DATA DESIGN AND IMPLEMENTATION

After studying this chapter, you should be able to

- Describe an ADT from three perspectives: the logical level, the application level, and the implementation level
- Explain how a specification can be used to represent an abstract data type
- Describe the component selector at the logical level, and describe appropriate applications for the C++ built-in types: structs, classes, one-dimensional arrays, and two-dimensional arrays
- Declare a class object
- Implement the member functions of a class
- Manipulate instances of a class (objects)
- Define the three ingredients of an object-oriented programming language: encapsulation, inheritance, and polymorphism
- Distinguish between containment and inheritance
- Use inheritance to derive one class from another class
- Use the C++ exception handling mechanism
- Access identifiers within a namespace
- Explain the use of Big-O notation to describe the amount of work done by an algorithm

What is Data?

- Data are the nouns of the programming world:
  - The objects that are manipulated
  - The information that is processed
- **Data abstractions** separate our *logical* view of data from the computer’s *implementation* view

Data Abstraction

- Logical view:
  - What are the possible values?
  - What are the operations on this data?
- Implementation view:
  - How is this data used?
  - How is it stored in memory?
  - How can it be implemented in C++?

Data Encapsulation

- **Encapsulation** separates the representation of data from applications that use the data at the logical level
- The physical representation is hidden behind an interface for interacting with the data
- **Abstract Data Type:** A data type whose operations and domain of values are specified independently of any implementation

Data Encapsulation (cont.)

