Data

• The representation of information in a manner suitable for communication or analysis by humans or machines

• Data are the nouns of the programming world:
  • The objects that are manipulated
  • The information that is processed
Data Abstraction

- Separation of a data type’s logical properties from its implementation.

<table>
<thead>
<tr>
<th>LOGICAL PROPERTIES</th>
<th>IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>What are the possible values?</td>
<td>How can this be done in C++?</td>
</tr>
<tr>
<td>What operations will be needed?</td>
<td>How can data types be used?</td>
</tr>
</tbody>
</table>
Data Encapsulation

• is the separation of the representation of data from the applications that use the data at a logical level; a programming language feature that enforces information hiding.

```
int y;
y = 25;
```
Encapsulated C++ Data Type int

**Value range:**
INT_MIN . . INT_MAX

**Operations:**

- prefix +
- prefix -
+ infix +
- infix -
* infix *
/ infix /
% infix %

**Relational Operators:**
infix

**Representation of int as 16 bits two’s complement:**

Implementation of Operations +
Abstract Data Type (ADT)

- A data type whose properties (domain and operations) are specified independently of any particular implementation.
Collection ordered in different ways
Data from 3 different levels

- **Application (or user) level**: modeling real-life data in a specific context.

- **Logical (or ADT) level**: abstract view of the domain and operations. **WHAT**

- **Implementation level**: specific representation of the structure to hold the data items, and the coding for operations. **HOW**
Communication between the Application Level and Implementation Level
Viewing a library from 3 different levels

- **Application (or user) level**: Library of Congress, or Baltimore County Public Library.

- **Logical (or ADT) level**: domain is a collection of books; operations include: check book out, check book in, pay fine, reserve a book.

- **Implementation level**: representation of the structure to hold the “books”, and the coding for operations.
Composite Data Type

A composite data type is a type which

• stores a collection of individual data components under one variable name,

• and allows the individual data components to be accessed.
4 Basic Kinds of ADT Operations

• **Constructor** -- creates a new instance (object) of an ADT.

• **Transformer** -- changes the state of one or more of the data values of an instance.

• **Observer** -- allows us to observe the state of one or more of the data values without changing them.

• **Iterator** -- allows us to process all the components in a data structure sequentially.
Two Forms of Composite Data Types

**UNSTRUCTURED**

Components are not organized with respect to one another.

**EXAMPLES:**
classes and structs

**STRUCTURED**

The organization determines method used to access individual data components.

**EXAMPLES:**
arrays
C++ Built-In Data Types

Simple
- Integral
  - char
  - short
  - int
  - long
  - enum
- Floating
  - float
  - double
  - long double

Composite
- array
- struct
- union
- class

Address
- pointer
- reference
A record is a composite data type made up of a finite collection of not necessarily homogeneous elements called members or fields. For example . . .

```
thisCar at Base Address 6000

.year  2008
.maker 'h' 'o' 'n' 'd' 'a' '\0' . . .
.price 18678.92
```
struct CarType
{
    int     year;
    char    maker[10];
    float   price;
}

CarType    thisCar;  //CarType variables
CarType    myCar;
Accessing struct members

The **member selection operator** (period . ) is used between the variable name and the member identifier to access individual members of a record (struct or class) type variable.

**EXAMPLES**

- myCar.year
- thisCar.maker[4]
Valid struct operations

- Operations valid on an entire struct type variable:
  - assignment to another struct variable of same type,
  - pass as a parameter to a function (either by value or by reference),
  - return as the value of a function.
Pass-by-value

sends a copy of the contents of the actual parameter

SO, the actual parameter cannot be changed by the function.
Pass-by-reference

sends the location (memory address) of the actual parameter

can change value of actual parameter
void AdjustForInflation(CarType& car, float perCent)

// Increases price by the amount specified in perCent
{
    car.price = car.price * perCent + car.price;
};

SAMPLE CALL

AdjustForInflation(myCar, 0.03);
Using struct type
Value Parameter to examine a member

```cpp
bool LateModel(CarType car, int date)
{
    return (car.year >= date);
}
```

**SAMPLE CALL**

```cpp
if (LateModel(myCar, 1995))
    std::cout << myCar.price << std::endl;
```
One-Dimensional Array at the Logical Level

A one-dimensional array is a structured composite data type made up of a finite, fixed size (known at compile time) collection of homogeneous (all of the same data type) elements having relative positions and to which there is direct access (any element can be accessed immediately).

