2023

Program Design II

Kun-Ta Chuang
Department of Computer Science and Information Engineering
National Cheng Kung University

Introduction to Arrays

- Array definition:
  - A collection of data of same type
- First "aggregate" data type
  - Means "grouping"
  - int, float, double, char are simple data types
- Used for lists of like items
  - Test scores, temperatures, names, etc.
  - Avoids declaring multiple simple variables
  - Can manipulate "list" as one entity

Declaring Arrays

- Declare the array → allocates memory
  int score[5];
  - Declares array of 5 integers named "score"
  - Similar to declaring five variables:
    int score[0], score[1], score[2], score[3], score[4]
- Individual parts called many things:
  - Indexed or subscripted variables
  - "Elements" of the array
  - Value in brackets called index or subscript
    - Numbered from 0 to size - 1

Accessing Arrays

- Access using index/subscript
  - cout << score[3];
- Note two uses of brackets:
  - In declaration, specifies SIZE of array
  - Anywhere else, specifies a subscript
- Size, subscript need not be literal
  - int score[MAX_SCORES];
  - score[n+1] = 99;
    - If n is 2, identical to: score[3]

Array Usage

- Powerful storage mechanism
- Can issue command like:
  - "Do this to i^{th} indexed variable"
    where i is computed by program
  - "Display all elements of array score"
  - "Fill elements of array score from user input"
  - "Find highest value in array score"
  - "Find lowest value in array score"

Array Program Example:
Display 5.1 Program Using an Array (1 of 2)

Display 5.1 Program Using an Array

```
//Reads in five scores and shows how much each
//score differs from the highest score.
#include <iostream>
using namespace std;
int main( )
{
    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].
    }
```

Array Program Example:
Display 5.1 Program Using an Array (2 of 2)

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

SAMPLE DIALOGUE

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

Major Array Pitfall

- Array indexes always start with zero!
- Zero is "first" number to computer scientists
- C++ will "let" you go beyond range
  - Unpredictable results
  - Compiler will not detect these errors!
- Up to programmer to "stay in range"

Major Array Pitfall Example

- Indexes range from 0 to (array_size – 1)
  - Example:
    double temperature[24]; // 24 is array size
    // Declares array of 24 double values called temperature
    - They are indexed as:
      temperature[0], temperature[1] … temperature[23]
  - Common mistake:
    temperature[24] = 5;
    - Index 24 is "out of range"!
    - No warning, possibly disastrous results

Defined Constant as Array Size

- Always use defined/named constant for array size
- Example:
  const int NUMBER_OF_STUDENTS = 5;
  int score[NUMBER_OF_STUDENTS];
- Improves readability
- Improves versatility
- Improves maintainability

Ranged-Based For Loop

- The C++11 ranged-based for loop makes it easy to iterate over each element in a loop
- Format

```
for (datatype varname : array)
{
    //  varname is set to each successive
     //  element in the array
}
```

- Example

```
int arr[] = {20, 30, 40, 50};
for (int x : arr)
   cout << x << " ";
cout << endl;
```

Output: 20 30 40 50

Auto-Initializing Arrays

- If fewer values than size supplied:
  - Fills from beginning
  - Fills "rest" with zero of array base type
- If array-size is left out
  - Declares array with size required based on number of initialization values
  - Example:
    int b[] = {5, 12, 11};
    - Allocates array b to size 3

Entire Arrays as Arguments

- Formal parameter can be entire array
  - Argument then passed in function call is array name
  - Called "array parameter"
- Send size of array as well
  - Typically done as second parameter
  - Simple int type formal parameter

Entire Array as Argument Example:
Display 5.3 Function with an Array Parameter

Display 5.3 Function with an Array Parameter

SAMPLE DIALOGUE

FUNCTION DECLARATION

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

SAMPLE DIALOGUE

FUNCTION DEFINITION

```
void fillUp(int a[], int size)
{
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    cout << "The last array index used is " << (size - 1) << endl;
}
```

Entire Array as Argument Example

- Given previous example:
- In some main() function definition, consider this calls:

```
int score[5], numberOfScores = 5;
fillup(score, numberOfScores);
```

  - 1st argument is entire array
  - 2nd argument is integer value
- Note no brackets in array argument!

Array as Argument: How?