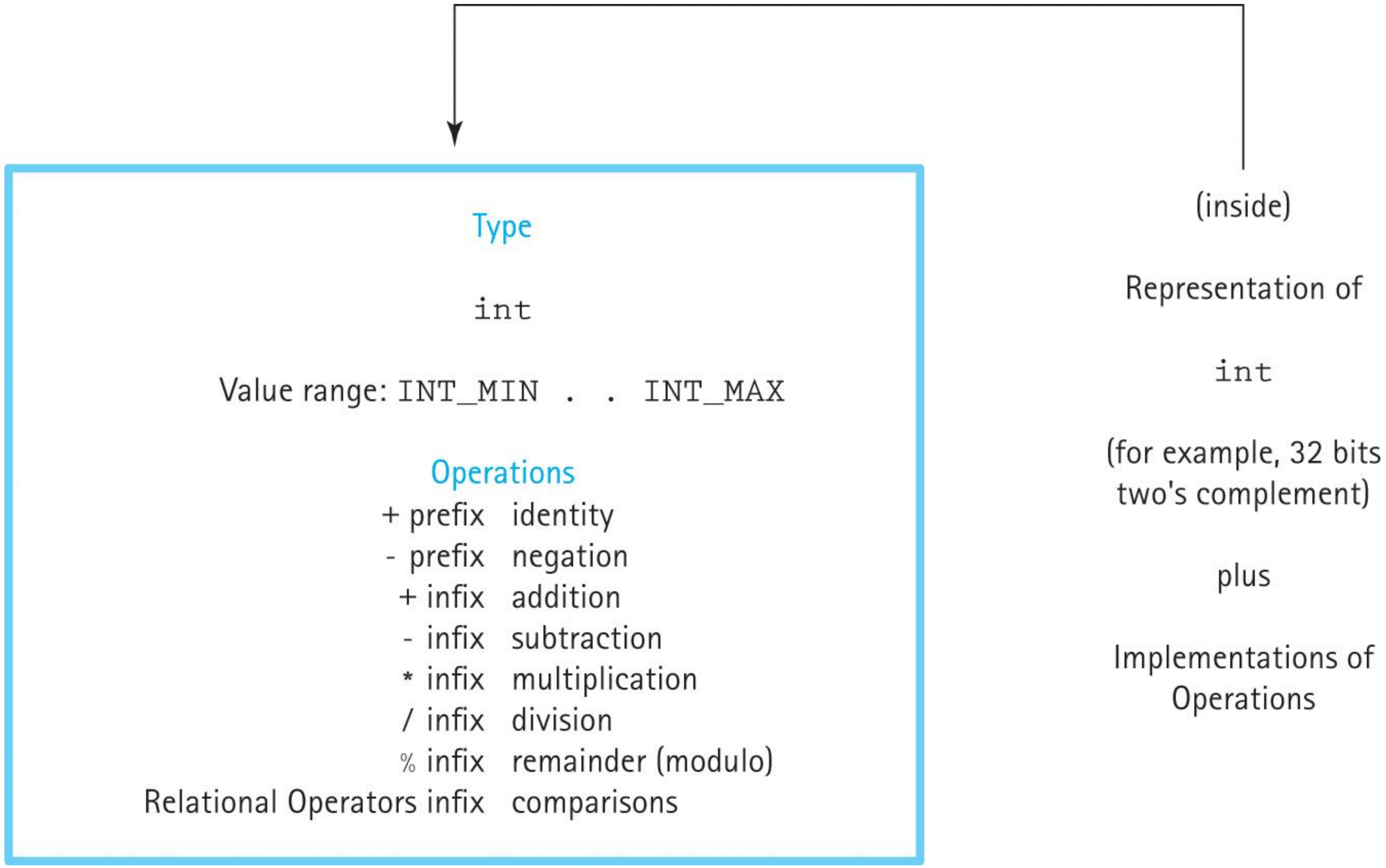


Figure 2.1 A black box representing an integer

Data Structures

- A collection of data elements with operations that store and manipulate individual elements
  - Can be decomposed into individual elements
  - The arrangement of elements in the structure is significant
  - Arrangement and access of elements can be encapsulated
- Used to implement ADTs

Example: Library as an ADT

- A library’s data elements are the books
- ADT interface: Users can check books in or out, reserve books, and pay fines
- Data structure: Can order books randomly, alphabetically by title, or use the Dewey Decimal System
- Users don’t need to know how the library organizes the books

Data From Different Levels

- **Application (user) level:** The problem domain; modeling real-life data in the problem's context
- **Logical (abstract) level:** Abstract view of data values and operations on data
  - "What" questions: What do we do to the data?
- **Implementation level:** Specific representation of data in the program
  - "How" questions: How do we implement the ADT?

Data Levels of a Library

- **Application level:** Library of Congress or Baltimore County Public Library
- **Logical level:** The domain is a collection of books
  - Operations include: check a book out, check a book in, pay a fine, reserve a book
- **Implementation level:** Representation of the book data structure to hold the library’s data, and the coding for operations

Application and Implementation

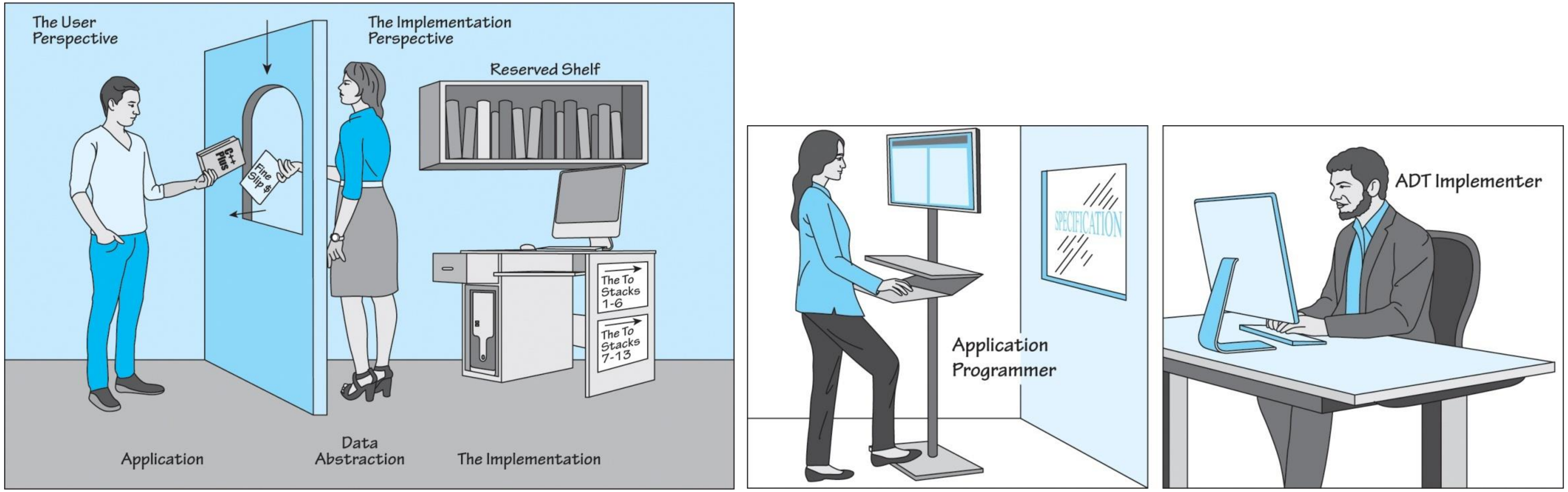


Figure 2.3 Communication between the application level and implementation level

ADT Operator Categories

Operations on ADTs can be classified as:

