the variable at the contained address
- syntax of pointer declaration: `type *identifier;`
- syntax of access of address value: `identifier`
- syntax of access of variable pointed to: `*identifier`

64

## Dereferencing Operator

- examples of declaring and using pointers:

```
int *ip;
int a = 3;
ip = &a;
cout << *ip; // dereferencing: outputs 3

double *dp;
double d = 1.2;
dp = &d;
cout << *dp; // outputs 1.2
```

65

## NULL

- pointers can be assigned the special value NULL
- NULL is a universally invalid address
- NULL can be assigned to a pointer of any type
- example:

```
int *p;
p = NULL;
```
- 0 can be used instead of NULL

66

## Structs, Classes and Pointers

- pointers can point to struct and class variables
- syntax of pointer and dot notaton:

```
(*pointer).member
```

- preferred syntax:

```
pointer->member
```

67

## Struct and Pointer Example

```
struct EmploymentStruct
{
    string name, position;
    double grossPay;
};

EmploymentStruct *ep, emp;
ep = &emp;
ep->name = "Joe";
ep->position = "Engineer";
ep->grossPay = 2500.0;
```

68

## Class and Pointer Example

```
class EmpClass
{ public:
    EmpClass(string iName,
             string iPos, double iPay);
    void PrintCheck();
private:
    string name, position;
    double grossPay;
};

EmpClass *ep, emp("Joe", "Sanitary
Engineer", 25000.0);
ep = &emp;
ep->PrintCheck();
```

69

## Dynamic Allocation

allocation of memory space for a variable at run time (dynamic as opposed to static allocation at compile time)

## Heap (Free Store)

pool of available memory locations reserved for dynamic allocation to dynamic objects

70

## Dynamic Variables

- variable that consists of type and memory address only
- created (memory allocated) when needed
- deleted (memory returned to heap) when no longer needed
- a pointer can point to a dynamic variable of the correct type

71

## Dynamic Variables

- the operator *new* is used to create dynamic variables
- syntax of *new*: *new constructor-or-type*
- examples:

```
int *p;
p = new int;
*p = 3;
cout << *p; // outputs 3
```

```
EmpStruct *esp;
esp = new EmpStruct;
esp->name = "Joe";

Empclass *ecp;
ecp = new EmpClass("Joe", "Eng", 25000);
ecp->PrintCheck();
```

72

## Deleting Dynamic Variables

- a dynamic variable must be explicitly returned to the memory heap with the *delete* operator when not needed
- syntax: `delete identifier;`
- examples:

```
delete p;
delete esp;
delete ecp;
```

73

## Operations on Pointer Variables

- if *p* and *q* are both pointers to the same type:

```
p = q; // copies the value in q into p; i.e. address q points to
// is copied, not the value at the address of p

p == q // true if p and q point to the same address

p != q // true if p and q do not point to the same address

p++ // increments the address stored in p by the width
// (number of bytes wide) of the type of p
```

74

## Functions can return Pointers:

```
#include <iostream>
using namespace std;
double * MyFn();

int main(){
    double *q;
    q = MyFn();
    cout << *q <<endl; // output: 3.14259
    delete q;
    return 0;
}

double * MyFn(){
    double *p = new double;
    *p = 3.14159;
    return p;
}
```

75

## Parameters can be Pointers:

```
#include <iostream>
using namespace std;
void MyFn(double* &p, double *q);

int main(){
    double *a, *b;
    a = new double;
    b = new double;
    *a = 9.8;
    *b = 7.6;
    MyFn(a,b);
    cout << *a << " " << *b; // output: 1.2 3.4
    return 0;
}

void MyFn(double* &p, double *q) {
    // p is pass-by-ref.
    // q is pass-by-val.
    *p = 1.2; // p refers to the address of a
    *q = 3.4; // q is a variable that contains
              // the address that b points to
}
```

76

12.2

## **Dynamic Arrays**

- A **dynamic array** is an array whose size is determined when the program is running, not when you write the program

77

## **Pointer Variables and Array Variables**

- Array variables are actually pointer variables that point to the first indexed variable
  - Example: 

```
int a[10];
typedef int* IntPtr;
IntPtr p;
```