- What's really passed?
- Think of array as 3 "pieces"
  - Address of first indexed variable (arrName[0])
  - Array base type
  - Size of array
- Only 1st piece is passed!
  - Just the beginning address of array
  - Very similar to "call-by-reference"

Partially-filled Arrays

- Difficult to know exact array size needed
- Must declare to be largest possible size
  - Must then keep "track" of valid data in array
  - Additional "tracking" variable needed
    - int numberUsed;
    - Tracks current number of elements in array

Partially-filled Arrays Example:
Display 5.5 Partially Filled Array (1 of 5)

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)

Partially-filled Arrays Example:
Display 5.5 Partially Filled Array (2 of 5)

Display 5.5 Partially Filled Array

```
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;
}
```

Partially-filled Arrays Example:
Display 5.5 Partially Filled Array (3 of 5)

```
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:
Display 5.5 Partially Filled Array (4 of 5)

```
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;
    }
}
```

Partially-filled Arrays Example: **Display 5.5** Partially Filled Array (5 of 5)

Display 5.5 Partially Filled Array

```
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 number.
69 74 68 -1
Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333
```

Searching/Sorting an Array

Display 5.6
Searching an Array (1 of 4)

Display 5.6 Searching an Array

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

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.

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

Display 5.6
Searching an Array (2 of 4)

```
int main( )
{
    int arr[DECLARED_SIZE], listSize, target;

    fillArray(arr, DECLARED_SIZE, listSize);

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

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

Display 5.6
Searching an Array (3 of 4)

Display 5.6 Searching an Array

```
        cout << "Search again?(y/n followed by Return): ";
        cin >> ans;
    } while ((ans != 'n') && (ans != 'N'));
    cout << "End of program.\n";
    return 0;
}

void fillArray(int a[], int size, int& numberUsed)
<The rest of the definition of fillArray is given in Display 5.5>
int search(const int a[], int numberUsed, int target)
{
    int index = 0;
    bool found = false;
    while ((!found) && (index < numberUsed))
    if (target == a[index])
        found = true;
    else
        index++;
```

Display 5.6
Searching an Array (4 of 4)

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

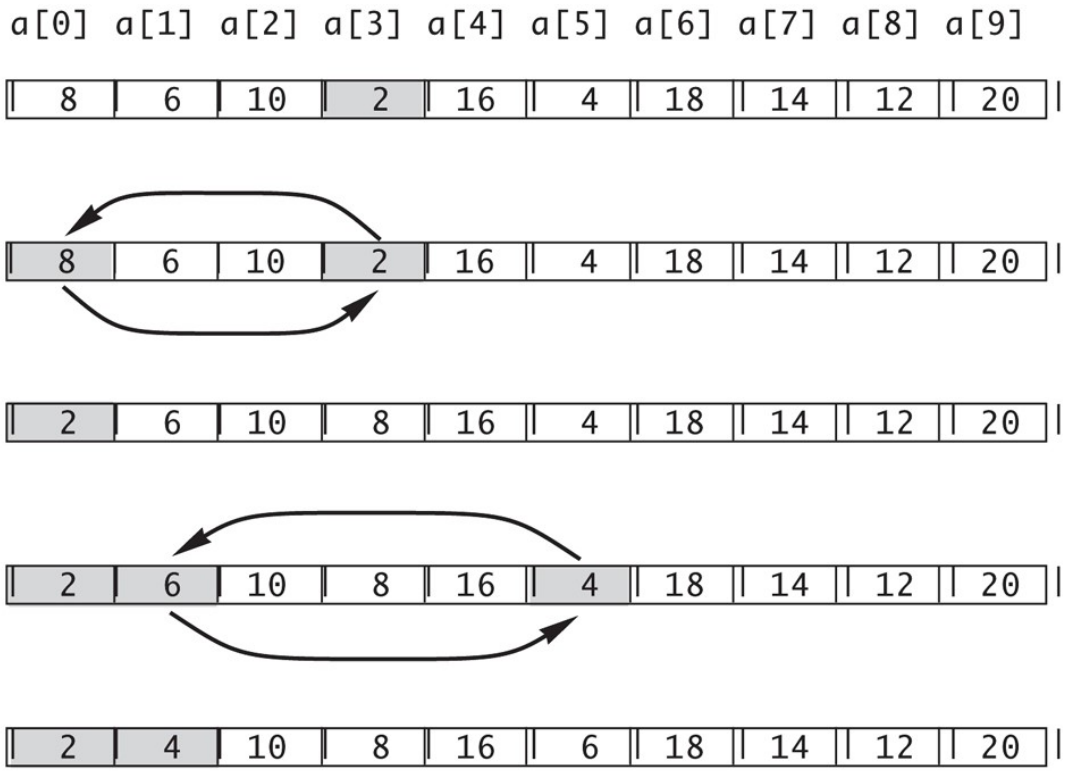
SAMPLE DIALOGUE

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

Sorting an Array:
Display 5.7 Selection Short

- Selection Sort Algorithm

Display 5.7 Selection Sort



https://www.hackerearth.com/practice/algorithms/sorting/selection-sort/visualize/