- **Constructors:** Create new instances (objects) of an ADT
- **Transformers:** Change the internal state of the object
- **Observers:** View the state of the object - Observers come in several forms: predicates, accessors, and summaries
- **Iterators:** For the sequential processing of elements

Composite Data Type

- A **composite data type** stores a collection of individual data components under one name and allows access to individual components
- Two forms:
  - **Unstructured:** Components are not organized with respect to each other (e.g., classes)
  - **Structured:** Components are organized and it affects how they are accessed (e.g., arrays)

Records: Logical Level

- **Record:** A finite collection of elements that are called *members* or *fields*
- Members are accessed using named selectors, such as mystruct.fieldname
  - Can also be used to assign values to fields

Records: Application Level

- Collect related data in one structure
- Used to implement other data structures

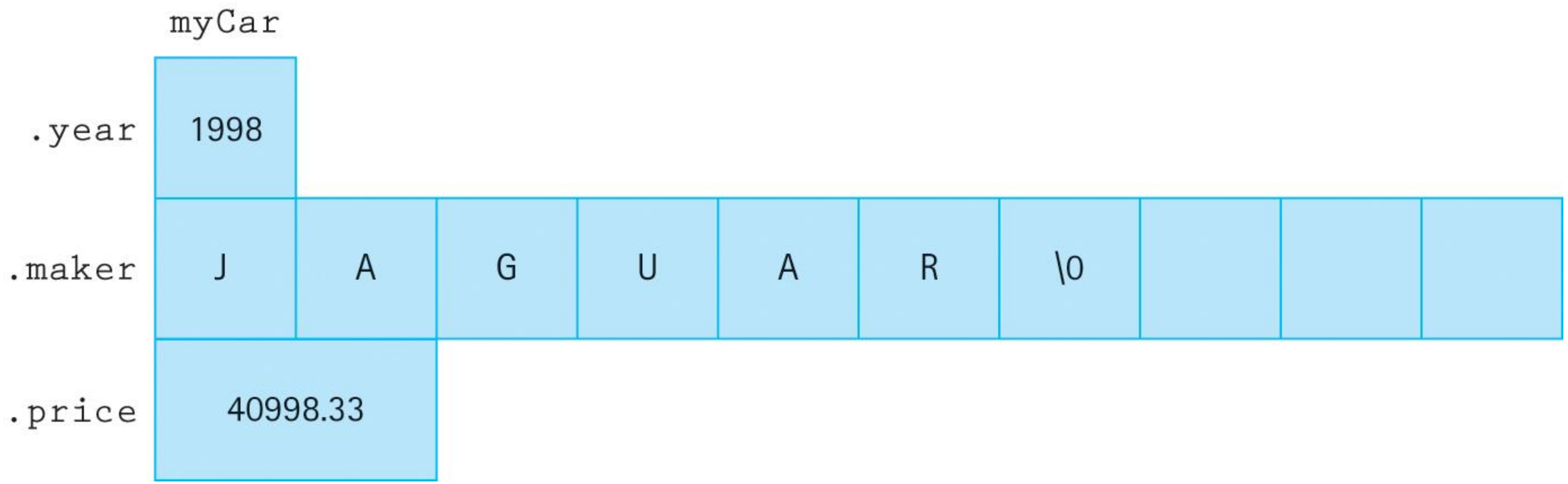


Figure 2.4 Record myCar

Records: Implementation Level

- Records occupy a contiguous chunk of memory
- Each member is located at an offset from the beginning of the record
- Calculating offsets is handled automatically by the compiler and run-time system
- The implementation is structured even though the logical view is not

Implementation of a CarType

Member	*Length*	*Offset*
year	1	0
maker	10	1
price	2	11

Address	
8500	year member (length=1)
8501	
8502	
⋮	maker member (length=10)
8509	
8510	
8511	price member (length=2)
8512	

Figure 2.6 Implementation-level view of CarType

Passing Records to Functions

C++ supports two ways of passing arguments:

- **By value:** A copy of the argument is passed; the original argument cannot be modified by the function
- **By reference:** The function receives the memory location of the argument, allowing changes to be made directly

One-Dimensional Arrays: Logical Level

- **Arrays** are finite, fixed-size collections of ordered homogeneous elements
- Permit direct, random access of elements using indices: `myArray[2]` accesses the third element
- Arrays can only be passed by reference
  - Functions can change array contents
  - Can prevent changes by using the `const` keyword

One-Dimensional Arrays: Application Level

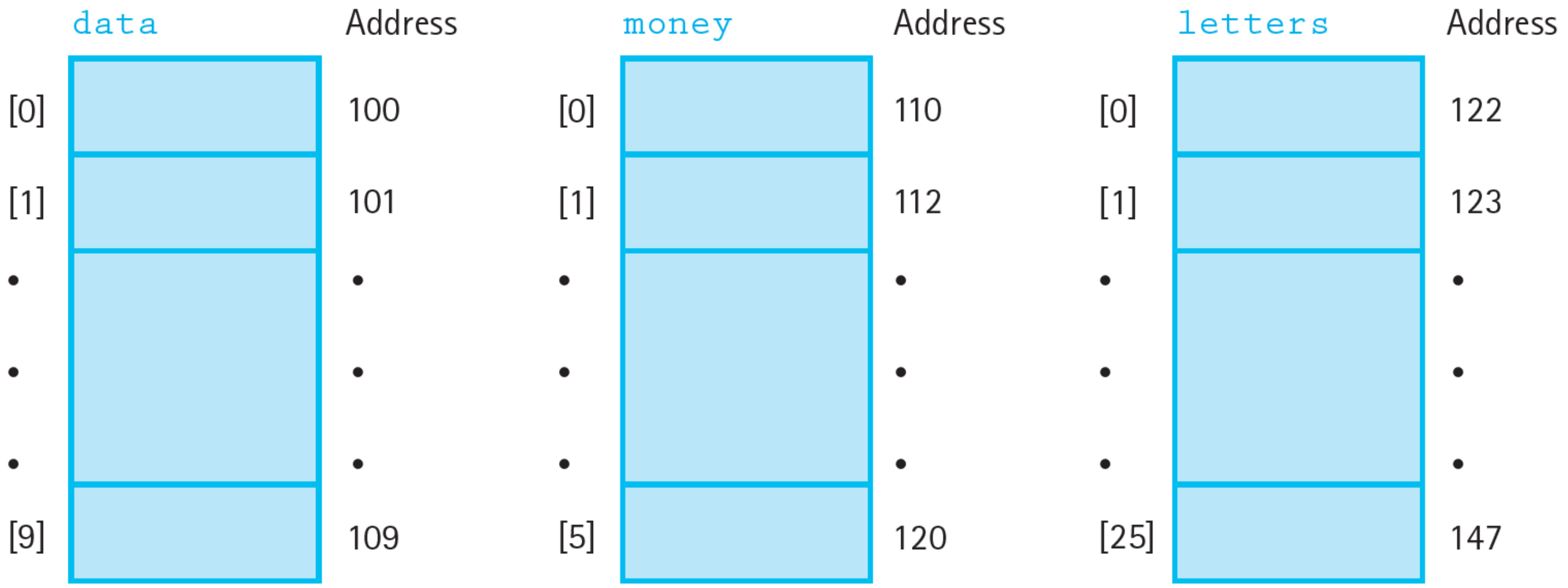
- Used to store lists of data elements
- Strings are arrays of characters
- Arrays must be homogeneous (all one type)
- Operations: Creation and element access

One-Dimensional Arrays: Implementation Level

The *array declaration* describes the name of the array and the type and number of elements

- An array of ten integers: **`int data[10];`**
- To access an element, calculate the offset from the base address (beginning of the array)
- This is automatically handled by the compiler:
- Address = base + index * size of element