Array operations (creation, storing a value, retrieving a value) are performed using a declaration and indexes.
Implementation Example

This ACCESSING FUNCTION gives position of values[Index]

Address(Index) = BaseAddress + Index * SizeOfElement

float values[5]; // assume element size is 4 bytes

Base Address

<table>
<thead>
<tr>
<th>7000</th>
<th>7004</th>
<th>7008</th>
<th>7012</th>
<th>7016</th>
</tr>
</thead>
</table>


Indexes
One-Dimensional Arrays in C++

- The index must be of an integral type (char, short, int, long, or enum).
- The index range is always 0 through the array size minus 1.
- Arrays cannot be assigned one to another, and cannot be the return type of a function.
Another Example

This ACCESSING FUNCTION gives position of name[Index]

Address(Index) = BaseAddress + Index * SizeOfElement

```
char name[10];       // assume element size is 1 byte
```

```
Base Address

6000  6001  6002  6003  6004  6005  6006  6007  6008  6009

```
Passing Arrays as Parameters

• In C++, arrays are always passed by reference, and & is not used with the formal parameter type.

• Whenever an array is passed as a parameter, its base address is sent to the called function.
const array parameter

Because arrays are always passed as reference parameters, you can protect the actual parameter from unintentional changes by using const in formal parameter list and function prototype.

FOR EXAMPLE . . .

```c
// prototype

float SumValues(const float values[],
                 int numOfValues);
```
float SumValues (const float values[],
    numOfValues)
    // Pre: values[0] through values[numOfValues-1]
    //      have been assigned
    // Returns the sum of values[0] through
    // values[numOfValues-1]
{
    float sum = 0;
    for ( int index = 0; index < numOfValues; 
        index++ )
    {
        sum = values [index] + sum;
    }
    return sum;
}
Two-Dimensional Array at the Logical Level

A two-dimensional array is a structured composite data type made up of a finite, fixed size collection of homogeneous elements having relative positions and to which there is direct access.

Array operations (creation, storing a value, retrieving a value) are performed using a declaration and a pair of indexes (called row and column) representing the component’s position in each dimension.
EXAMPLE -- To keep monthly high temperatures for 50 states in a two-dimensional array.

```c
const int NUM_STATES = 50;
const int NUM_MONTHS = 12;
int stateHighs[NUM_STATES][NUM_MONTHS];
```

Row 2, col 7 might be Arizona's high for August.
Finding the average high temperature for Arizona

```c
float total = 0;
int month;
float average;

for ( month = 0; month < NUM_MONTHS; month ++ )
    total = total + stateHighs[2][month];

average = ( total / 12.0 + 0.5 );
```
const int NUM_STATES = 50;
const int NUM_MONTHS = 12;
int stateHighs[NUM_STATES][NUM_MONTHS];

STORAGE
• In memory, C++ stores arrays in row order. The first row is followed by the second row, etc.

Base Address

8000  8024  8048  ...
To locate an element such as `stateHighs[2][7]` the compiler needs to know that there are 12 columns in this two-dimensional array.

At what address will `stateHighs[2][7]` be found?

Assume 2 bytes for type int.
Example of a 2-dimensional object
Two-Dimensional Array Parameters

• Just as with a one-dimensional array, when a two- (or higher) dimensional array is passed as a parameter, the base address of the actual array is sent to the function.

• The size of all dimensions except the first must be included in the function heading and prototype.

• The sizes of those dimensions for the formal parameter must be exactly the same as in the actual array.
Use the two-dimensional stateHighs array to fill a one-dimensional stateAverages array

const int NUM_STATES = 50;
const int NUM_MONTHS = 12;
int stateHighs[NUM_STATES][NUM_MONTHS];
float stateAverages[NUM_STATES];
void findAverages (const int stateHighs [ ] [ NUM_MONTHS],
                int stateAverages [ ] )

    // Pre:  stateHighs[ 0..NUM_STATES-1] [ 0..NUM_MONTHS-1]
    // assigned
    // Post:  stateAverages[ 0..NUM_STATES-1 ] contains rounded
    // high temperature for each state
    {
        int state;
        int month;
        float total;
        for ( state = 0 ;  state  <  NUM_STATES;  state++ )
        {
            total = 0.0;
            for ( month = 0 ;  month  <  NUM_MONTHS ; month++ )
                total = stateHighs [ state ][ month ] + total;
            stateAverages [ state ] = total / 12.0 + 0.5;
        }
Using typedef with arrays helps eliminate the chances of size mismatches between formal and actual parameters. FOR EXAMPLE,

```c
typedef int StateHighsType [ NUM_STATES ] [ NUM_MONTHS ];

typedef float StateAveragesType [ NUM_STATES ];

void findAverages( const StateHighsType stateHighs,
                    StateAveragesType stateAverages )
{
    
    
}
```
Declaring Multidimensional Arrays