Sorting an Array Example:
Display 5.8 Sorting an Array (1 of 4)

Display 5.8 Sorting an Array

```
//Tests the procedure sort.
#include <iostream>
using namespace std;

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.
void sort(int a[], int numberUsed);
//Precondition: numberUsed <= declared size of the array a.
```

(continued)

Sorting an Array Example:
Display 5.8 Sorting an Array (2 of 4)

Display 5.8 Sorting an Array

```
//The array elements a[0] through a[numberUsed - 1] have values.
//Postcondition: The values of a[0] through a[numberUsed - 1] have
//been rearranged so that a[0] <= a[1] <= ... <= a[numberUsed - 1].

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

int indexOfSmallest(const int a[], int startIndex, int numberUsed);
//Precondition: 0 <= startIndex < numberUsed. Reference array elements
//have values. Returns the index i such that a[i] is the smallest of the
//values a[startIndex], a[startIndex + 1], ..., a[numberUsed - 1].

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

    int sampleArray[10], numberUsed;
    fillArray(sampleArray, 10, numberUsed);
    sort(sampleArray, numberUsed);

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

    return 0;
}
```

Sorting an Array Example:
Display 5.8 Sorting an Array (3 of 4)

```
void fillArray(int a[], int size, int& numberUsed)
            <The rest of the definition of fillArray is given in Display 5.5.>

void sort(int a[], int numberUsed)
{
    int indexOfNextSmallest;
    for (int index = 0; index < numberUsed - 1; index++)
    {//Place the correct value in a[index]:
        indexOfNextSmallest =
                    indexOfSmallest(a, index, numberUsed);
        swapValues(a[index], a[indexOfNextSmallest]);
        //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 swapValues(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
```

Sorting an Array Example:
Display 5.8 Sorting an Array (4 of 4)

Display 5.8 Sorting an Array

```
    v2 = temp;
}

int indexOfSmallest(const int a[], int startIndex, int numberUsed)
{
    int min = a[startIndex],
        indexOfMin = startIndex;
    for (int index = startIndex + 1; index < numberUsed; index++)
        if (a[index] < min)
        {
            min = a[index];
            indexOfMin = index;
            //min is the smallest of a[startIndex] through a[index]
        }

    return indexOfMin;
}
```

SAMPLE DIALOGUE

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

Summary 1

- Array is collection of "same type" data
- Indexed variables of array used just like any other simple variables
- for-loop "natural" way to traverse arrays
- Programmer responsible for staying "in bounds" of array
- Array parameter is "new" kind
  - Similar to call-by-reference

Summary 2

- Array elements stored sequentially
  - "Contiguous" portion of memory
  - Only address of 1st element is passed to functions
- Partially-filled arrays → more tracking
- Constant array parameters
  - Prevent modification of array contents
- Multidimensional arrays
  - Create "array of arrays“ (left for exercises by yourself)

Learning Objectives

- Structures
  - Structure types
  - Structures as function arguments
  - Initializing structures
- Classes
  - Defining, member functions
  - Public and private members
  - Accessor and mutator functions
  - Structures vs. classes

Structures

- 2nd aggregate data type: struct
- Recall: aggregate meaning "grouping"
  - Recall array: collection of values of same type
  - Structure: collection of values of different types
- Treated as a single item, like arrays
- Major difference: Must first "define" struct
  - Prior to declaring any variables

Structure Types

- Define struct globally (typically)
- No memory is allocated
  - Just a "placeholder" for what our struct will "look like"
- Definition:

```
struct CDAccountV1   ←Name of new struct "type"
{
        double balance;       ← member names
        double interestRate;
        int term;
};
```

Declare Structure Variable

- With structure type defined, now declare variables of this new type:
  CDAccountV1 account;
  - Just like declaring simple types
  - Variable *account* now of type CDAccountV1
  - It contains "member values"
    - Each of the struct "parts"

Accessing Structure Members

- Dot Operator to access members
  - account.balance
  - account.interestRate
  - account.term
- Called "member variables"
  - The "parts" of the structure variable
  - Different structs can have same name member variables
    - No conflicts

Structure Example:
Display 6.1 A Structure Definition (1 of 3)

Display 6.1 A Structure Definition

```
//Program to demonstrate the CDAccountV1 structure type.
#include <iostream>
using namespace std;

//Structure for a bank certificate of deposit:
struct CDAccountV1
{
    double balance;
    double interestRate;
    int term;//months until maturity
};

void getData(CDAccountV1& theAccount);
//Postcondition: theAccount.balance, theAccount.interestRate, and
//theAccount.term have been given values that the user entered at the keyboar
```

An improved version of this structure will be given later in this chapter.

