Week 6: Data

- criteria for external and internal representation
- basic elementary data types
- composite data types
- container data types
- derived subtypes
- abstract data types (ADT)
- programming language support

Week 6: Data

- criteria for external and internal representation
- basic elementary data types
- composite data types
- container data types
- derived subtypes
- abstract data types (ADT)
- programming language support

COMP 370, Fall, 2014
Mr. Weisert

Computer programs manipulate two kinds of data

- Application domain data
  - Exist in the real world
  - Usually known to users / sponsors
  - May be persistent or transient

- Program data
  - Have no real-world existence
  - None of the users' business
  - Usually transient

We're going to focus first on application domain data.

Part 1: Choosing data representations

- Who does it? When?
- What are some criteria?
- Three examples
  - color (of an object)
  - name (or a person)
  - weight (of a shipment)

Example 1: representing color

<table>
<thead>
<tr>
<th>Color</th>
<th>Choice A</th>
<th>Choice B</th>
</tr>
</thead>
<tbody>
<tr>
<td>white</td>
<td>&quot;WHT&quot;</td>
<td>1</td>
</tr>
<tr>
<td>black</td>
<td>&quot;BLK&quot;</td>
<td>2</td>
</tr>
<tr>
<td>red</td>
<td>&quot;RED&quot;</td>
<td>3</td>
</tr>
<tr>
<td>blue</td>
<td>&quot;BLU&quot;</td>
<td>4</td>
</tr>
<tr>
<td>green</td>
<td>&quot;GRN&quot;</td>
<td>5</td>
</tr>
<tr>
<td>aquamarine</td>
<td>&quot;AQM&quot;</td>
<td>6</td>
</tr>
</tbody>
</table>

Which is better? Why?

What other choices should we consider?
Consider these code fragments

a. if (color == "WHT")
   ++whiteCount;
else if (color == "BLK")
   ++blackCount;
else if (color == "GRN")
   ++greenCount;
else if (color == "BLU")
   ++bluecount;
else if (color == "RED")
   ++redcount;

b. ++itemCount[color];

Example 2: representing a person's name

First choice: a 32-byte field with embedded delimiter
Limbaugh, Rush
De Gaulle, Chuck
Quayle, J. Danforth

Which is best? Why?

Second choice: a 32-byte field
Rush Limbaugh
Chuck De Gaulle
J. Danforth Quayle

What other choices should we consider?

Third choice: three 15-byte fields
Chuck
J.
Danforth
De Gaulle
Quayle

Name representation considerations
- What if we want to sort the names?
- What if someone has more than three names?
  George Herbert Walker Bush
- What if someone has only one name?
  Madonna
- What if some cultures put the family name first?
  Chiang Kai-Shek
- What if one part of the name is very long?

Example 3: representing shipping weight

First choice:
- unit of measure: kilograms
- range: 0.1 - 50
- precision: .05
- scale: float

Which is better? Why?

Second choice:
- unit of measure: <pounds, ounces>
- range: <0,8> - <99,15>
- precision <0,1>
- scale integer, integer

What other representations should we consider?
What do these examples show?

- End users prefer representations that are **familiar**.
- Programmers prefer representations that are **simple**.
- Therefore a computer application often needs two (or more) representations of the same data item.

Data representation -- 2 kinds

- **External** representations are *visible* to human users. They appear in:
  - printed reports
  - screen displays
  - transactions and other input data
  - source documents
- **Internal** representations (rarely seen by the end user) appear in:
  - computer programs
  - data bases / master files
  - temporary (work / scratch) files

*Criteria for choosing each are quite different.*

Criteria for choosing a data representation

- **External** representations should be:
  - familiar to the people who enter them or have to understand them.
  - reliable; not error-prone
- **Internal** representations should be:
  - simple and efficient to store and manipulate
  - flexible to accommodate growth or likely future changes
  - (if possible) interchangeable among application systems, organizations, etc. (i.e. *standard*)

How do programs convert between the two representation?