**EXAMPLE USING TYPEDEF**

```c
const int NUM_DEPTS = 5;
// mens, womens, childrens, electronics, linens
const int NUM_MONTHS = 12;
const int NUM_STORES = 3;
// White Marsh, Owings Mills, Towson

typedef long MonthlySalesType [NUM_DEPTS] [NUM_MONTHS] [NUM_STORES];

MonthlySalesType monthlySales;
```
```c
const int NUM_DEPTS = 5; // mens, womens, childrens, electronics, linens
const int NUM_MONTHS = 12;
const int NUM_STORES = 3; // White Marsh, Owings Mills, Towson

typedef long MonthlySalesType [NUM_DEPTS] [NUM_MONTHS] [NUM_STORES];
MonthlySalesType monthlySales;

monthlySales[3][7][0] sales for electronics in August at White Marsh
```
C++ class data type

- A class is an unstructured type that encapsulates a fixed number of data components (data members) with the functions (called member functions) that manipulate them.

- The predefined operations on an instance of a class are whole assignment and component access.
class DateType Specification

// SPECIFICATION FILE ( datatype.h )

class DateType // declares a class data type
{

public:
    // 4 public member functions
    void Initialize ( int newMonth, int newDay, int newYear ) ;
    int GetYear() const ;   // returns year
    int GetMonth() const ;  // returns month
    int GetDay() const ;    // returns day

private:
    // 3 private data members
    int year ;
    int month ;
    int day ;
};

; must be there!!!
Use of C++ data type class

- Variables of a class type are called **objects** (or instances) of that particular class.

- Software that declares and uses objects of the class is called a **client**.

- Client code uses public member functions (called methods in OOP) to handle its class objects.

- - means calling a public member function.
#include "datatype" // includes specification of the class
using namespace std;

int main ( void )
{
    DateType startDate; // declares 2 objects of DateType
    DateType endDate;
    bool retired = false;
    startDate.Initialize ( 6, 30, 1998 );
    endDate.Initialize ( 10, 31, 2002 );
    cout << startDate.MonthIs() << "/" << startDate.DayIs() 
        << "/" << startDate.YearIs() << endl;
    while ( ! retired )
    {
        // finishSomeTask
    }
}
2 separate files generally used for class type

// SPECIFICATION FILE (datatype.h)
// Specifies the data and function members.
class DateType
{
    public:
        . . .
    private:
        . . .
};

// IMPLEMENTATION FILE (datatype.cpp)

// Implements the DateType member functions.
**Start Date**

- Initialize
- GetYear
- GetMonth
- GetDay

**Private data:**
- **year:** 1998
- **month:** 6
- **day:** 30

**End Date**

- Initialize
- GetYear
- GetMonth
- GetDay

**Private data:**
- **year:** 2002
- **month:** 10
- **day:** 31

DateType Class Instance Diagrams
Implementation of DateType member functions

// IMPLEMENTATION FILE                     (datetype.cpp)
#include “datetype.h”     // also must appear in client code

void DateType :: Initialize ( int  newMonth, int  newDay,
                                int  newYear )

//  Post:  year is set to newYear.
//         month is set to newMonth.
//         day is set to newDay.
{
    year =  newYear;
    month =  newMonth;
    day  =  newDay;
}
int DateType :: GetMonth ( ) const
// Accessor function for data member month
{
    return  month;
}

int DateType :: GetYear ( ) const
// Accessor function for data member year
{
    return  year;
}

int DateType :: GetDay ( ) const
// Accessor function for data member day
{
    return  day;
}
Familiar Class Instances and Member Functions

• The member selection operator ( . ) selects either data members or member functions.

• Header files `iostream` and `fstream` declare the istream, ostream, and ifstream, ofstream I/O classes.

• Both `cin` and `cout` are class objects and `get` and `ignore` are member functions.

```cpp
    cin.get (someChar);
    cin.ignore (100, ‘\n’);
```

• These statements declare `myInfile` as an instance of class ifstream and invoke member function `open`.

```cpp
    ifstream   myInfile  ;
    myInfile.open ( “mydata.dat” ) ;
```
Scope Resolution Operator ( :: )

• C++ programs typically use several class types.

• Different classes can have member functions with the same identifier, like Write().

• Member selection operator is used to determine the class whose member function Write() is invoked.

```cpp
    currentDate.Write( ) ; // class DateType
    numberZ.Write( ) ; // class ComplexNumberType
```

• In the implementation file, the scope resolution operator is used in the heading before the member function’s name to specify its class.

```cpp
    void DateType :: Write ( ) const
    {
        . . .
    }
```
Inheritance
A Short Review of Object-Oriented Programming

- Three inter-related constructs: classes, objects, and inheritance
- Objects are the basic run-time entities in an object-oriented system.
- A class defines the structure of its objects.
- Classes are organized in an “is-a” hierarchy defined by inheritance.
Inheritance

1. Allows programmers to create a new class that is a specialization of an existing class.
2. The new class is called a derived class of the existing class; the existing class is the base class of the new class.
Inheritance

- **Inheritance** fosters reuse by allowing an application to take an already-tested class and derive a class from it that inherits the properties the application needs.