Structure Example:
Display 6.1 A Structure Definition (2 of 3)

```
int main( )
{
    CDAccountV1 account;
    getData(account);

    double rateFraction, interest;
    rateFraction = account.interestRate/100.0;
    interest = account.balance*(rateFraction*(account.term/12.0));
    account.balance = account.balance + interest;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "When your CD matures in "
         << account.term << " months,\n"
         << "it will have a balance of $"
         << account.balance << endl;

    return 0;
}
```

(continued)

Structure Example:
Display 6.1 A Structure Definition (3 of 3)

Display 6.1 A Structure Definition

```
//Uses iostream:
void getData(CDAccountV1& theAccount)
{
    cout << "Enter account balance: $";
    cin >> theAccount.balance;
    cout << "Enter account interest rate: ";
    cin >> theAccount.interestRate;
    cout << "Enter the number of months until maturity: ";
    cin >> theAccount.term;
}
```

SAMPLE DIALOGUE

Enter account balance: **$100.00**
Enter account interest rate: **10.0**
Enter the number of months until maturity: **6**
When your CD matures in 6 months,
it will have a balance of $105.00

Structure Assignments

- Given structure named CropYield
- Declare two structure variables:
  CropYield apples, oranges;
  - Both are variables of "struct type CropYield"
  - Simple assignments are legal:
    apples = oranges;
    - Simply copies each member variable from apples into member variables from oranges

Structures as Function Arguments

- Passed like any simple data type
  - Pass-by-value
  - Pass-by-reference
  - Or combination
- Can also be returned by function
  - Return-type is structure type
  - Return statement in function definition sends structure variable back to caller

Initializing Structures

- Can initialize at declaration
  - Example:

```
struct Date
{
        int month;
        int day;
        int year;
};
Date dueDate = {12, 31, 2003};
```

  - Declaration provides initial data to all three member variables

Classes

- Similar to structures
  - Adds member FUNCTIONS
  - Not just member data
- Integral to object-oriented programming
  - Focus on objects
    - Object: Contains data and operations
    - In C++, variables of class type are objects

Class Definitions

- Defined similar to structures
- Example:

```
class DayOfYear    ← name of new class type
{
public:
        void output();    ← member function!
        int month;
        int day;
};
```

- Notice only member function's prototype
  - Function's implementation is elsewhere

Declaring Objects

- Declared same as all variables
  - Predefined types, structure types
- Example:

  DayOfYear today, birthday;
  - Declares two objects of class type DayOfYear
- Objects include:
  - Data
    - Members month, day
  - Operations (member functions)
    - output()

Class Member Access

- Members accessed same as structures
- Example:

  today.month
  today.day

  – And to access member function:
  today.output(); ← Invokes member function

Class Member Functions

- Must define or "implement" class member functions
- Like other function definitions
  - Can be after main() definition
  - Must specify class:
    void DayOfYear::output()
    {…}
    - :: is scope resolution operator
    - Instructs compiler "what class" member is from
    - Item before :: called type qualifier

Complete Class Example:
Display 6.3 Class With a Member Function (1 of 4)

Display 6.3 Class with a Member Function

```
//Program to demonstrate a very simple example of a class.
//A better version of the class DayOfYear will be given in Display 6.4.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    void output( );
    int month;
    int day;
};

int main( )
{
    DayOfYear today, birthday;
    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;
```

Normally, member variables are **private** *and not* **public**, *as in this example. This is discussed a bit later in this chapter.*

Member function declaration

(continued)

Complete Class Example:
Display 6.3 Class With a Member Function (2 of 4)

Display 6.3 Class with a Member Function

```
    cout << "Today's date is ";
    today.output( );
    cout << endl;
    cout << "Your birthday is ";
    birthday.output( );
    cout << endl;

    if (today.month == birthday.month && today.day == birthday.day)
        cout << "Happy Birthday!\n";
    else
        cout << "Happy Unbirthday!\n";
    return 0;
}
//Uses iostream:
void DayOfYear::output( )
{
    switch (month)
    {
        case 1:
            cout << "January "; break;
        case 2:
            cout << "February "; break;
        case 3:
            cout << "March "; break;
        case 4:
            cout << "April "; break;
```

Calls to the member function **output**

Member function definition

Complete Class Example:
Display 6.3 Class With a Member Function (3 of 4)

```
        case 5:
            cout << "May "; break;
        case 6:
            cout << "June "; break;
        case 7:
            cout << "July "; break;
        case 8:
            cout << "August "; break;
        case 9:
            cout << "September "; break;
        case 10:
            cout << "October "; break;
        case 11:
            cout << "November "; break;
        case 12:
            cout << "December "; break;
        default:
            cout << "Error in DayOfYear::output. Contact software vendor.";
    }

    cout << day;
}
```