- Internal to external
- External to internal
- When do these conversions ideally occur?
**Data Conversion:**

**external to internal (input)**
- usually done by an editing program, which:
  1. **converts** each data item from its external (input) representation to a standard internal representation, so that later programs needn't know about or deal with messy external representations
  2. **validates** that each data item conforms to its specification. It rejects those that fail, so that subsequent programs can never encounter illegal data.
- Thorough input editing is essential in any program that accepts input from the external world, whether batch, interactive, event-driven, ...

**Data Conversion:**

**internal to external (output)**
- simpler than input  **Why?**
- usually done in a **reporting** or output display program
- In C++ can be implemented as a customized *output stream insertion* operator (```<<```)
  or a class-specific **print** function (method).
- In Java and C# can be implemented in the **toString()** or **ToString()** method.

---

**Any Questions?**

- About data representation in general?
- About the need to distinguish between
  - internal representation and
  - external representation
- The criteria for choosing both

  **This is an important topic that's not covered in many textbooks.**

---

**How can object-oriented programming help in distinguishing between internal and external representations?**

- **Member data** items (usually private) constitute the internal representation
- Output-stream functions and **toString()** handle internal to external conversion
- **Constructors** handle external to internal conversion (sometimes not very well)
Part 2: A taxonomy of data types

- The 3 fundamentally different kinds of data
- Some basic subtypes of those 3.

This is all independent of any programming language.

We talked about some of this (numeric data) in week 3

Data items: 3 kinds

- **Elementary** items are
  - not composed of other data items
  - sometimes called "fields" or "elements" (when part of a composite item).
  - defined in terms of their real world meaning
- **Composite** items are:
  - composed of other data items, which may be either elementary or composite,
  - sometimes called "structures", "records", "blocks", "data flows",
  - also called "entities" or "subjects" when they play a primary role in an application system
  - defined mainly in terms of their components.

Data items: 3 kinds

- **Container** items are
  - structures that can hold other data items, either elementary or composite (or sometimes other containers).
  - either:
    - static (staying the same size and shape throughout their life span), or
    - dynamic (growing, shrinking, or reconfiguring)
  - either:
    - homogeneous (all elements are of the same type), or
    - heterogeneous (multiple kinds of data can be elements)
  - defined in terms of their behavior

Elementary data items

- Every *elementary* data item belongs to one (and only one) of these basic types.
  - **discrete** (or coded or enumerated)
    possible values belong to a finite set
  - **numeric**
    some arithmetic operation is meaningful
  - **text** (or character string)
  - **logical** (or Boolean)
    2 possible values

 Which basic types did the three earlier examples belong to?
Elementary data items

- Every elementary data item belongs to one (and only one) of these basic types.
  - discrete (or coded or enumerated)
    possible values belong to a finite set
  - numeric
    some arithmetic operation is meaningful
  - text (or character string)
  - logical (or Boolean)
    2 possible values

What support is built into your favorite programming language for these basic types?

Attributes (or properties) of elementary data items

- Attributes of numeric data items:
  - unit of measure
  - range
  - precision
  - scale
- Attributes of text data items:
  - length
  - internal format, delimiters
- Attributes of discrete data items:
  - number of possible values
    (both current and possible)
  - coding structure

Avoid false numerics

- Many discrete data items are represented by a sequence of numeric digits.
  Examples?
- That doesn't make them numeric.
  Why not?
- We emphasize what a data item is, not what it looks like. Some old-fashioned tools take the opposite view.
  Which ones?

Examples of container data

- Static structures
  - Arrays
- Dynamic structures:
  - lists, stacks, queues
  - trees, graphs
  - dynamic arrays
  What about character strings?
- External structures:
  - files, data bases
  - display / interface (GUI) objects
Terminology update

- Java uses the term **container** in a narrower sense, to mean a GUI screen object (e.g. a window) that can contain other GUI objects.
- Java now uses the term **collection** in the more general sense, where we’ve been using **container**.
- Use whichever term you prefer as long as the context is clear.