One-Dimensional Arrays: Memory Layout



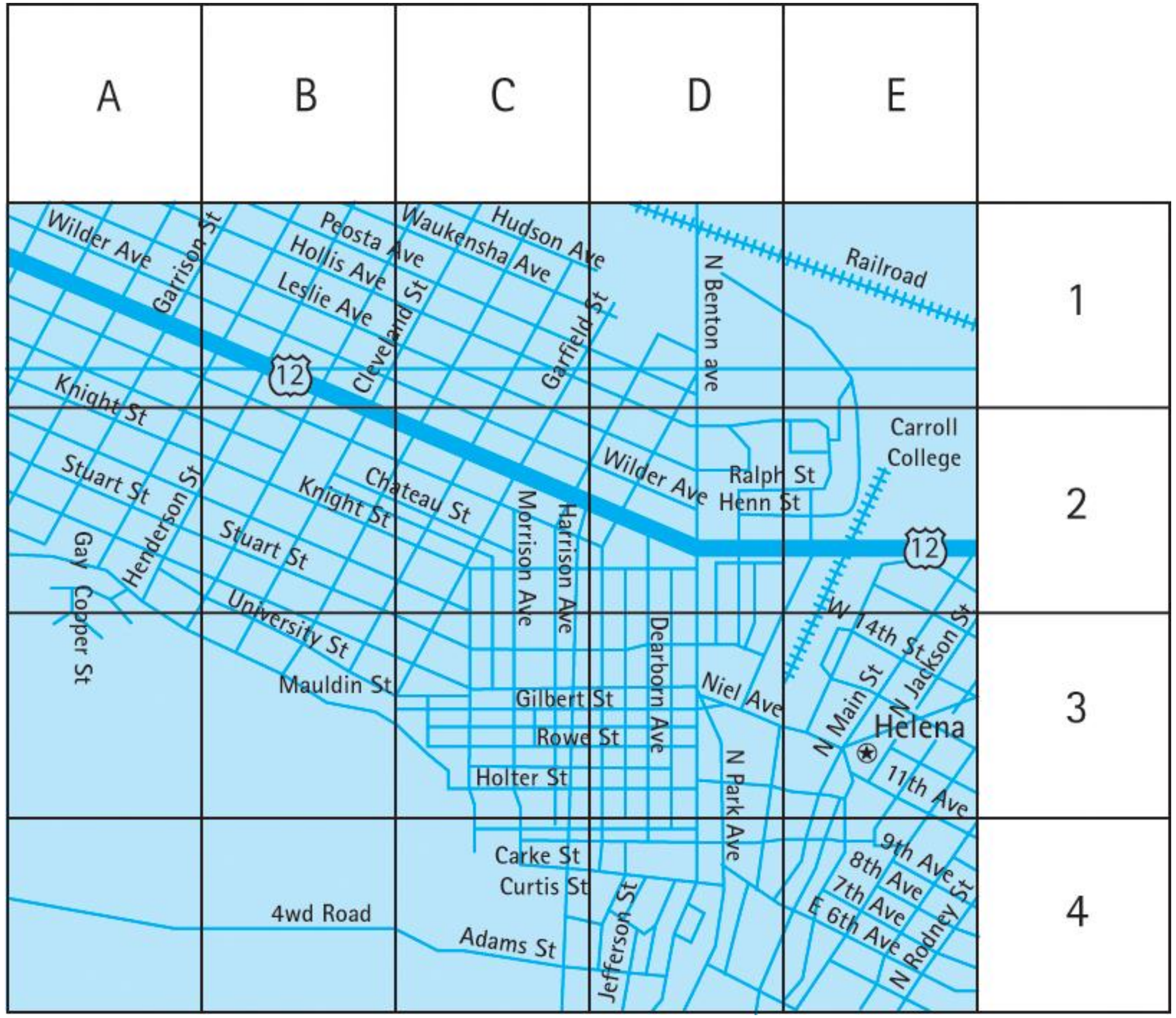
Two-Dimensional Arrays: Logical Level

- Finite, fixed-size, ordered collections of homogeneous array elements
  - Very similar to one-dimensional arrays
- Can be thought of as a table of rows and columns: `table[row][col]`
- When used as parameters, must include the size of the second dimension:
- `int ProcessValues(int values[][5])`

Two-Dimensional Arrays: Application Level

- Ideal for representing tables of data
- Operations: Creation and element access
- Primarily used to implement other data structures
- Can be thought of as an array of arrays, or as a two-dimensional matrix

Map as a 2-D Array:



Two-Dimensional Arrays: Implementation Level

- Stored as a single contiguous piece of memory
- **Row-major order:** Two-dimensional arrays in C++ are stored row-by-row
  - An NxM array uses (N*M*element size) memory
  - The first M cells are the first row, and so on
  - In *column-major order*, the first N cells would be the first column