Complete Class Example:
Display 6.3 Class With a Member Function (4 of 4)

Display 6.3 Class with a Member Function

SAMPLE DIALOGUE

```
Enter today's date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
Today's date is October 15
Your birthday is February 21
Happy Unbirthday!
```

Dot and Scope Resolution Operator

- Used to specify "of what thing" they are members
- Dot operator:
  - Specifies member of particular object
- Scope resolution operator:
  - Specifies what class the function definition comes from

A Class's Place

- Class is full-fledged type!
  - Just like data types int, double, etc.
- Can have variables of a class type
  - We simply call them "objects"
- Can have parameters of a class type
  - Pass-by-value
  - Pass-by-reference
- Can use class type like any other type!

Encapsulation

- Any data type includes
  - Data (range of data)
  - Operations (that can be performed on data)
- Example:
  int data type has:
  Data: -2147483648 to 2147483647 (for 32 bit int)
  Operations: +,-,*,/,%,logical,etc.
- Same with classes
  - But WE specify data, and the operations to be allowed on our data!

More Encapsulation

- Encapsulation
  - Means "bringing together as one"
- Declare a class → get an object
- Object is "encapsulation" of
  - Data values
  - Operations on the data (member functions)

Principles of OOP

- Information Hiding
  - Details of how operations work not known to "user" of class
- Data Abstraction
  - Details of how data is manipulated within ADT(Abstract Data Type)/class not known to user
- Encapsulation
  - Bring together data and operations, but keep "details" hidden

Public and Private Members

- Data in class almost always designated private in definition!
  - Upholds principles of OOP
  - Hide data from user
  - Allow manipulation only via operations
    - Which are member functions
- Public items (usually member functions) are "user-accessible"

Public and Private Example

- Modify previous example:

```
class DayOfYear
{
public:
	void input();
	void output();
private:
	int month;
	int day;
};
```

- Data now private
- Objects have no direct access

Public and Private Example 2

- Given previous example
- Declare object:
  DayOfYear today;
- Object *today* can ONLY access public members
  - cin >> today.month; // NOT ALLOWED!
  - cout << today.day; // NOT ALLOWED!
  - Must instead call public operations:
    - today.input();
    - today.output();

Public and Private Style

- Can mix & match public & private
- More typically place public first
  - Allows easy viewing of portions that can be USED by programmers using the class
  - Private data is "hidden", so irrelevant to users
- Outside of class definition, cannot change (or even access) private data

Accessor(get) and Mutator(set) Functions

- Object needs to "do something" with its data
- Call accessor member functions
  - Allow object to read data
  - Also called "get member functions"
  - Simple retrieval of member data
- Mutator member functions
  - Allow object to change data
  - Manipulated based on application

Separate Interface and Implementation

- User of class need not see details of how class is implemented
  - Principle of OOP → encapsulation
- User only needs "rules"
  - Called "interface" for the class
    - In C++ → public member functions and associated comments
- Implementation of class hidden
  - Member function definitions elsewhere
  - User need not see them

Thinking Objects

- Focus for programming changes
  - Before → algorithms center stage
  - OOP → data is focus
- Algorithms still exist
  - They simply focus on their data
  - Are "made" to "fit" the data
- Designing software solution
  - Define variety of objects and how they interact

Learning Objectives

- Constructors
  - Definitions
  - Calling
- More Tools
  - const parameter modifier
  - Inline functions
  - Static member data
- Vectors
  - Introduction to vector class

Constructors

- Initialization of objects
  - Initialize some or all member variables
  - Other actions possible as well
- A special kind of member function
  - Automatically called when object declared
- Very useful tool
  - Key principle of OOP

Constructor Definitions

- Constructors defined like any member function
  - Except:
    1. Must have same name as class
    2. Cannot return a value; not even void!

Constructor Definition Example

- Class definition with constructor:
  -
```
class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
        //Constructor initializes month and day
    void input();
    void output();
    …
private:
    int month;
    int day;
}
```

Constructor Notes

- Notice name of constructor: DayOfYear
  - Same name as class itself!
- Constructor declaration has no return-type
  - Not even void!
- Constructor in public section
  - It’s called when objects are declared
  - If private, could never declare objects!

Calling Constructors

- Declare objects:

  DayOfYear date1(7, 4),
  date2(5, 5);

- Objects are created here
  - Constructor is called
  - Values in parens passed as arguments to constructor
  - Member variables month, day initialized:
    date1.month → 7 date2.month → 5
    date1.dat → 4 date2.day → 5

Constructor Equivalency

- Consider:
  - DayOfYear date1, date2
    date1.DayOfYear(7, 4); // ILLEGAL!
    date2.DayOfYear(5, 5); // ILLEGAL!
- CANNOT call constructors like other member functions!