Taxonomy summary:

- **Data item**
  - **Elementary item**
    - Discrete
      - COLOR (of a product)
      - ACCOUNT NUMBER
      - STATE (in an address)
      - MARRITAL STATUS
      - BRANCH OFFICE CODE
      - CREDIT CARD TYPE
      - TELEPHONE NUMBER
    - Text
      - NAME (of an employee)
      - DESCRIPTION (of a product)
      - CITY (in an address)
      - DUNNING LETTER
    - Logical
      - CREDIT APPROVAL (for an order)
      - NEW CUSTOMER FLAG
      - UNION MEMBER (for an employee)
      - AUDIT TRACE OPTION
  - **Composite item**
    - Records, entities
    - Other static tree structures
    - Arrays, vectors, matrices, tables
    - Lists, stacks, queues
    - Trees, Graphs
    - Files, databases
  - **Container item**
    - Arrays, vectors, matrices, tables
    - Lists, stacks, queues
    - Trees, Graphs
    - Files, databases
    - Windows, boxes

Which ones are suited to being represented as object-oriented classes?

Avoid false composites

- Don’t confuse a *mixed unit or structured representation* of an elementary item with a true composite item.
- For example, these should be treated as elementary, not composite, items:
  - **Date** = year + month + day
  - **Person Name** = first + middle + last
  - **Time** = hours + minutes + seconds

Avoiding false composites is especially important to avoid clutter in a data dictionary.

Examples

- **Elementary items**
  - Discrete
    - COLOR (of a product)
    - ACCOUNT NUMBER
    - STATE (in an address)
    - MARRITAL STATUS
    - BRANCH OFFICE CODE
    - CREDIT CARD TYPE
    - TELEPHONE NUMBER
  - Text
    - NAME (of an employee)
    - DESCRIPTION (of a product)
    - CITY (in an address)
    - DUNNING LETTER
  - Logical
    - CREDIT APPROVAL (for an order)
    - NEW CUSTOMER FLAG
    - UNION MEMBER (for an employee)
    - AUDIT TRACE OPTION
  - **Composite items**
    - HOME ADDRESS (of an employee)
    - CREDIT HISTORY (of a customer)
    - SUBASSEMBLY (of a product)
    - CURRENT ORBIT (of a satellite)

- **Other**
Data classes and inheritance

- Each basic data type can be divided into subtypes or data classes.
- Each such class can in turn be further divided into subclasses to any level.

Classes and data items

- In a class hierarchy, properties of a class T are inherited by both:
  - subclasses of T
  - specific data items (instances, objects) of type T
- It is, therefore, unnecessary and undesirable (why?) to specify attributes in the definition of every data item.
  - Nevertheless, many older tools (COBOL, some data-dictionary systems, etc.) demand that we do so!
  - Would the "year-2000 crisis" have occurred, if everyone had understood this principle?

Data classes and inheritance

- Each class inherits the properties of its parent classes:
  - data representation, esp. internal, and attributes
  - associated functions and operations
- This inheritance principle can greatly simplify the definition of both data items and the functions or processes that operate on them (even without any object-oriented tools.)

Classes (or types) versus data items (or instances or objects)

- Inexperienced systems analysts sometimes confuse these two very different things.
- For example, which (class or data item) is each of these likely to be?
  - temperature
  - date
  - address
- How do we know?
**Data representation standards**

- For any class, there's usually a big advantage in choosing the same internal representation for all instances (data items). That promotes:
  - data interchange
  - re-usable program modules
  - reliability
  - efficiency

- Level can be
  - a generic class or subclass
  - a basic type

- Scope can be:
  - industry-wide
  - corporate
  - project

**Abstract data type (ADT)**

- A class defined:
  - not in terms of the representation
  - but in terms of the operations or functions that are permitted / supported ("behavior", "services")

- When we discuss or manipulate an ADT, we:
  - are unaware of the internal representation, including names, types, or sequence, or memory layout of any component items
  - interact with objects of that type only through well-defined interface functions ("methods")

  *Why is that good?*

---

**Implementing an ADT in software**

- Program designers can always (and often should) exploit the ADT concept as a design aid, even with non-OOP tools, but

- Object-oriented languages (Smalltalk, C++, Java, ...) support ADT's through a class definition capability.
  - Hides (i.e. localizes) the internal representation
  - Encapsulates functions that operate on objects (data items, instances) of the class

  *What C facility was extended to support ADT's in C++?*