C++ Class Type

- An unstructured type that encapsulates a fixed number of data components along with functions that manipulate them
- Operations: Member access and whole assignment
  - More operations can be defined per class

Using Classes

- A variable of a class type is called an *object* or *instance* of the class
- Software that uses instances of a class is called a *client*
- Clients must have #include “ClassName.h” in order to be able to use ClassName objects
- Clients can only access public members of an object

Class Specification

- Describes data fields and functions of the class
- Typically in its own file with “.h” extension
- The implementation resides in a separate file
  - Allows for a cleaner interface
  - Programmers can focus on designing a class without worrying about implementation details
  - Can easily change the implementation without touching the interface

Class Implementation

- Must define all constructors and methods
- Use the **scope resolution operator** “::” to indicate a function implementation belongs to a class: **`int DateType::GetMonth() const`**

Using “Self”

- Don’t need to qualify member names in class functions
  - e.g., If a class has a member “foo,” then the name “foo” in the function refers to the object’s field
- The “self” keyword can be used to refer to the object on which the member function is called
  - e.g., In today.GetMonth(), “self” points to today

Classes vs. Structs

- Both are unordered collections of members
- **Structs:** All members *public* by default, typically used as basic container of related data, operations on data are performed by passing struct to global functions
- **Classes:** All members *private* by default, operations on data are defined as member functions

Inheritance

- Classes can **inherit** properties (data and methods) from other classes
  - Base class: The class being inherited from
  - Derived class: The class that inherits
- Derived classes are more specialized and typically have more fields
- Inheritance creates a hierarchy of classes
- Can be viewed as an “is-a” relationship