Constructor Code

- Constructor definition is like all other member functions:

```
DayOfYear::DayOfYear(int monthValue, int dayValue)
{
    month = monthValue;
    day = dayValue;
}
```

- Note same name around ::
  - Clearly identifies a constructor
- Note no return type
  - Just as in class definition

Constructor Additional Purpose

- Not just initialize data
- Body doesn’t have to be empty
  - In initializer version
- Validate the data!
  - Ensure only appropriate data is assigned to
    class private member variables
  - Powerful OOP principle

Overloaded Constructors

- Can overload constructors just like other functions
- Recall: a signature consists of:
  - Name of function
  - Parameter list
- Provide constructors for all possible argument-lists
  - Particularly "how many"

Class with Constructors Example:
Display 7.1 Class with Constructors (1 of 3)

Display 7.1 Class with Constructors

```
#include <iostream>
#include <cstdlib> //for exit
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    //Initializes the month and day to arguments.

    DayOfYear(int monthValue);
    //Initializes the date to the first of the given month.

    DayOfYear( );
    //Initializes the date to January 1.

    void input( );
    void output( );
    int getMonthNumber( );
    //Returns 1 for January, 2 for February, etc.
```

This definition of **DayOfYear** *is an improved version of the class* **DayOfYear** *given in Display 6.4.*

default constructor

Class with Constructors Example:
Display 7.1 Class with Constructors (2 of 3)

```
    int getDay( );
private:
    int month;
    int day;
    void testDate( );
};

int main( )
{
    DayOfYear date1(2, 21), date2(5), date3;
    cout << "Initialized dates:\n";
    date1.output( ); cout << endl;
    date2.output( ); cout << endl;
    date3.output( ); cout << endl;

    date1 = DayOfYear(10, 31);
    cout << "date1 reset to the following:\n";
    date1.output( ); cout << endl;
    return 0;
}

DayOfYear::DayOfYear(int monthValue, int dayValue)
                            : month(monthValue), day(dayValue)
{
    testDate( );
}
```

This causes a call to the default constructor. Notice that there are no parentheses.

an explicit call to the constructor `DayOfYear::DayOfYear`

Class with Constructors Example:
Display 7.1 Class with Constructors (3 of 3)

Display 7.1 Class with Constructors

```
DayOfYear::DayOfYear(int monthValue) : month(monthValue), day(1)
{
    testDate( );
}

DayOfYear::DayOfYear( ) : month(1), day(1)
{/*Body intentionally empty.*/}

//uses iostream and cstdlib:
void DayOfYear::testDate( )
{
    if ((month < 1) || (month > 12))
    {
        cout << "Illegal month value!\n";
        exit(1);
    }
    if ((day < 1) || (day > 31))
    {
        cout << "Illegal day value!\n";
        exit(1);
    }
}
```

<Definitions of the other member functions are the same as in Display 6.4.>

SAMPLE DIALOGUE

```
Initialized dates:
February 21
May 1
January 1
date1 reset to the following:
October 31
```

Constructor with No Arguments

- Can be confusing
- Standard functions with no arguments:
  - Called with syntax: callMyFunction();
    - Including empty parentheses
- Object declarations with no "initializers":
  - DayOfYear date1; // This way!
  - DayOfYear date(); // NO!
    - What is this really?
    - Compiler sees a function declaration/prototype!
    - Yes! Look closely!

Explicit Constructor Calls

- Can also call constructor AGAIN
  - After object declared
    - Recall: constructor was automatically called then
  - Can call via object's name; standard member function call
- Convenient method of setting member variables
- Method quite different from standard member function call

Explicit Constructor Call Example

- Such a call returns "anonymous object"
  - Which can then be assigned
  - **In Action**:
    DayOfYear holiday(7, 4);
    - Constructor called at object's declaration
    - Now to "re-initialize":
      holiday = DayOfYear(5, 5);
      - Explicit constructor call
      - Returns new "anonymous object"
      - Assigned back to current object

Class Type Member Variables

- Class member variables can be any type
  - Including objects of other classes!
  - Type of class relationship
    - Powerful OOP principle
- Need special notation for constructors
  - So they can call "back" to member object’s constructor

Class Member Variable Example:
Display 7.3 A Class Member Variable (1 of 5)