– Variables *a* and *p* are the same kind of variable
- Since *a* is a pointer variable that points to *a[0]*,

```
p = a;
```

causes *p* to point to the same location as *a*

78

## Pointer Variables As Array Variables

- Continuing the previous example:  
Pointer variable `p` can be used as if it were an array variable
- Example: `p[0], p[1], ...p[9]`  
are all legal ways to use `p`
- Variable `a` can be used as a pointer variable except the pointer value in `a` cannot be changed
  - This is not legal: `IntPtr p2;`  
`... // p2 is assigned a value`  
`a = p2 // attempt to change a`

79

## Creating Dynamic Arrays

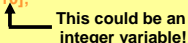
- Normal arrays require that the programmer determine the size of the array when the program is written
  - What if the programmer estimates too large?
    - Memory is wasted
  - What if the programmer estimates too small?
    - The program may not work in some situations
- Dynamic arrays can be created with just the right size while the program is running

80

## Creating Dynamic Arrays

- Dynamic arrays are created using the **new** operator
  - Example: To create an array of 10 elements of type `double`:

```
typedef double* DoublePtr;  
DoublePtr d;  
d = new double[10];
```

  
`d` can now be used as if it were an ordinary array!

81

## Dynamic Arrays (cont.)

- Pointer variable `d` is a pointer to `d[0]`
- When finished with the array, it should be deleted to return memory to the freestore
  - Example: `delete [] d;`
    - The brackets tell C++ a dynamic array is being deleted so it must check the size to know how many indexed variables to remove
    - Forgetting the brackets, is not legal, but would tell the computer to remove only one variable

82

## Pointer Arithmetic (Optional)

- Arithmetic can be performed on the addresses contained in pointers
  - Using the dynamic array of doubles, `d`, declared previously, recall that `d` points to `d[0]`
  - The expression `d+1` evaluates to the address of `d[1]`  
and `d+2` evaluates to the address of `d[2]`
    - Notice that adding one adds enough bytes for one variable of the type stored in the array

83

## Pointer Arithmetic Operations

- You can add and subtract with pointers
  - The `++` and `--` operators can be used
  - Two pointers of the same type can be subtracted to obtain the number of indexed variables between
    - The pointers should be in the same array!
  - This code shows one way to use pointer arithmetic:

```
for (int i = 0; i < array_size; i++)  
cout << *(d + i) << " ";  
// same as cout << d[i] << " " ;
```

84

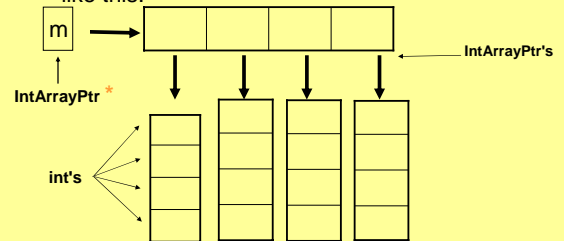
## Multidimensional Dynamic Arrays

- To create a 3x4 multidimensional dynamic array
  - View multidimensional arrays as arrays of arrays
  - First create a one-dimensional dynamic array
    - Start with a new definition:  
`typedef int* IntArrayPtr;`
    - Now create a dynamic array of pointers named `m`:  
`IntArrayPtr *m = new IntArrayPtr[3];`
  - For each pointer in `m`, create a dynamic array of int's
    - `for (int i = 0; i < 3; i++)`  
`m[i] = new int[4];`

85

- The dynamic array created on the previous slide could be visualized like this:

## A Multidimensional Dynamic Array



86

## Deleting Multidimensional Arrays

- To delete a multidimensional dynamic array
  - Each call to `new` that created an array must have a corresponding call to `delete[]`
  - Example: To delete the dynamic array created on a previous slide:  
`for (i = 0; i < 3; i++)`  
`delete [] m[i]; //delete the arrays of 4 int's`  
`delete [] m; // delete the array of IntArrayPtr's`

87

## Example – Processing 4D fMRI data

• D:\vo.dir\UCLA\_Classes\2007\Winter\Stat130D\_C\_PlusPlus.dir\SourceCode\ClassSourceDemos

- **fMRI\_Stats.c**

88