Inheritance Hierarchy

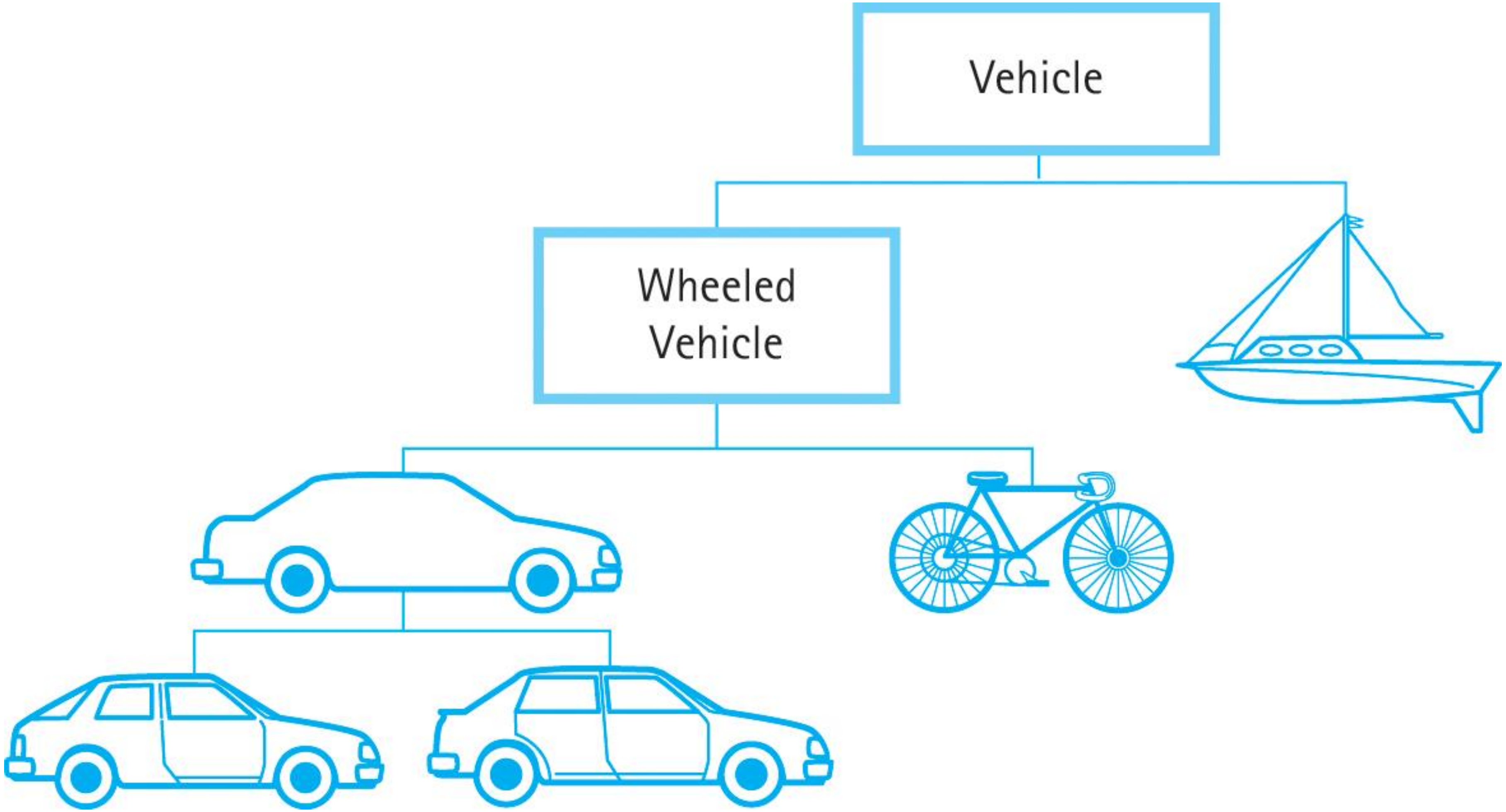


Figure 2.7: Inheritance hierarchy

Polymorphism

- **Overloading:** When multiple functions have the same name
  - They must have unique signatures, such as different numbers or types of parameters
- **Polymorphism:** The ability to statically or dynamically determine which version of an overloaded function to use

Binding

- **Binding Time:** The time at which a name or symbol is bound to some code
- **Static Binding:** Occurs at compile time
- **Dynamic Binding:** Occurs during run time
- Polymorphism can use either kind of binding when determining which function to use

C++ Constructs for OOP

- **Composition:** A class contains an object of another class type; also called *containment*
- **Inheritance:** In the form of
- **`class Derived : public Base`**
- **Virtual Methods:** Allow for dynamic binding, used to implement polymorphism

Exceptions

- Enable programs to gracefully handle exceptional conditions
- *Try-catch* statement: *try* clause protects code that can cause exceptions, *catch* clause is executed if an exception is thrown
  - Keyword *throw* is used to trigger exceptions
- C++ standard library has many predefined exception classes

Try-Catch Block

```
try {
  // code that contains a possible error
  // when the error occurs:
  throw string("An error has occurred…");
}
catch (string message) {
  // execution continues here
  std::cout << message << std::endl;
  return 1;
}
```

Exceptions in Code

```
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;
```

Comparison of Algorithms

- How do we compare the efficiency of different algorithms?
- Comparing execution time: Too many assumptions, varies greatly between different computers
- Compare number of instructions: Varies greatly due to different languages, compilers, programming styles...

Big-O Notation

- The best way is to compare algorithms by the amount of work done in a critical loop, as a function of the number of input elements (*N*)
- **Big-O:** A notation expressing execution time (complexity) as the term in a function that increases most rapidly relative to *N*
- Consider the *order of magnitude* of the algorithm

Common Orders of Magnitude

- O(1): Constant or *bounded* time; not affected by *N* at all
- $O(\log_2 N)$: Logarithmic time; each step of the algorithm cuts the amount of work left in half
- $O(N)$: Linear time; each element of the input is processed
- $O(N \log_2 N)$: $N \log_2 N$ time; apply a logarithmic algorithm N times or vice versa

Common Orders of Magnitude (cont.)

- $O(N^2)$: Quadratic time; typically apply a linear algorithm N times, or process every element with every other element
- $O(N^3)$: Cubic time; naive multiplication of two NxN matrices, or process every element in a three-dimensional matrix
- $O(2^N)$: Exponential time; computation increases dramatically with input size

What About Other Factors?

- Consider $f(N) = 2N^4 + 100N^2 + 10N + 50$
- We can ignore $100N^2 + 10N + 50$ because $2N^4$ grows so quickly
- Similarly, the 2 in $2N^4$ does not greatly influence the growth
- The final order of magnitude is $O(N^4)$
- The other factors may be useful when comparing two very similar algorithms

Elephants and Goldfish

- Think about buying elephants and goldfish and comparing different pet suppliers
- The price of the goldfish is trivial compared to the cost of the elephants
- Similarly, the growth from $100N^2 + 10N + 50$ is trivial compared to $2N^4$
- The smaller factors are essentially noise

Example: Phone Book Search

- Goal: Given a name, find the matching phone number in the phone book
- Algorithm 1: Linear search through the phone book until the name is found
- Best case: O(1) (it's the first name in the book)
- Worst case: O(N) (it's the final name)
- Average case: The name is near the middle, requiring N/2 steps, which is O(N)

Example: Phone Book Search (cont.)