Display 7.3 A Class Member Variable

```
#include <iostream>
#include<cstdlib>
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int monthValue, int dayValue);
    DayOfYear(int monthValue);
    DayOfYear( );
    void input( );
    void output( );
    int getMonthNumber( );
    int getDay( );
private:
    int month;
    int day;
    void testDate( );
};
```

The class **DayOfYear** *is the same as in Display 7.1, but we have repeated all the details you need for this discussion.*

Class Member Variable Example:
Display 7.3 A Class Member Variable (2 of 5)

```
class Holiday
{
public:
    Holiday( );//Initializes to January 1 with no parking enforcement
    Holiday(int month, int day, bool theEnforcement);
    void output( );
private:
    DayOfYear date;
    bool parkingEnforcement;//true if enforced
};

int main( )
{
    Holiday h(2, 14, true);
    cout << "Testing the class Holiday.\n";
    h.output( );

    return 0;
}

Holiday::Holiday( ) : date(1, 1), parkingEnforcement(false)
{/*Intentionally empty*/}

Holiday::Holiday(int month, int day, bool theEnforcement)
                    : date(month, day), parkingEnforcement(theEnforcement)
{/*Intentionally empty*/}
```

member variable of a class type

Invocations of constructors from the class `DayOfYear`.

(continued)

Class Member Variable Example:
Display 7.3 A Class Member Variable (3 of 5)

Display 7.3 A Class Member Variable

```
void Holiday::output( )
{
    date.output( );
    cout << endl;
    if (parkingEnforcement)
        cout << "Parking laws will be enforced.\n";
    else
        cout << "Parking laws will not be enforced.\n";
}

DayOfYear::DayOfYear(int monthValue, int dayValue)
                          : month(monthValue), day(dayValue)
{
    testDate( );
}
```

Class Member Variable Example:
Display 7.3 A Class Member Variable (4 of 5)

```
//uses iostream and cstdlib:
void DayOfYear::testDate( )
{
    if ((month < 1) || (month > 12))
    {
        cout << "Illegal month value!\n";
        exit(1);
    }
    if ((day < 1) || (day > 31))
    {
        cout << "Illegal day value!\n";
        exit(1);
    }
}

//Uses iostream:
void DayOfYear::output( )
{
    switch (month)
    {
        case 1:
            cout << "January "; break;
        case 2:
            cout << "February "; break;
        case 3:
            cout << "March "; break;
          .
          .
          .
```

The omitted lines are in Display 6.3, but they are obvious enough that you should not have to look there.

Class Member Variable Example:
Display 7.3 A Class Member Variable (5 of 5)

Display 7.3 A Class Member Variable

```
        case 11:
            cout << "November "; break;
        case 12:
            cout << "December "; break;
        default:
            cout << "Error in DayOfYear::output. Contact software vendor.";
    }

    cout << day;
}
```

SAMPLE DIALOGUE

```
Testing the class Holiday.
February 14
Parking laws will be enforced.
```

Parameter Passing Methods

- Efficiency of parameter passing
  - Call-by-value
    - Requires copy be made → Overhead
  - Call-by-reference
    - Placeholder for actual argument
    - Most efficient method
  - Negligible difference for simple types
  - For class types → clear advantage
- Call-by-reference desirable
  - Especially for "large" data, like class types

The const Parameter Modifier

- Large data types (typically classes)
  - Desirable to use pass-by-reference
  - Even if function will not make modifications
- Protect argument
  - Use constant parameter
    - Also called constant call-by-reference parameter
  - Place keyword *const* before type
  - Makes parameter "read-only"
  - Attempt to modify parameter results in compiler error

Use of const

- All-or-nothing
- If no need for function modifications
  - Protect parameter with const
  - Protect ALL such parameters
- This includes class member function parameters

Inline Functions

- For non-member functions:
  - Use keyword *inline* in function declaration and function heading
- For class member functions:
  - Place implementation (code) for function IN class definition → automatically inline
- Use for very short functions only
- Code actually inserted in place of call
  - Eliminates overhead
  - More efficient, but only when short!

Inline Member Functions

- Member function definitions
  - Typically defined separately, in different file
  - Can be defined IN class definition
    - Makes function "in-line"
- Again: use for very short functions only
- More efficient
  - If too long → actually less efficient!