- **Polymorphism**: the ability of a language to have duplicate method names in an inheritance hierarchy and to apply the method that is appropriate for the object to which the method is applied.
Inheritance and polymorphism combined allow the programmer to build useful hierarchies of classes that can be reused in different applications.

Mapping of problem into solution.
```cpp
#include <string>
class MoneyType
{
public:
    void Initialize(long, long);
    long DollarsAre( ) const;
    long CentsAre( ) const;
private:
    long dollars;
    long cents;
};
```
class ExtMoneyType: public MoneyType
{
public:
    string CurrencyIs();
    void Initialize(long, long, const string);
private:
    string currency;
};

ExtMoneyType extMoney;
void ExtMoneyType::Initialize
    (long newDollars, long newCents, string newCurrency)
{
    currency = newCurrency;
    MoneyType::Initialize(newDollars, newCents);
}

String ExtMoneyType::CurrencyIs() const
{
    return currency;
}
Exceptions

• An exception is an unusual situation that occurs when the program is running.

• Exception Management
  • Define the error condition
  • Enclose code containing possible error (*try*).
  • Alert the system if error occurs (*throw*).
  • Handle error if it is thrown (*catch*).
try, catch, and throw

Try
{
    // code that contains a possible error
    ... throw string("An error has occurred in function ...");
}

Catch (string message)
{
    std::cout << message << std::endl;
    return 1;
}
try
{
    infile >> value;
    do
    {
        if (value < 0)
            throw string("Negative value");
        sum = sum + value;
    } while (infile);
}
catch (string message)
// Parameter of the catch is type string
{
// Code that handles the exception
    cout << message << " found in file. Program aborted.";
    return 1;
}
// Code to continue processing if exception not thrown
    cout << "Sum of values on the file: " << sum;
namespace mySpace
{
    // All variables and
    // functions within this
    // block must be accessed
    // using scope
    // resolution operator (::).
}

Purpose: Avoid namespace pollution.
Three Ways to Access Members within a Namespace

• **Qualify each reference:**
  
  `mySpace::name` with every reference.

• **Using declaration:**
  
  `using mySpace::name;`
  
  All future references to `name` refer to `mySpace::name`.

• **Using directive:**
  
  `using namespace mySpace;`
  
  All members of `mySpace` can be referenced without qualification.
Rules for Use of Namespace std
(within text)

• Qualify names in prototypes and/or function definitions.
• If name used more than once in a function block, use a using declaration.
• If more than one name is used from a namespace, use a using directive.
Map to Joe’s Diner
Which Cost More to Feed?
Order of Magnitude of a Function

The order of magnitude, or Big-O notation, of a function expresses the computing time of a problem as the term in a function that increases most rapidly relative to the size of a problem.
Names of Orders of Magnitude

\[ O(1) \] bounded (by a constant) time

\[ O(\log_2 N) \] logarithmic time

\[ O(N) \] linear time

\[ O(N \cdot \log_2 N) \] \(N\cdot\log_2 N\) time

\[ O(N^2) \] quadratic time

\[ O(2^N) \] exponential time
<table>
<thead>
<tr>
<th>N</th>
<th>log₂N</th>
<th>N*log₂N</th>
<th>N²</th>
<th>2ᴺ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1.0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4.0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>16.0</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>24</td>
<td>64.0</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>64</td>
<td>256.0</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>160</td>
<td>1024.0</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>384</td>
<td>4096.0</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>896</td>
<td>16,384.0</td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Algorithms
Find “John Smith”
# Big-O Comparison of List Operations

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>UnsortedList</th>
<th>SortedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetItem</td>
<td>O(N)</td>
<td>O(N) linear search</td>
</tr>
<tr>
<td>PutItem</td>
<td>O(1)</td>
<td>O(N) binary search</td>
</tr>
<tr>
<td>Find</td>
<td>O(1)</td>
<td>O(N) search</td>
</tr>
<tr>
<td>Put</td>
<td>O(1) swap</td>
<td>O(N) moving down</td>
</tr>
<tr>
<td>Combined</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>DeleteItem</td>
<td>O(N)</td>
<td>O(N) search</td>
</tr>
<tr>
<td>Find</td>
<td>O(N)</td>
<td>O(N) moving up</td>
</tr>
<tr>
<td>Put</td>
<td>O(1) swap</td>
<td>O(N)</td>
</tr>
<tr>
<td>Combined</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>