Algorithm 2: Since the phone book is sorted, we can use a more efficient search

1) Check the name in the middle of the book
2) If the target name is less than the middle name, search the first half of the book
3) If the target name is greater, search the last half
4) Continue until the name is found

Example: Phone Book Search (cont.)

Algorithm 2 Characteristics:

- Each step reduces the search space by half
- Best case: O(1) (we find the name immediately)
- Worst case: $O(\log_2 N)$ (we find the name after cutting the space in half several times)
- Average case: $O(\log_2 N)$ (it takes a few steps to find the name)

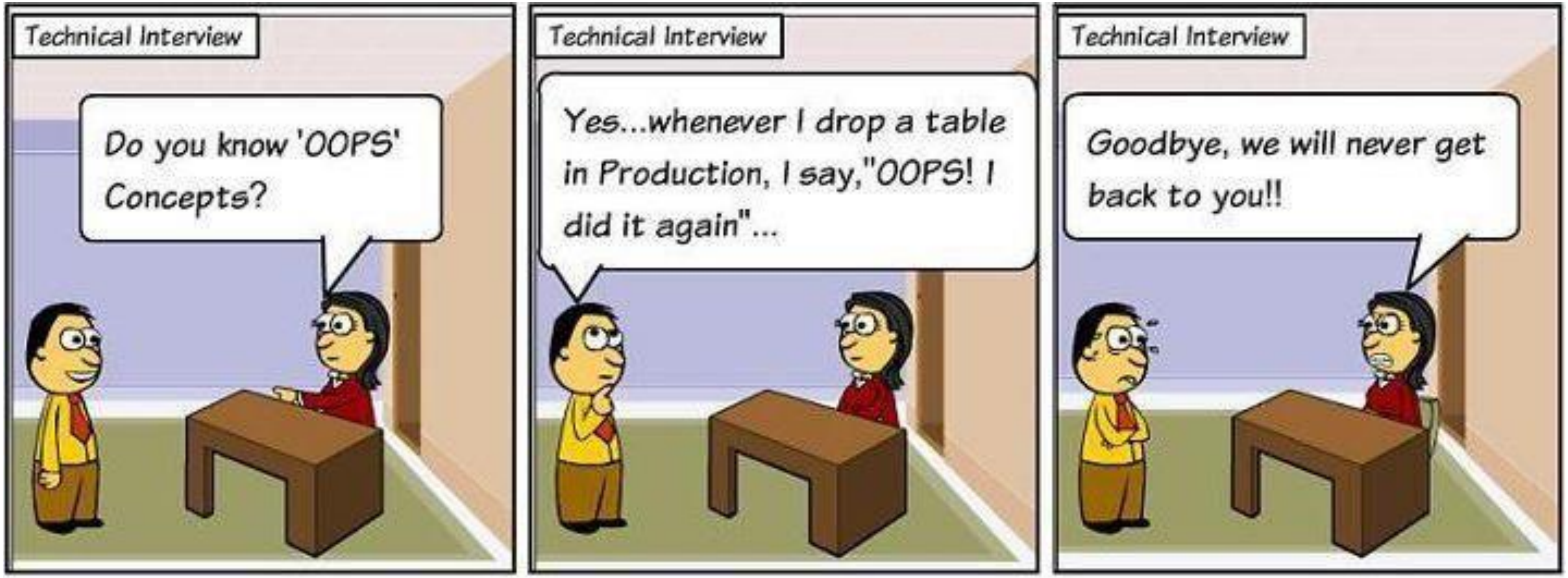
Example: Phone Book Search (cont.)

Which algorithm is better?

- For very small *N*, algorithm may be faster
- For target names in the very beginning of the phone book, algorithm 1 can be faster
- Algorithm 2 will be faster in every other case
- Success of algorithm 2 relies the fact that the phone book is sorted

Data structures matter!

The End!



Technical Interview
Do you know 'OOPS' Concepts?
Technical Interview
Yes...whenever I drop a table in Production, I say,"OOPS! I did it again"...
Technical Interview
Goodbye, we will never get back to you!!