Member Initializers

- C++11 supports a feature called member initialization
  - This feature allows you to set default values for member variables

```
class Coordinate
{
  public:
        Coordinate();
  private:
        int x=1;
        int y=2;
};
Coordinate::Coordinate()
{ }
```

Member Initializers

`Coordinate c1;` Initializes c1.x to 1 and c1.y to 2

Constructor Delegation

- C++11 allows one constructor to invoke another

```
Coordinate::Coordinate(int xval, int yval) : x(xval), y(yval)
{ }

Coordinate::Coordinate() : Coordinate(99,99)
{ }
```

- The default constructor invokes the constructor to initialize x and y to 99,99

Static Members

- Static member variables
  - All objects of class "share" one copy
  - One object changes it → all see change
- Useful for "tracking"
  - How often a member function is called
  - How many objects exist at given time
- Place keyword *static* before type

Static Functions

- Member functions can be static
  - If no access to object data needed
  - And still "must" be member of the class
  - Make it a static function
- Can then be called outside class
  - From non-class objects:
    - E.g., Server::getTurn();
  - As well as via class objects
    - Standard method: myObject.getTurn();
- Can only use static data, functions!

Static Members Example:
Display 7.6 Static Members (1 of 4)

Display 7.6 Static Members

```
#include <iostream>
using namespace std;

class Server
{
public:
    Server(char letterName);
    static int getTurn( );
    void serveOne( );
    static bool stillOpen( );
private:
    static int turn;
    static int lastServed;
    static bool nowOpen;
    char name;
};

int Server:: turn = 0;
int Server:: lastServed = 0;
bool Server::nowOpen = true;
```

Static Members Example:
Display 7.6 Static Members (2 of 4)

```
int main( )
{
    Server s1('A'), s2('B');
    int number, count;
    do
    {
        cout << "How many in your group? ";
        cin >> number;
        cout << "Your turns are: ";
        for (count = 0; count < number; count++)
            cout << Server::getTurn( ) << ' ';
        cout << endl;
        s1.serveOne( );
        s2.serveOne( );
    } while (Server::stillOpen( ));

    cout << "Now closing service.\n";

    return 0;
}
```

Static Members Example:
Display 7.6 Static Members (3 of 4)

Display 7.6 Static Members

```
Server::Server(char letterName) : name(letterName)
{/*Intentionally empty*/}

int Server::getTurn( )
{
    turn++;
    return turn;
}
bool Server::stillOpen( )
{
    return nowOpen;
}

void Server::serveOne( )
{
    if (nowOpen && lastServed < turn)
    {
        lastServed++;
        cout << "Server " << name
            << " now serving " << lastServed << endl;
     }
```

Since **getTurn** *is static, only static members can be referenced in here.*

Static Members Example:
Display 7.6 Static Members (4 of 4)

```
58        if (lastServed >= turn) //Everyone served
59            nowOpen = false;
60    }
```

SAMPLE DIALOGUE

How many in your group? **3**
Your turns are: 1 2 3
Server A now serving 1
Server B now serving 2
How many in your group? **2**
Your turns are: 4 5
Server A now serving 3
Server B now serving 4
How many in your group? **0**
Your turns are:
Server A now serving 5
Now closing service.

Vectors

- Vector Introduction
  - Recall: arrays are fixed size
  - Vectors: "arrays that grow and shrink"
    - During program execution
  - Formed from Standard Template Library (STL)
    - Using template class

Vector Basics

- Similar to array:
  - Has base type
  - Stores collection of base type values
- Declared differently:
  - Syntax: vector<Base_Type>
    - Indicates template class
    - Any type can be "plugged in" to Base_Type
    - Produces "new" class for vectors with that type
  - Example declaration:
    vector<int> v;

Vector Use

- vector<int> v;
  - "v is vector of type int"
  - Calls class default constructor
    - Empty vector object created
- Indexed like arrays for access
- But to add elements:
  - Must call member function push_back
- Member function size()
  - Returns current number of elements

Vector Example:
Display 7.7 Using a Vector (1 of 2)

Display 7.7 Using a Vector

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> next;
    }
```

Vector Example:
Display 7.7 Using a Vector (2 of 2)

```
    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

SAMPLE DIALOGUE

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size = 1
4 added. v.size = 2
6 added. v.size = 3
8 added. v.size = 4
You entered:
2 4 6 8
```

Vector Efficiency

- Member function capacity()
  - Returns memory currently allocated
  - Not same as size()
  - Capacity typically > size
    - Automatically increased as needed
- If efficiency critical:
  - Can set behaviors manually
    - v.reserve(32); //sets capacity to 32
    - v.reserve(v.size()+10); //sets capacity to 10 more than size

Summary 1

- Constructors: automatic initialization of class data
  - Called when objects are declared
  - Constructor has same name as class
- Default constructor has no parameters
  - Should always be defined
- Class member variables
  - Can be objects of other classes
    - Require initialization-section

Summary 2

- Constant call-by-reference parameters
  - More efficient than call-by-value
- Can *inline* very short function definitions
  - Can improve efficiency
- Static member variables
  - Shared by all objects of a class
- Vector classes
  - Like: "arrays that grow and shrink"